UNIT 1: BASICS OF JAVA – QUICK REVISION NOTES

1. Features of Java

- **Simple** – Easy to learn, syntax similar to C/C++.
- **Object-Oriented** – Everything revolves around objects and classes.
- **Platform Independent** – "Write Once, Run Anywhere" due to bytecode.
- **Secure** – No pointers, has a strong memory management and security model.
- **Robust** – Exception handling and garbage collection ensure reliability.
- **Multithreaded** – Supports multiple threads of execution.
- **Portable** – Works across different operating systems.
- **High Performance** – Just-In-Time (JIT) compiler improves performance.
- **Dynamic** – Can load classes dynamically at runtime.

---

2. Byte Code and Java Virtual Machine (JVM)

- **Bytecode:** Intermediate code generated after compilation of .java file.
    - Platform-independent.
    - Stored in .class file.
- **JVM:** Java Virtual Machine executes the bytecode.
    - Converts bytecode into machine code.
    - Provides memory management and garbage collection.
    - Ensures platform independence.

3. Java Development Kit (JDK)

- Contains tools required to develop Java programs.
- Includes:
    - **JRE (Java Runtime Environment)**
    - **Compiler (javac)**
    - **Debugger, JavaDoc, and other utilities**
- Used for **compiling, debugging, and executing** Java applications.

---

4. Data Types

Primitive Data Types

Type	Size	Example
byte	1 byte	10
short	2 bytes	1000
int	4 bytes	10000
long	8 bytes	100000L
float	4 bytes	12.5f
double	8 bytes	23.45
char	2 bytes	'A'
boolean	1 bit	true/false

Non-Primitive Data Types

- **Strings, Arrays, Classes, Interfaces**

5. Operators

- **Arithmetic:** +, -, *, /, %
- **Relational:** ==, !=, >, <, >=, <=
- **Logical:** &&, ||, !
- **Assignment:** =, +=, -=, *=, /=
- **Increment/Decrement:** ++, --
- **Conditional (Ternary):** condition ? true : false
- **Bitwise:** &, |, ^, ~, <<, >>

---

6. Control Statements

a. If Statement

```
if(condition) {
  // statements
}
```

b. If-Else

```
if(condition) { ... } else { ... }
```

c. Nested If

```
if(a>0) {
  if(b>0) { ... }
}
```

d. If-Else Ladder

```
if(a==1) {...}
else if(a==2) {...}
else {...}
```

e. Switch Statement

```
switch(choice) {
  case 1: ...; break;
  case 2: ...; break;
```

default: ...;

}

f. Loops

- **While Loop:**
- while(condition) { ... }
- **Do-While Loop:**
- do { ... } while(condition);
- **For Loop:**
- for(int i=0; i<5; i++) { ... }
- **For-Each Loop:**
- for(int x : arr) { ... }

g. Jump Statements

- **break** → exits from loop/switch.
- **continue** → skips current iteration.

---

7. Arrays

- Used to store multiple values of same type.

Single Dimensional Array

int[] arr = {1,2,3,4};

Multidimensional Array

int[][] mat = {{1,2},{3,4}};

8. Strings

String Class

- Immutable (cannot be changed).

String s = "Java";

StringBuffer Class

- Mutable (can be changed).

StringBuffer sb = new StringBuffer("Hello");

sb.append(" Java");

Common String Operations

- length(), charAt(), substring(), concat(), equals(), compareTo(), toLowerCase(), toUpperCase(), trim()

---

9. Command Line Arguments

- Passed to main method during program execution.

```
class Test {
  public static void main(String[] args) {
    System.out.println(args[0]);
  }
}
```

Execution:
java Test Hello

---

10. Wrapper Classes

Here's the **expanded version** of the **Wrapper Class** section — still short enough for quick revision but with a clearer explanation and examples 👇

---

10. Wrapper Classes

🔹 **Introduction**

- In Java, **primitive data types** (like int, float, char, etc.) are **not objects** — they store simple values.
- Sometimes, we need to **treat these primitive values as objects** (e.g., when working with **Collections**, **Generics**, or **Object methods**).
- To achieve this, Java provides **Wrapper Classes** — one for each primitive type — that "wrap" the primitive value into an **object**.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

🔹 **Purpose / Use**

- To use primitives in **object-based frameworks** (e.g., ArrayList<Integer>).
- To **convert** data between primitives and Strings (parseInt(), toString()).
- To use **utility methods** like Integer.MAX_VALUE, Character.isDigit(), etc.

🔹 **Example**

```
public class WrapperDemo {
    public static void main(String[] args) {
        int a = 10;                 // primitive type
        Integer obj = Integer.valueOf(a); // Boxing (manual)
        int b = obj.intValue();          // Unboxing (manual)

        // Auto Boxing and Auto Unboxing (automatic)
        Integer x = a;   // auto boxing
        int y = x;       // auto unboxing

        System.out.println("Object: " + obj);
        System.out.println("Primitive: " + b);
    }
}
```

🔹 **Key Terms**

- **Boxing:** Converting primitive → object manually.
- **Unboxing:** Converting object → primitive manually.
- **Auto Boxing / Auto Unboxing:** Automatic conversion by Java compiler (since JDK 1.5).

Unit 2: Objects, Classes and Inheritance – OOP with Java

1. Class

- A **blueprint or template** for creating objects.
- Defines **data members (variables)** and **methods (functions)**.

```
class Student {
  int id;
  String name;
  void show() {
    System.out.println(id + " " + name);
  }
}
```

---

2. Object

- A **runtime instance** of a class.
- Represents real-world entities.

```
Student s1 = new Student();
```

---

3. Object Reference

- The variable that **holds the address** of an object in memory.

```
Student s1 = new Student();  // s1 is object reference
```

---

4. Constructor

- Special method used to **initialize objects**.
- Has the **same name as the class**, **no return type**.

```
class Demo {
  Demo() { System.out.println("Constructor called"); }
}
```

5. Constructor Overloading

- Multiple constructors in the same class with **different parameter lists**.

```
Demo() {}

Demo(int x) {}

Demo(String y) {}
```

---

6. Method Overloading

- Multiple methods with **same name but different parameters**.

```
void add(int a, int b){}

void add(double a, double b){}
```

---

7. Recursion

- A method calling **itself**.

```
int fact(int n){
  if(n==1) return 1;
  else return n*fact(n-1);
}
```

---

8. Passing and Returning Objects

- Methods can **take objects as parameters** and **return objects**.

```
Student display(Student s) { return s; }
```

---

9. new Operator

- Used to **create objects** and allocate memory.

```
Student s = new Student();
```

10. this Keyword

- Refers to the **current object**.
- Used to resolve name conflicts, call constructors, or pass current instance.

this.name = name;

---

11. static Keyword

- Belongs to the **class, not the object**.
- Can be used for **variables, methods, or blocks**.

static int count = 0;

static void show() { ... }

---

12. finalize() Method

- Called by **garbage collector** before an object is destroyed.

protected void finalize() { System.out.println("Destroyed"); }

---

13. Access Control Modifiers

Modifier	Scope
public	Everywhere
protected	Same package + subclass
default	Same package only
private	Within same class

14. Nested and Inner Classes

- **Nested class:** Class inside another class.
- **Inner class:** Non-static nested class; has access to outer class members.

```
class Outer {
  class Inner { void show(){} }
}
```

---

15. Anonymous Inner Class

- Class **without name**, used for **short implementations**.

```
Runnable r = new Runnable() {
  public void run() { System.out.println("Running"); }
};
```

---

16. Abstract Class

- Declared using abstract keyword.
- May have **abstract (unimplemented)** and **concrete** methods.

```
abstract class Shape {
  abstract void draw();
}
```

---

17. Inheritance

- Enables a class to **inherit** data and methods from another class.

```
class A {}
class B extends A {}
```

---

18. Inheriting Data Members and Methods

- Subclass can access **non-private** members of the parent class.

19. Constructor in Inheritance

- Parent class constructor runs **before** child class constructor.

---

20. Multilevel Inheritance

class A {}

class B extends A {}

class C extends B {}

- **Method Overriding:** Child class redefines parent's method.

@Override

void show() { ... }

---

21. super Keyword

- Refers to **parent class** members or constructors.

super(); // call parent constructor

super.display(); // call parent method

---

22. final Keyword

- **final variable:** constant
- **final method:** cannot be overridden
- **final class:** cannot be inherited

23. Interface

- Collection of **abstract methods** (and static/default methods in Java 8+).
- Implemented using implements keyword.

interface Animal { void eat(); }

class Dog implements Animal { public void eat(){} }

---

24. Interface Reference

- Interface variable can **refer to object** of implementing class.

Animal a = new Dog();

a.eat();

---

25. instanceof Operator

- Checks whether an object is an instance of a specific class or subclass.

if(obj instanceof Student) { ... }

---

26. Interface Inheritance

- An interface can **extend** another interface.

interface A {}

interface B extends A {}

---

27. Dynamic Method Dispatch

- Runtime polymorphism; **method call decided at runtime** based on object type.

A obj = new B(); // B overrides A's method

obj.show(); // B's version executed

28. Java Object Class

- Parent class of **all Java classes**.
- Common methods: toString(), equals(), hashCode(), clone(), getClass(), finalize().

---

29. Abstract Class vs Interface

Feature	Abstract Class	Interface
Inheritance	extends	implements
Methods	Can have abstract + concrete	Only abstract (till Java 7)
Variables	Can be non-final	Always final & static
Multiple Inheritance	Not supported	Supported
Constructor	Yes	No

---

30. Lambda Expressions (Java 8+)

- Short way to write **anonymous functions** (used with functional interfaces).

```
interface Sayable { void say(String msg); }

Sayable s = (msg) -> System.out.println(msg);

s.say("Hello Java");
```

🟢 1. Simple Java Program – Print "Hello Ishan"

```
// Simple Java Program to Print "Hello Ishan"
class HelloIshan {
    public static void main(String[] args) {
        // Print message on console
        System.out.println("Hello Ishan");
    }
}
```

🟢 2. Abstract Class Example

```
// Example of Abstract Class in Java

// Abstract class can have abstract (unimplemented) and concrete (implemented) methods
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method
    void sleep() {
        System.out.println("Animals need sleep.");
    }
}
// Subclass must provide implementation for abstract method
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
// Main class
class AbstractDemo {
    public static void main(String[] args) {
        // Create object of subclass
        Dog d = new Dog();
        d.sound();   // Calls implemented method
        d.sleep();   // Calls inherited concrete method
    }
}
```

🟢 3. Constructor Example

```
// Example to Demonstrate Constructor in Java
class Student {
    int id;
    String name;

    // Constructor - automatically called when object is created
    Student(int i, String n) {
        id = i;
        name = n;
    }

    // Method to display student details
    void display() {
        System.out.println("ID: " + id + ", Name: " + name);
    }
}

class ConstructorDemo {
    public static void main(String[] args) {
        // Creating objects and calling constructor
        Student s1 = new Student(101, "Ishan");
        Student s2 = new Student(102, "Raj");
        // Display information
        s1.display();
        s2.display();
    }
}
```

🟢 4. Interface Example

```
// Example of Interface in Java

// Interface defines method signatures only (no implementation)
interface Vehicle {
    void start();   // abstract method
}

// Class implementing the interface must define all methods
class Car implements Vehicle {
    public void start() {
        System.out.println("Car started with key ignition.");
    }
}

class InterfaceDemo {
    public static void main(String[] args) {
        // Create object using class
        Car c = new Car();
        c.start();

        // Interface reference variable
        Vehicle v = new Car();
        v.start();  // calls Car's implementation
    }
}
```