SECTION 1 — Java Basics

---

1. Introduction to Programming

Definition:
Programming is the process of writing instructions for a computer to perform specific tasks.
A program is made of statements that tell the computer **what to do** and **how to do it**.

Example:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Programming!");
    }
}
```

Tip: The `main()` method is where every Java program starts running.

---

2. Introduction to Java

Definition:
Java is a **high-level**, **object-oriented**, **platform-independent** programming language developed by Sun Microsystems (now owned by Oracle).
Java runs on the **JVM** (Java Virtual Machine) — "Write Once, Run Anywhere."

Example:

```
class Intro {
    public static void main(String[] args) {
        System.out.println("Java is platform
independent!");
    }
}
```

---

3. JDK Installation

Definition:
JDK (Java Development Kit) contains:

- **JRE** (Java Runtime Environment) → To run Java programs
- **Compiler** (`javac`) → To convert `.java` to `.class`
- Development tools

Steps:

1. Download from Oracle
2. Install & set **JAVA_HOME** environment variable
3. Verify with:

```
java -version
javac -version
```

---

4. Keywords, Identifiers, Variables

- **Keywords** → Reserved words in Java (e.g., `class`, `if`, `static`).
- **Identifiers** → Names given to variables, classes, methods.
- **Variables** → Containers for storing data.

Example:

```
class VariablesDemo {
    public static void main(String[] args) {
        int age = 25; // integer variable
        String name = "Sunil"; // string variable
        System.out.println(name + " is " + age + "
years old.");
    }
}
```

---

5. Operators

Definition:
Symbols that perform operations on variables/values.

- **Arithmetic:** `+`, `-`, `*`, `/`, `%`
- **Relational:** `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Logical:** `&&`, `||`, `!`
- **Assignment:** `=`, `+=`, `-=`
- **Unary:** `++`, `--`

Example:

```
class OperatorsDemo {
    public static void main(String[] args) {
        int a = 10, b = 3;
        System.out.println(a + b); // Arithmetic
        System.out.println(a > b); // Relational
```

```
        System.out.println(a > 5 && b < 5); //
    Logical
    }
}
```

---

6. Methods / Functions

Definition:
A method is a block of code that performs a specific task and can be reused.

Example:

```
class MethodsDemo {
    static int add(int x, int y) {
        return x + y;
    }
    public static void main(String[] args) {
        System.out.println(add(5, 7));
    }
}
```

---

7. Flow Control Statements

- **If-Else**
- **Switch**
- **Loops** (`for`, `while`, `do-while`)
- **Break / Continue**

Example:

```
class FlowDemo {
    public static void main(String[] args) {
        int num = 5;
        if(num > 0) {
            System.out.println("Positive");
        } else {
            System.out.println("Negative");
        }

        for(int i = 1; i <= 3; i++) {
            System.out.println("Count: " + i);
        }
    }}
```

8. Arrays

Definition:
Array is a collection of elements of the same type stored in contiguous memory.

Example:

```
class ArraysDemo {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30};
        for(int num : numbers) {
            System.out.println(num);
        }
    }
}
```

---

9. Strings

Definition:
A `String` is an object that represents a sequence of characters. Strings in Java are **immutable**.

Example:

```
class StringDemo {
    public static void main(String[] args) {
        String name = "Java";
        System.out.println(name.toUpperCase());
        System.out.println(name.length());
    }
}
```

---

10. Interactive Programs in Java using Scanner

Definition:
`Scanner` is a class used to take user input from the console.

Example:

```
import java.util.Scanner;

class ScannerDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
```

```
        String name = sc.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

SECTION 2 — OOPS Concepts

---

1. Classes and Objects

Definition:

- **Class** → Blueprint for creating objects (defines properties & behavior).
- **Object** → Instance of a class with actual values.

Example:

```
class Car {
    String brand;
    void drive() {
        System.out.println(brand + " is
driving!");
    }
}

class Main {
    public static void main(String[] args) {
        Car c1 = new Car();  // Object creation
        c1.brand = "Toyota";
        c1.drive();
    }
}
```

---

2. Object Creation

Objects are created using the `new` keyword:

```
Car c = new Car();
```

Here:

- `Car()` calls the constructor.
- `c` is the reference variable pointing to the object.

3. Reference Variable

Definition: A variable that holds the memory address of an object.

Example:

```
class Person {
    String name;
}
class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.name = "Sunil";
        Person p2 = p1; // Both refer to same object
        p2.name = "Aldo";
        System.out.println(p1.name); // Aldo
    }
}
```

---

4. Global and Local Variables

- **Global (Instance) Variables**: Declared inside class but outside any method.
- **Local Variables**: Declared inside methods.

Example:

```
class Demo {
    int globalVar = 10; // Global

    void display() {
        int localVar = 5; // Local
        System.out.println(globalVar + localVar);
    }
}
```

5. Constructors

Definition: Special method called when an object is created, used to initialize values.
Rules: Same name as class, no return type.

Example:

```
class Student {
    String name;
```

```
    Student(String n) {
        name = n;
    }
    void show() {
        System.out.println("Name: " + name);
    }
}
class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Sunil");
        s1.show();
    }
}
```

---

6. Aggregation

Definition: HAS-A relationship between classes (weaker form of association).

Example:

```
class Address {
    String city;
    Address(String city) { this.city = city; }
}

class Employee {
    String name;
    Address address; // Aggregation
    Employee(String name, Address addr) {
        this.name = name;
        this.address = addr;
    }
}
class Main {
    public static void main(String[] args) {
        Address a1 = new Address("Chennai");
        Employee e1 = new Employee("Sunil", a1);
        System.out.println(e1.name + " lives in " +
e1.address.city);
    }
}
```

---

7. Composition

Definition: Strong HAS-A relationship; the part (contained object) cannot exist without the whole.

Example:

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }
}

class Bike {
    private Engine engine = new Engine(); //
Composition
    void startBike() {
        engine.start();
        System.out.println("Bike started");
    }
}
class Main {
    public static void main(String[] args) {
        Bike b = new Bike();
        b.startBike();
    }
}
```

---

8. Inheritance ★

Definition: Mechanism where one class acquires properties and methods of another using `extends`.

Example:

```
class Animal {
    void sound() { System.out.println("Animal makes
sound"); }
}
class Dog extends Animal {
    void bark() { System.out.println("Dog barks"); }
}
class Main {
```

```
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
        d.bark();
    }
}
```

---

9. Method Overloading

Definition: Multiple methods with the same name but different parameter lists.

Example:

```
class MathOps {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
}
```

---

10. Method Overriding

Definition: Subclass provides a specific implementation of a method already defined in the parent.

Example:

```
class Animal {
    void sound() { System.out.println("Animal
sound"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Dog barks"); }
}
```

---

11. Abstract Classes

Definition: Class that **cannot** be instantiated, may have abstract (unimplemented) methods.

Example:

```
abstract class Shape {
```

```
    abstract void draw();
}
class Circle extends Shape {
    void draw() { System.out.println("Drawing
Circle"); }
}
```

---

12. Interfaces

Definition: Blueprint of a class containing only abstract methods (Java 8+ allows default & static methods).

Example:

```
interface Animal {
    void sound();
}
class Cat implements Animal {
    public void sound() { System.out.println("Meow");
}
}
```

---

13. Typecasting

- **Upcasting:** Subclass → Superclass
- **Downcasting:** Superclass → Subclass

Example:

```
Animal a = new Dog(); // Upcasting
Dog d = (Dog) a;      // Downcasting
```

---

14. JVM Architecture

Definition: JVM is the engine that runs Java bytecode. Main components:

- **Class Loader**
- **Runtime Data Areas** (Heap, Stack, Method Area)
- **Execution Engine** (Interpreter + JIT Compiler)

---

15. Polymorphism

Definition: Ability of one interface to have many implementations (compile-time via overloading, run-time via overriding).

---

16. Abstraction

Definition: Hiding implementation details and showing only the essential features.

---

17. Java Packages

Definition: Group of related classes and interfaces.

Example:

```
package mypack;
public class MyClass {
    public void show() { System.out.println("Hello
from package"); }
}
```

---

18. Access Specifiers

- **public**: Accessible everywhere.
- **protected**: Accessible in same package & subclasses.
- **default**: Same package only.
- **private**: Same class only.

SECTION 3 — Java Built-in Packages and API

---

1. Overview of Java API

Definition:
Java API is a huge collection of pre-written classes and interfaces provided in **packages** like `java.lang`, `java.util`, `java.io`, etc.
You just import and use them instead of writing from scratch.

Example:

```
import java.util.Date;
```

```
class APIDemo {
    public static void main(String[] args) {
        Date now = new Date();
        System.out.println("Today: " + now);
    }
}
```

---

2. Object Class

Definition:
`Object` is the root class of all Java classes. Every class implicitly extends it.
Common methods:

- `toString()` → String representation
- `equals()` → Compares objects
- `hashCode()` → Returns object hash code

Example:

```
class Person {
    String name;
    Person(String name) { this.name = name; }

    public String toString() {
        return "Name: " + name;
    }
}
class Main {
    public static void main(String[] args) {
        Person p = new Person("Sunil");
        System.out.println(p.toString());
    }
}
```

---

3. String, StringBuffer, StringBuilder

- **String:** Immutable sequence of characters.
- **StringBuffer:** Mutable, thread-safe.
- **StringBuilder:** Mutable, faster (not thread-safe).

Example:

```
class StringDemo {
```

```
    public static void main(String[] args) {
        String s = "Java";
        s = s + " Rocks"; // New object created

        StringBuffer sb = new StringBuffer("Hello");
        sb.append(" World"); // Changes same object

        StringBuilder sb2 = new
StringBuilder("Fast");
        sb2.append(" Builder");

        System.out.println(s);
        System.out.println(sb);
        System.out.println(sb2);
    }
}
```

---

4. Exception Handling

Definition:
Process of handling runtime errors to avoid program crash.
Keywords: `try`, `catch`, `finally`, `throw`, `throws`.

Example:

```
class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by
zero");
        } finally {
            System.out.println("Cleanup code");
        }
    }
}
```

---

5. Threads and Multithreading

Definition:

- **Thread:** Smallest unit of execution.
- **Multithreading:** Running multiple threads simultaneously.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " +
getName());
    }
}
class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

---

6. Wrapper Classes

Definition:
Convert primitive types to objects (`int → Integer`, `double → Double`, etc.).

Example:

```
class WrapperDemo {
    public static void main(String[] args) {
        int a = 5;
        Integer obj = Integer.valueOf(a); //
Autoboxing
        int b = obj; // Unboxing
        System.out.println(obj + " " + b);
    }
}
```

---

7. Data Structures

Java provides **Collections Framework**: `List`, `Set`, `Map`, `Queue`.

---

8. Java Collection Frameworks ★

Definition:
Set of interfaces and classes for storing and manipulating groups of objects.

Example:

```
import java.util.*;

class CollectionDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        for(String lang : list) {
            System.out.println(lang);
        }
    }
}
```

---

9. File Handling

Definition:
Reading/writing files using classes from `java.io` or `java.nio`.

Example:

```
import java.io.*;

class FileDemo {
    public static void main(String[] args) throws
IOException {
        FileWriter fw = new FileWriter("test.txt");
        fw.write("Hello Java");
        fw.close();

        BufferedReader br = new BufferedReader(new
FileReader("test.txt"));
        System.out.println(br.readLine());
        br.close();
    }
}
```

---

10. Serialization

Definition:
Process of converting an object into a byte stream to save to a file or send over network.
Deserialization is the reverse.

Example:

```
import java.io.*;

class Student implements Serializable {
    String name;
    Student(String name) { this.name = name; }
}

class Main {
    public static void main(String[] args) throws
Exception {
        Student s = new Student("Sunil");
        ObjectOutputStream oos = new
ObjectOutputStream(new FileOutputStream("data.ser"));
        oos.writeObject(s);
        oos.close();
    }
}
```

---

11. Garbage Collector

Definition:
Automatic memory management in Java that removes unused objects.

Example:

```
class GCExample {
    public static void main(String[] args) {
        GCExample obj = new GCExample();
        obj = null; // Eligible for GC
        System.gc(); // Suggest GC
    }
    protected void finalize() {
        System.out.println("Object is garbage
collected");
    }
}
```

---

12. Encapsulation

Definition:
Wrapping variables and methods into a single unit (class) with restricted access using **private** and public getter/setter.

Example:

```
class Account {
    private int balance = 1000;

    public int getBalance() { return balance; }
    public void setBalance(int amount) { balance =
amount; }
}
class Main {
    public static void main(String[] args) {
        Account acc = new Account();
        acc.setBalance(2000);
        System.out.println(acc.getBalance());
    }
}
```
