Here's a **detailed set of notes** covering the concepts you asked for, including explanations, diagrams, and examples.

---

1. **What is Java?**

Java is a **high-level, class-based, object-oriented programming language** that is designed to have as few implementation dependencies as possible.

- Developed by **James Gosling** at Sun Microsystems.
- It's **platform-independent** (Write Once, Run Anywhere – WORA).
- Used for desktop apps, mobile apps (Android), web apps, games, servers, etc.

---

2. **What is JVM (Java Virtual Machine)?**

- JVM is a **runtime environment** that executes Java bytecode.
- It is **platform-dependent** (different for Windows, Mac, Linux).
- Responsibilities:
    - Loads code
    - Verifies code
    - Executes code
    - Provides runtime environment (memory management, security)

---

3. **What is JRE (Java Runtime Environment)?**

- JRE = **JVM + Libraries + Other files** needed to run Java applications.
- It **does not** contain tools for development (like compiler).

Contains:

- JVM
- Java class libraries
- Deployment technologies (Java Web Start, Java Plug-in)

---

4. **What is WORA?**

> **WORA** = **Write Once, Run Anywhere**

Java code is compiled into **bytecode** that can run on any platform with a JVM. This gives Java its **platform independence**.

---

5. **What is OS (Operating System)?**

- OS is **system software** that manages computer hardware, software resources, and provides services for computer programs.
- Examples: Windows, Linux, macOS, Android

🔣 6. What is Bytecode?

- Bytecode is the **intermediate code** generated by the Java compiler (`.class` file).
- It is **not machine code**, but a **portable code** for JVM.

🧠 Example:

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

→ Compile: `javac Hello.java` → Generates `Hello.class` (bytecode) → Run using `java Hello`

⚙️ 7. How Java Executes Code – Life Cycle

✍️ Steps:

1. **Write Java Code** – `.java` file
2. **Compile** using `javac` – creates `.class` bytecode
3. **Class Loader** – loads `.class` files into JVM
4. **Bytecode Verifier** – checks code for validity
5. **JIT Compiler** – compiles bytecode to native machine code
6. **Execution Engine** – executes native code

🧩 Diagram:

```
Source Code (.java)
        ↓ javac
 Bytecode (.class)
        ↓ Class Loader
       [JVM]
        ↓ Bytecode Verifier
        ↓ JIT Compiler
        ↓ Execution Engine
    Machine Code → Output
```

🔡 8. What are Variables in Java?

A **variable** is a named container for storing data values.

🧠 Syntax:

```
int age = 25;
String name = "Sameer";
```

Types:

- Local Variable
- Instance Variable
- Static Variable

🔢 9. **What are Data Types in Java?**

Java is **strongly typed**, so every variable must have a type.

◇ Primitive Data Types:

Type	Size	Example
byte	1B	byte b = 1;
short	2B	short s = 2;
int	4B	int x = 10;
long	8B	long l = 100;
float	4B	float f = 2.3f;
double	8B	double d = 5.7;
char	2B	char c = 'A';
boolean	1bit	boolean b = true;

◇ Non-Primitive Types:

- Arrays
- Strings
- Classes
- Interfaces

✨ 10. **What are Literals?**

Literals are **fixed values** assigned to variables.

```
int x = 10;          // Integer literal
char y = 'A';        // Character literal
float pi = 3.14f;    // Floating-point literal
boolean flag = true; // Boolean literal
```

🔄 11. Type Conversion and Type Casting

☑ Type Conversion (Implicit/Widening)

Java automatically converts a smaller data type to a larger one.

```
int a = 10;
double d = a; // int → double
```

☑ No data loss, safe conversion.

⚠ Type Casting (Explicit/Narrowing)

You manually convert a larger type into a smaller type.

```
double d = 9.7;
int a = (int) d; // double → int → a = 9
```

⚠ Might lose data or precision.

🔁 Example with Both:

```
public class Conversion {
    public static void main(String[] args) {
        int x = 10;
        double y = x; // Implicit
        System.out.println(y); // 10.0

        double z = 5.9;
        int w = (int) z; // Explicit
        System.out.println(w); // 5
    }
}
```

Would you like a **PDF version** or these notes converted into **PowerPoint/Google Slides** format too?

Here are **comprehensive notes** on the remaining Java concepts you asked for:

🔁 1. **Conditional Statements in Java**

Conditional statements control the flow of a program **based on conditions** (true/false).

◇ Types:

- **if**
- **if-else**
- **if-else-if**
- **switch**

🧠 Examples:

```
int age = 18;
if (age >= 18) {
    System.out.println("Eligible to vote");
} else {
    System.out.println("Not eligible");
}
```

❓ 2. **Ternary Operator in Java**

The **ternary operator** is a **short form of if-else**.

◇ Syntax:

```
condition ? value_if_true : value_if_false;
```

🧠 Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
System.out.println(max); // 20
```

🔄 3. **Loops in Java**

Loops allow **repetitive execution** of a block of code.

◇ Types:

- **for loop**
- **while loop**
- **do-while loop**
- **for-each loop** (for arrays/collections)

🧠 Examples:

```
// for loop
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}

// while loop
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}

// do-while loop
int j = 1;
do {
    System.out.println(j);
    j++;
} while (j <= 5);
```

💡 4. **OOPs in Java (Object-Oriented Programming)**

Java is an **Object-Oriented Programming Language**.

◇ 4 Pillars of OOP:

Concept	Description
Encapsulation	Binding data and code into a single unit (class).
Inheritance	One class inherits properties of another.
Polymorphism	One task performed in many ways (method overloading/overriding).
Abstraction	Hiding internal details and showing only essential info.

🧱 5. **Class and Object**

◇ Class:

- A **blueprint** for objects.
- It defines properties (variables) and behaviors (methods).

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

◇ Object:

- An **instance** of a class.
- Created using the `new` keyword.

```
Car c1 = new Car();
c1.color = "Red";
c1.drive();
```

---

6. **JDK (Java Development Kit)**

- JDK is the **complete package** for Java development.
- Contains:
    - JRE (Java Runtime Environment)
    - Development tools like `javac`, `java`, debugger, etc.

JDK = JRE + Compiler + Debugger + Tools

---

7. **Method Overloading**

- Defining **multiple methods** in the same class with **same name but different parameters** (type or number).

Example:

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

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
```

```
    }
}
```

🧠 8. **Stack and Heap Memory**

◇ Stack Memory:

- Stores **method calls**, local variables.
- Follows **LIFO (Last In First Out)**.
- Faster and automatically managed.

◇ Heap Memory:

- Stores **objects and instance variables**.
- Slower, but large memory size.
- Requires garbage collection.

📊 Difference Table:

Feature	Stack	Heap
Stores	Method calls, local vars	Objects, class fields
Access Speed	Fast	Slower
Memory Size	Small	Large
Lifecycle	Short (method scope)	Long (global scope)
Managed by	JVM	Garbage Collector

☑ Summary

Concept	Explanation
Conditional	Executes blocks based on true/false conditions.
Ternary	Shortcut for if-else.
Loops	Repeats code (for, while, do-while).
OOPs	Class-based structure: Encapsulation, Inheritance, Polymorphism, Abstraction.
Class/Object	Class is a blueprint; object is an instance.
JDK	Complete toolkit for Java (includes compiler + runtime).
Method Overloading	Same method name, different parameters.
Stack vs Heap	Stack is for execution; heap is for objects.

Would you like all these notes in **PDF, PPT, or Word format** for printing or sharing?

Java Concepts Notes

1. Array An array is a fixed-size, ordered collection of elements of the same data type.

Declaration: type[] arrayName; Initialization: arrayName = new type[size]; Example:

int[] numbers = new int[5]; numbers[0] = 10; numbers[1] = 20;

2. Encapsulation Encapsulation is the bundling of data (fields) and methods that operate on that data, restricting direct access to some of an object's components using access modifiers.

Key Points: Use private fields and public getter/setter methods. Protects data integrity and hides implementation details.

Example:

class Human { private int age; private String name;

```
public int getAge() {
    return age;
}

public void setAge(int age) {
    if (age > 0) this.age = age; // Validation
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
```

}

public class Encapsulation { public static void main(String[] args) { Human h = new Human(); h.setAge(25); h.setName("John Doe"); System.out.println("Age: " + h.getAge() + ", Name: " + h.getName()); } }

3. Constructor A constructor is a special method used to initialize objects. It has the same name as the class and no return type.

Types: Default, parameterized. Example:

class Car { String model; int year;

```
// Parameterized constructor
Car(String model, int year) {
    this.model = model;
```

```
        this.year = year;
    }
```

}

public class ConstructorDemo { public static void main(String[] args) { Car car = new Car("Toyota", 2020); System.out.println("Model: " + car.model + ", Year: " + car.year); } }

4. Access Modifiers Access modifiers control the visibility of class members.

Types: public: Accessible everywhere. protected: Accessible within the same package and subclasses. default (package-private): Accessible within the same package. private: Accessible only within the same class.

Example:

class AccessDemo { public int publicVar = 1; protected int protectedVar = 2; int defaultVar = 3; private int privateVar = 4;

```
    public void show() {
        System.out.println("Private: " + privateVar); // Accessible
    }
```

}

5. Upcasting and Downcasting

Upcasting: Casting a subclass object to a superclass type (implicit). Downcasting: Casting a superclass object to a subclass type (explicit, requires type checking). Example:

class Animal { void sound() { System.out.println("Animal sound"); } }

class Dog extends Animal { void sound() { System.out.println("Bark"); } }

public class CastingDemo { public static void main(String[] args) { // Upcasting Animal animal = new Dog(); animal.sound(); // Outputs: Bark

```
        // Downcasting
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal;
            dog.sound(); // Outputs: Bark
        }
    }
```

}

6. Abstract Class An abstract class cannot be instantiated and may contain abstract methods (without implementation).

Key Points: Declared with abstract keyword. Subclasses must implement abstract methods.

Example:

abstract class Shape { abstract void draw(); }

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

public class AbstractDemo { public static void main(String[] args) { Shape shape = new Circle(); shape.draw(); } }

7. this, super, new

this: Refers to the current object. super: Refers to the superclass object or constructor. new: Creates a new object. Example:

class Parent { int x = 10; }

class Child extends Parent { int x = 20;

```
Child() {
    super(); // Call parent constructor
    System.out.println("Child x: " + this.x); // Current object
    System.out.println("Parent x: " + super.x); // Parent object
}
```

}

public class ThisSuperDemo { public static void main(String[] args) { Child child = new Child(); // new creates object } }

8. Autoboxing Autoboxing is the automatic conversion between primitive types and their wrapper classes.

Example:

public class AutoboxingDemo { public static void main(String[] args) { int i = 10; Integer intObj = i; // Autoboxing int j = intObj; // Unboxing System.out.println("Integer: " + intObj + ", int: " + j); } }

9. Polymorphism Polymorphism allows methods to be used in different ways based on the object type (method overriding or overloading).

Types: Compile-time (overloading). Runtime (overriding).

Example:

class Animal { void sound() { System.out.println("Some sound"); } }

class Cat extends Animal { void sound() { System.out.println("Meow"); } }

public class PolymorphismDemo { public static void main(String[] args) { Animal animal = new Cat(); // Runtime polymorphism animal.sound(); // Outputs: Meow } }

10. Overriding Overriding is when a subclass provides a specific implementation of a method defined in its superclass.

Example:

class Vehicle { void start() { System.out.println("Vehicle starting"); } }

class Bike extends Vehicle { void start() { System.out.println("Bike starting"); } }

public class OverridingDemo { public static void main(String[] args) { Bike bike = new Bike(); bike.start(); // Outputs: Bike starting } }

11. Overloading Overloading is defining multiple methods with the same name but different parameters.

Example:

class Calculator { int add(int a, int b) { return a + b; }

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

}

public class OverloadingDemo { public static void main(String[] args) { Calculator calc = new Calculator(); System.out.println(calc.add(5, 10)); // Outputs: 15 System.out.println(calc.add(5.5, 10.5)); // Outputs: 16.0 } }

12. String, StringBuffer, StringBuilder

String: Immutable, thread-safe. StringBuffer: Mutable, thread-safe (synchronized). StringBuilder: Mutable, not thread-safe (faster). Example:

public class StringDemo { public static void main(String[] args) { String str = "Hello"; StringBuffer sbf = new StringBuffer("Hello"); StringBuilder sbd = new StringBuilder("Hello");

```
    sbf.append(" World");
    sbd.append(" World");

    System.out.println("String: " + str); // Hello
    System.out.println("StringBuffer: " + sbf); // Hello World
    System.out.println("StringBuilder: " + sbd); // Hello World
}
```

}

13. Static Method Static methods belong to the class, not an instance, and are called using the class name.

Example:

class MathUtils { static int square(int x) { return x * x; } }

public class StaticDemo { public static void main(String[] args) { System.out.println("Square: " + MathUtils.square(5)); // Outputs: 25 } }

14. Final The final keyword prevents modification or extension.

Class: Cannot be subclassed. Method: Cannot be overridden. Variable: Cannot be reassigned. Example:

final class FinalClass { final int x = 10;

```
final void show() {
    System.out.println("Value: " + x);
}
```

}

public class FinalDemo { public static void main(String[] args) { FinalClass fc = new FinalClass(); fc.show(); } }

15. Wrapper Class Wrapper classes provide object representations of primitive types (e.g., Integer for int).

Example:

public class WrapperDemo { public static void main(String[] args) { Integer intObj = Integer.valueOf(100); System.out.println("Wrapper: " + intObj); } }

16. Package A package is a namespace for organizing classes and interfaces.

Example:

package com.example;

public class PackageDemo { public static void main(String[] args) { System.out.println("Inside com.example package"); } }

17. Import The import statement allows access to classes from other packages.

Example:

import java.util.ArrayList;

public class ImportDemo { public static void main(String[] args) { ArrayList list = new ArrayList<>(); list.add("Java"); System.out.println(list); } }

Here are the **Java notes** for the additional topics you mentioned — **Interface, Annotation, Inner Class, Lambda Function, and Enum** — complete with **theory** and **examples**:

Here are your **structured OOPS concept notes with previous code examples** for quick revision:

OOPS Concepts Notes

1. Encapsulation

☑ **Definition:** Wrapping data (variables) and methods into a single unit (class) and restricting direct access to some components.

☑ **Advantages:**

- Data hiding
- Control over data via getters/setters
- Increased security and maintainability

☑ **Example:**

```
class Student {
    private String name; // private data members
    private int age;

    // public getters and setters
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setAge(int age){
        this.age = age;
    }

    public int getAge(){
        return age;
    }
}

public class TestEncapsulation {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Sameer");
        s.setAge(21);
        System.out.println(s.getName() + " " + s.getAge());
    }
}
```

---

2. Inheritance

☑ **Definition:** Acquiring properties and behavior of parent class by child class using `extends` keyword.

☑ **Advantages:**

- Code reusability
- Method overriding support
- Hierarchical classification

☑ **Example:**

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

public class TestInheritance {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat(); // inherited method
        d.bark(); // own method
    }
}
```

---

3. Polymorphism

☑ **Definition:** One name, many forms. Two types:

(a) Compile-time Polymorphism (Method Overloading)

☑ **Definition:** Same method name with different parameters in the same class.

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

public class TestOverloading {
    public static void main(String[] args) {
        MathUtils m = new MathUtils();
        System.out.println(m.add(2, 3));    // calls int version
        System.out.println(m.add(2.5, 3.5)); // calls double version
```

```
    }
}
```

(b) Runtime Polymorphism (Method Overriding)

☑ **Definition:** Child class provides specific implementation of a method already defined in its parent class.

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Meow");
    }
}

public class TestOverriding {
    public static void main(String[] args) {
        Animal a = new Cat(); // upcasting
        a.sound(); // calls Cat's sound()
    }
}
```

---

4. Abstraction

☑ **Definition:** Hiding internal implementation details and showing only functionality to the user.

☑ **Achieved via:**

- Abstract classes
- Interfaces

☑ **Advantages:**

- Reduces complexity
- Focus on what an object does rather than how

☑ **Example using Abstract Class:**

```
abstract class Shape {
    abstract void draw(); // abstract method

    void color() {
        System.out.println("Shape has color");
```

```
    }
}

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

public class TestAbstraction {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw(); // implemented method
        s.color(); // inherited concrete method
    }
}
```

🎯 Final Revision Table

◇ Concept	🔑 Definition	💻 Example
Encapsulation	Data hiding & access via getters/setters	Private variables + public methods
Inheritance	Child acquires parent properties	`extends` keyword
Polymorphism	One name many forms	Overloading & Overriding
Abstraction	Hiding implementation details	Abstract classes / Interfaces

---

☑ Tips for Interviews:

✔ Always define each concept in **1 line** ✔ Write **simple examples** from above ✔ Mention **advantages** for each

---

If you want, I can prepare **Interface examples, Java 8 OOPS enhancements, or UML diagrams** for your upcoming OOPS revision tomorrow. Let me know your plan.

☑ 17. **Interface**

Theory: An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. It is used to achieve **abstraction** and **multiple inheritance** in Java.

Types of Interface:

- **Normal Interface**
- **Functional Interface** (Single Abstract Method)
- **Marker Interface** (No methods)

Example:

```
interface Animal {
    void sound();  // abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Bark");
    }
}
```

☑ 18. **Annotation**

Theory: Annotations provide **metadata** about the program but are **not part of the program itself**. They have no direct effect on the operation of the code they annotate.

Common Annotations:

- `@Override`
- `@Deprecated`
- `@SuppressWarnings`
- Custom Annotations (`@interface`)

Example:

```
class Test {
    @Override
    public String toString() {
        return "Test class";
    }
}
```

Custom Annotation Example:

```
@interface Info {
    String author();
}

@Info(author = "Sameer")
class MyClass {}
```

☑ 19. **Inner Class**

Theory: A class declared inside another class. Useful for logically grouping classes that will be used only in one place.

Types of Inner Classes:

- **Member Inner Class**
- **Static Nested Class**
- **Local Inner Class**
- **Anonymous Inner Class**

Example (Member Inner Class):

```
class Outer {
    class Inner {
        void display() {
            System.out.println("Inner class method");
        }
    }
}
```

Usage:

```
Outer.Inner obj = new Outer().new Inner();
obj.display();
```

☑ 20. **Lambda Function (Lambda Expression)**

Theory: Lambda expressions are used to provide **implementation of functional interfaces** (interfaces with one abstract method). Introduced in **Java 8**.

Syntax:

```
(parameters) -> expression
```

Example:

```
interface Drawable {
    void draw();
}

public class LambdaDemo {
    public static void main(String[] args) {
        Drawable d = () -> System.out.println("Drawing");
        d.draw();
    }
}
```

With Parameters:

```
interface Addable {
    int add(int a, int b);
}

Addable sum = (a, b) -> a + b;
System.out.println(sum.add(10, 20));
```

☑ 21. **Enum (Enumeration)**

Theory: Enums define a fixed set of constants. They are a special Java type used to define collections of constants.

Example:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY;
}

public class EnumTest {
    public static void main(String[] args) {
        Day d = Day.MONDAY;
        System.out.println(d);
    }
}
```

Enum with fields and methods:

```
enum Status {
    SUCCESS(200), ERROR(404);

    private int code;
    Status(int code) {
        this.code = code;
    }
    public int getCode() {
        return code;
    }
}
```

Would you like me to compile **all 21 topics** into a single **PDF or formatted printable file** for easier revision?

Here's a detailed breakdown of your Java code with explanation on **Exception Handling**, its **types**, and improvements to help you learn better.

☑ 1. What is Exception Handling in Java?

Exception Handling is a mechanism to handle runtime errors so that the normal flow of the application can be maintained.

☑ 2. Types of Exceptions in Java

Java categorizes exceptions into two major types:

A. **Checked Exceptions**

- Occur at compile-time.
- Must be either caught or declared using `throws`.

Examples:

- `IOException`
- `ClassNotFoundException`

B. **Unchecked Exceptions**

- Occur at runtime.
- Not checked at compile-time.

Examples:

- `ArithmeticException`
- `NullPointerException`
- `ArrayIndexOutOfBoundsException`

☑ 3. Exception Handling Keywords in Java

- `try`: Contains code that might throw an exception.
- `catch`: Catches and handles the exception.
- `throw`: Throws an exception.
- `throws`: Declares exceptions.
- `finally`: Block that executes regardless of an exception.

☑ Code Explanation:

☑ Custom Exception

```
class SamException extends Exception {
    public SamException(String string) {
        super(string);
    }
}
```

You're creating a **custom checked exception** named `SamException`.

☑ Handling Checked Exception (ClassNotFoundException)

```
class A {
    public void show() throws ClassNotFoundException {
        Class.forName("DemodO");
    }
}
```

- This throws a **checked exception** since the class name may not exist.

```
try {
    obj.show();
} catch (Exception e) {
    System.out.println("something went wrong");
    e.printStackTrace();
}
```

- This catches the `ClassNotFoundException`.

☑ Using Custom Exception Handling (Commented Block)

```
// if(j == 0)
//     throw new SamException("lkfvmkll");
```

You were trying to throw your custom exception when `j == 0`. Uncommenting and handling this would demonstrate custom exception usage.

☑ Input Handling and Exception Safe Code

```
try (BufferedReader bf = new BufferedReader(new InputStreamReader(System.in))) {
    // int num = Integer.parseInt(bf.readLine());
    // System.out.println(num + 2);
}
```

This uses **try-with-resources**, which ensures `BufferedReader` is closed automatically — a modern and safe practice.

☑ Scanner Input

```
Scanner sc = new Scanner(System.in);
int num = sc.nextInt();
int num3 = sc.nextInt();
int result = num + num3;
System.out.println(result);
```

No exception handling here — ideally, wrap this in a `try-catch` to handle `InputMismatchException`.

🔁 Suggested Final Version of Input Handling:

```
try (Scanner sc = new Scanner(System.in)) {
    System.out.println("Enter two numbers:");
    int num = sc.nextInt();
    int num2 = sc.nextInt();
    int result = num + num2;
    System.out.println("Sum is: " + result);
} catch (Exception e) {
    System.out.println("Invalid input: " + e.getMessage());
}
```

📝 Summary Table:

Type	Example Class	Checked/Unchecked	Handling Required?
Arithmetic	`ArithmeticException`	Unchecked	Optional
Array Out of Bound	`ArrayIndexOutOfBoundsException`	Unchecked	Optional
File Not Found	`FileNotFoundException`	Checked	Yes
Class Not Found	`ClassNotFoundException`	Checked	Yes
Custom Exception	`SamException`	Depends (yours is Checked)	Yes

Let me know if you want:

- A simple diagram explaining exception flow
- A mini project example using custom and built-in exceptions
- Notes in PDF format for revision Here's a clear and concise explanation of each concept you've listed — **Thread**, **Record Classes**, **Stream API**, **Sorting**, **Map/Filter/Reduce**, **Sealed Classes**, and **Compact Canonical Constructor** — all highly relevant for Java interviews and real-world Java development.

☑ 1. **Thread in Java**

What is a Thread?

A **thread** is a lightweight subprocess, the smallest unit of execution in a Java program.

Types of Threads:

- **User Thread**: Created by the application (e.g., `Thread t = new Thread()`).
- **Daemon Thread**: Background thread (e.g., garbage collector). Ends when all user threads finish.

Ways to Create a Thread:

1. **Extend `Thread` class**

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}
```

2. **Implement `Runnable` interface**

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread...");
    }
}
```

3. **Using lambda (Java 8+)**

```
new Thread(() -> System.out.println("Lambda thread")).start();
```

☑ 2. **Record Classes in Java (Java 14+)**

What is a Record?

A **record** is a special class used to create immutable data carriers.

Example:

```
record Student(String name, int age) {}
```

Features:

- Auto-generates `constructor`, `toString()`, `equals()`, `hashCode()`.
- Cannot extend other classes.
- Good for DTOs.

☑ 3. **Stream API**

Introduced in **Java 8**, used to process collections of data in a **functional style**.

Common Operations:

- `filter()` → filters data
- `map()` → transforms data
- `reduce()` → combines data into a single result

☑ 4. **Sorting with Stream API**

Example: Sort a list of integers

```
List<Integer> nums = Arrays.asList(4, 1, 3, 2);
nums.stream()
    .sorted()
    .forEach(System.out::println);
```

Sorting custom objects:

```
students.stream()
    .sorted(Comparator.comparing(Student::getName))
    .forEach(System.out::println);
```

☑ 5. **Map, Filter, Reduce**

`map()`:

Transforms each element.

```
List<Integer> nums = List.of(1, 2, 3);
nums.stream().map(x -> x * 2).forEach(System.out::println);
```

`filter()`:

Filters elements based on a condition.

```
nums.stream().filter(x -> x % 2 == 0).forEach(System.out::println);
```

`reduce()`:

Combines elements into one.

```
int sum = nums.stream().reduce(0, (a, b) -> a + b);
```

☑ 6. **Sealed Classes (Java 15+)**

Purpose:

To **restrict** which other classes or interfaces may extend or implement them.

Example:

```
public sealed class Shape permits Circle, Rectangle {}

final class Circle extends Shape {}
final class Rectangle extends Shape {}
```

- Helps maintain **control** over class hierarchy.
- Enables better **security**, **modeling**, and **IDE support**.

☑ 7. **Compact Canonical Constructor**

When you define a **record**, Java auto-generates a constructor. You can also define your own **compact version** to add logic.

Example:

```
record Student(String name, int age) {
    public Student {
        if (age < 0)
            throw new IllegalArgumentException("Age must be positive");
    }
}
```

- You don't need to write the entire constructor — just add logic.

🧠 Want a Summary in Table Format?

Concept	Description
Thread	Unit of execution, supports multitasking
User vs Daemon Thread	User: primary tasks, Daemon: background tasks
Record Class	Immutable data class with auto-generated methods
Stream API	Functional processing of collections
Sorting	Sort elements using `sorted()` with or without `Comparator`
Map	Transform each element (e.g., multiply, uppercase)
Filter	Retain only matching elements (e.g., even numbers)
Reduce	Aggregate data to single value (e.g., sum, product)
Sealed Class	Restrict class hierarchy to specific subclasses
Compact Canonical Constructor	Constructor logic without redeclaring fields in records

---

Let me know if you want:

- Example problems with outputs
- One-pager printable notes/PDF
- Quiz or MCQs for revision

Here's a complete and beginner-friendly explanation of **Java Collections** focusing on **List, Collection, Map, and Set**, along with examples and differences.

☑ 1. What is the Java Collection Framework?

The **Java Collection Framework** is a set of classes and interfaces used to **store**, **manipulate**, and **retrieve** groups of objects.

It includes:

- **Interfaces**: `Collection`, `List`, `Set`, `Map`, `Queue`, etc.
- **Implementations**: `ArrayList`, `HashSet`, `HashMap`, `LinkedList`, etc.

☑ 2. Collection Interface (java.util.Collection)

The **root interface** for most collection types (excluding `Map`).

```
Collection<String> items = new ArrayList<>();
items.add("Apple");
items.add("Banana");
```

- Basic operations: `add()`, `remove()`, `clear()`, `contains()`, `size()`
- It is the **parent** of `List`, `Set`, and `Queue`

☑ 3. List Interface

- **Ordered**
- **Allows duplicates**
- Access elements via **index**

☑ Common Implementations:

- `ArrayList` (resizable array)
- `LinkedList` (doubly-linked list)
- `Vector` (synchronized)

☑ Example:

```
List<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("Java"); // allows duplicate
System.out.println(list.get(1)); // Output: Python
```

☑ 4. Set Interface

- **No duplicates**
- **Unordered** (unless using `LinkedHashSet` or `TreeSet`)

☑ Common Implementations:

- `HashSet` – Unordered, no duplicates
- `LinkedHashSet` – Maintains insertion order
- `TreeSet` – Sorted order

☑ Example:

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple"); // ignored
System.out.println(set); // Output: [Banana, Apple] (order may vary)
```

☑ 5. Map Interface

- **Key-value pairs**
- Keys are **unique**, values can **repeat**
- Not a part of the `Collection` hierarchy

☑ Common Implementations:

- `HashMap` – Unordered
- `LinkedHashMap` – Maintains insertion order
- `TreeMap` – Sorted by keys

☑ Example:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
map.put(1, "Uno"); // replaces value at key 1
System.out.println(map.get(1)); // Output: Uno
```

☑ Summary Table:

Interface	Allows Duplicates	Ordered	Sorted	Key-Value	Example Classes
Collection	Yes (in List)	Varies	No	No	ArrayList, HashSet
List	Yes	Yes	No	No	ArrayList, LinkedList
Set	No	No	Optional	No	HashSet, TreeSet
Map	Keys: No, Values: Yes	Yes (LinkedHashMap)	Yes (TreeMap)	Yes	HashMap, TreeMap

---

🧠 Bonus: When to Use What?

- Use **List** when you need **order + duplicates**.
- Use **Set** when you want **unique elements**.
- Use **Map** when you need to **associate keys with values**.

Would you like:

- Real-world examples for each?
- Practice programs with output?
- Interview MCQs on Collection Framework?

Let me know!