Module: 6 Core Java

1.History of Java

1991: Created by James Gosling and team at Sun Microsystems as **"**Oak**"**, for embedded devices.

1995: Renamed to Java and officially released. Known for **"**Write Once, Run Anywhere**".**

1998–2006: Major improvements like Swing, Collections, and Generics introduced (Java 2 to Java 5).

2010**:** Oracle acquired Sun Microsystems and took over Java development.

2014**:** Java 8 introduced major features like Lambda expressions and Stream API**.**

2017+**:** New version every 6 months (Java 9 to Java 21), with features like modules**,** var**,** records, and pattern matching**.**

Today: Java is widely used in web**,** enterprise**,** Android, and cloud applications**.**

2. Features of Java (Platform Independent, Object-Oriented, etc.)

Features of Java:

- Platform Independent: Code runs on any OS via JVM (Write Once, Run Anywhere).
- Object-Oriented: Everything is treated as an object; supports OOP concepts like inheritance, polymorphism, etc.
- Simple: Easy to learn with a clean syntax and no complex features like pointers.
- Secure: Offers runtime checking, bytecode verification, and no direct memory access.
- Robust: Strong memory management, exception handling, and type checking.
- Multithreaded: Supports concurrent execution using multiple threads.
- Portable:Java code is platform-independent and architecture-neutral.
- High Performance: Just-In-Time (JIT) compiler improves performance.
- Distributed: Supports RMI and EJB for building distributed applications.
- Dynamic: Can load classes at runtime and supports dynamic linking.

3. Understanding JVM, JRE, and JDK

JVM (Java Virtual Machine)

- Runs Java bytecode (.class files).
- Provides platform independence.
- Part of JRE.

JRE (Java Runtime Environment)

- Contains JVM + libraries to run Java programs.
- Used to run, not develop Java apps.

JDK (Java Development Kit)

- Contains JRE + development tools (compiler, debugger, etc.).
- Used to write, compile, and run Java programs.

4. Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)

1.Install JDK

Download from Oracle → Install → Set JAVA_HOME → Check with java -version.

2.Install IDE

Eclipse or IntelliJ IDEA (Community)

Create New Java Project → Add Class → Start coding.

3.Run Java Program

Write code → Click Run button in IDE.

5.Java Program Structure (Packages, Classes, Methods)

Package:

Groups related classes.

Example: package myapp;

Class:

Main building block of a program.

Example:

public class Main {

}

Method:

Performs actions.

Entry point:

public static void main (String[] args) {

// code here

}

6. Data Types, Variables, and Operators

1. Data Types:

- Primitive types: byte, short, int, long (integers); float, double (floating-point numbers); char (characters); Boolean (true/false).
- Non-primitive types: Classes, Arrays, Strings, etc.

2. Variables:

- Containers to store data values.
- Must be declared with a data type (e.g., int age;).
- Can be initialized when declared or later assigned a value.

3. Operators:

- Arithmetic operators: **+, -, *, /, %**
- Relational operators: **==, !=, >, <, >=, <=**
- Logical operators: && (AND), || (OR), ! (NOT)
- Assignment operators: **=, +=, -=, *=, /=, %=**
- Unary operators: **++ (**increment), -- (decrement), +, **-**

7. Variable Declaration and Initialization

- Declaration**:** Reserve space for a variable with type and name
  Example:int age;
- Initialization**:** Assign a value to the variable.
  Example: age = 25;
- Declaration + Initialization together:int age = 25;

8. Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise

Arithmetic Operators

+, -, *, /, %
Example: int sum = a + b;

Relational Operators

==, !=, >, <, >=, <=
Example: if(a > b) { }

Logical Operators
&& (AND), || (OR), ! (NOT)
Example: if(a > 0 && b < 10) { }

Assignment Operators
=, +=, -=, *=, /=, %=
Example: a += 5; (means a = a + 5;)

Unary Operators
++ (increment), -- (decrement), +, -, !
Example: i++;, !flag

Bitwise Operators
& (AND), | (OR), ^ (XOR), ~ (NOT), <<, >>
Example: int x = a & b;

9.Type Conversion and Type Casting

1.Type Conversion (Implicit): Automatic conversion by Java when assigning smaller type to larger type.
*Example:*int i = 100;

long l = i; // int to long (widening)

2.Type Casting (Explicit): Manually converting a larger type to a smaller type.
*Example:*long l = 100L;

int i = (int) l; // long to int (narrowing)

10.If-Else Statements

Used to execute code based on a condition.

Syntax:

if (condition)

{

// code if condition is true

}

Else

{

// code if condition is false

}

You can also use **else if** for multiple conditions:

```
if (condition1)
{
// code
}
else if (condition2)
{
// code
}
else {
// code
}
```

11. Switch Case Statements

Switch Statement Used to select one of many code blocks based on a variable's value.

Syntax:

```
switch (variable)
{
case value1:
// code block
break;
case value2:
// code block
break;
default:
// code if no case matches
}
```

break prevents fall-through to next case

12.Loops (For, While, Do-While)

1.For Loop: Repeats code a fixed number of times.

```
for (int i = 0; i < 5; i++)
{
// code
}
```

2.While Loop: Repeats code while condition is true.

```
while (condition)
{
// code
}
```

3.Do-While Loop: Executes code once, then repeats while condition is true.

```
Do
{
// code
} while (condition);
```

13.Break and Continue Keywords

Break: Exits the nearest enclosing loop or switch immediately.
Example:

```
for (int i = 0; i < 5; i++)
{
if (i == 3) break; // stops loop at i=3
}
```

Continue: Skips the current loop iteration and moves to the next one.
Example:

```
for (int i = 0; i < 5; i++)
{
if (i == 3) continue; // skips i=3, continues loop
}
```

14.Defining a Class and Object in Java

Class: A blueprint or template for creating objects.
Defines properties (variables) and behaviours (methods).

```
public class Car

{

String color;

void drive ()

{

System.out.println("Car is driving");

}

}
```

Object: An instance of a class.
Created using the new keyword.

```
Car myCar = new Car();

myCar.drive()
```

15.Constructors and Overloading

Constructors: Special method to initialize objects. Same name as the class, no return type.

Called automatically when creating an object.

```
public class Car {

String color;

Car () {  // Constructor

color = "Red";

}

}
```

Constructor Overloading

Multiple constructors with different parameters in the same class.

Allows different ways to create objects.

```
Car () {
```

```
color = "Red";
}
Car (String c) {
color = c;
}
```

16.Object Creation, Accessing Members of the Class

Object Creation: Use new keyword to create an object (instance of a class).

Class Name obj = new ClassName ();

Accessing Members

Use dot . operator to access variables and methods of the object.

obj. variable = value;

obj. method ();

17.this Keyword

this Keyword:

Refers to the **current object** inside a class.

Used to distinguish between instance variables and parameters with the same name.

Can call another constructor in the same class using this ().

Example:

```
public class Car
{
String color;
Car (String color)
{
this. Color = color; // 'this. Color' refers to instance variable
}
}
```

18.Defining Methods

A method is a block of code that performs a task.

Syntax:

return Type method Name(parameters) {

// method body

}

Example:

public void greet ()

{

System.out.println("Hello!");

}

Method Parameters and Return Types

Parameters:
Inputs given to a method inside parentheses.
Example:

void greet (String name)

{

System.out.println("Hello, " + name);

}

Return Types:
Type of value a method returns. Use void if no return.
Example:

int add (int a, int b)

{

return a + b;

}

19.Method Overloading

Defining multiple methods with the same name but different parameters (type, number, or order).

Allows different ways to call the same method name.

Return type can be the same or different.

Example:

void show(int a) { }

void show(String s) { }

void show (int a, int b) { }

20.Static Methods and Variables

Static Variable: Belongs to the class, shared by all objects.

Ex: static int count;

Static Method**:** Can be called without creating an object.
Cannot access instance variables directly.

Ex:static void display ()

{

System.out.println("Static method");

}

21.Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction

1.Encapsulation

- Wrapping data (variables) and methods into a single unit (class).
- Use private variables + public getters/setters.

2.Inheritance

- Child class inherits properties and methods from parent class.
- Uses extends keyword.

3.Polymorphism

- Ability to take many forms.
- Method overloading (compile-time) and overriding (run-time).

4Abstraction

- Hiding internal details and showing only essentials.
- Achieved via abstract classes and interfaces.

22.Inheritance: Single, Multilevel, Hierarchical

1.Single Inheritance

- One child class inherits from one parent class.
- class Parent { }
- class Child extends Parent { }

2.Multilevel Inheritance

- A class inherits from a child class, forming a chain.
- class Grandparent { }
- class Parent extends Grandparent { }
- class Child extends Parent { }

3.Hierarchical Inheritance

- Multiple child classes inherit from one parent class.
- class Parent { }
- class Child1 extends Parent { }
- class Child2 extends Parent { }

23.Method Overriding and Dynamic Method Dispatch

Method Overriding:

Child class redefines a method of its parent class with the same name and parameters.

Enables run-time polymorphism.

Use @Override annotation (optional but recommended).

```
class Parent
{
void show ()
{
System.out.println("Parent");
}
}
class Child extends Parent
{
@Override
```

```
void show ()
{
System.out.println("Child");
}
}
```

Dynamic Method Dispatch:

Java decides at runtime which overridden method to call, based on the object type (not reference type).

```
Parent obj = new Child ();
obj.show();  // Calls Child's show()
```

24.Constructor Types (Default, Parameterized)

Default Constructor:

No parameters, automatically provided if no constructor is defined.

Initializes objects with default values.

```
class Car
{
Car() {
// default constructor
}
}
```

Parameterized Constructor:

Takes parameters to initialize objects with specific values.

```
class Car {
String color;
Car(String c) {
color = c;
}
}
```

25.Copy Constructor (Emulated in Java)

Copy Constructor:

Java doesn't have built-in copy constructors like C++, but you can create one manually**.**

It takes an object of the same class as a parameter and copies its fields.

```
class Car {
String color;
Car (Car c) {          // Copy constructor
this. color = c.color;
}
}
```

Used to create a new object with the same state as an existing object.

26.Constructor Overloading

Constructor Overloading :

Multiple constructors in the same class with different parameter lists.

Allows objects to be created in different ways.

Example:

```
class Car {
Car() {
System.out.println("Default Constructor");
}
Car (String color) {
System.out.println("Color: " + color);
}
}
```

27.Object Life Cycle and Garbage Collection

object Life Cycle :

Creation – Using new keyword.

Usage – Accessing fields and methods.

No Reference – Object becomes unreachable.

Garbage Collected – JVM reclaims memory.

Garbage Collection:

Automatic memory management by JVM.

Unused objects are removed to free memory.

Can suggest GC using:

System.gc();

28.One-Dimensional and Multidimensional Arrays

One-Dimensional Array:

A linear list of elements.

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

Multidimensional Array:

An array of arrays (e.g., 2D matrix).

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

29.String Handling in Java: String Class, String Buffer, StringBuilder

String Class:

Immutable (can’t be changed after creation).

Used for constant/fixed string operations.

```
String s = "Hello";
s = s + " World";  // creates new object
```

String Buffer:

Mutable and **thread-safe** (synchronized).

Good for multi-threaded environments.

```
StringBuffer sb = new StringBuffer("Hello");
```

sb.append(" World");

StringBuilder:

Mutable but **not thread-safe** (faster than StringBuffer).

Best for single-threaded use.

StringBuilder sb = new StringBuilder("Hello");

sb. append(" World");

30.Array of Objects

Array of Objects :

An array where each element is an object of a class.

Useful for storing multiple objects of the same type.

Example:

class Car {

String color;

Car(String c) { color = c; }

}

Car[] cars = new Car[2];

cars[0] = new Car("Red");

cars[1] = new Car("Blue");

31.String Methods (length, charAt, substring, etc.)

length () – Returns the length of the string.

str.length();

charAt (int index) – Returns character at specified index.

str.charAt(0);

substring(int begin) – Returns substring from index to end.

str.substring(2);

substring(int begin, int end) – Returns substring between indices.

str.substring(1, 4);

toLowerCase() / toUpperCase() – Converts case.

str.toLowerCase();

equals(String s) – Compares strings (case-sensitive).

str.equals("hello");

contains(String s) – Checks if substring exists.

str.contains("world");

replace(old, new) – Replaces characters or substrings.

str.replace("a", "b");

32. Inheritance Types and Benefits

Single Inheritance – One child, one parent class.

Multilevel Inheritance – Chain of inheritance.

Hierarchical Inheritance – Multiple children, one parent.

🔷 **Benefits of Inheritance**

Code Reusability – Reuse parent class code.

Method Overriding – Customize behavior.

Improved Maintainability – Easier to update shared logic.

Supports Polymorphism – Enables dynamic method dispatch.

33. Method Overriding

Occurs when a subclass defines a method with the same name, return type, and parameters as in the parent class.

Used for runtime polymorphism.

Example:

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

class Dog extends Animal {
```

```
@Override
void sound() {
System.out.println("Bark");
}
}
```

✅ **Rules:**

- Must have same method signature.
- Only possible in inheritance.
- Use @Override for clarity.
-

34. Dynamic Binding (Run-Time Polymorphism)

- Dynamic Binding (Run-Time Polymorphism)
- Method call is resolved at runtime, not compile-time.
- Achieved through method overriding and parent class reference pointing to child class object.

Example:

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

Animal a = new Dog ();  // Upcasting
a. sound ();  // Output: Dog barks (resolved at runtime)
```

35. Super Keyword and Method Hiding

Refers to the parent class.

Used to:

Access parent class variables/methods.

Call parent class constructor.

```
class Parent {
void show()
{
 System.out.println("Parent");
}
}

class Child extends Parent
 {
void show()
 {
super.show();  // calls Parent's show()
System.out.println("Child");
}
}
```

Method Hiding:

Happens when a **static method** in a subclass has the **same name** as in the parent class.

No overriding; method call depends on **reference type**, not object.

```
class Parent {
static void display()
{
 System.out.println("Parent");
 }
}
```

```
class Child extends Parent
 {
static void display()
 {
 System.out.println("Child");
 }
}


Parent p = new Child();
p.display();  // Output: Parent (method hiding)
```

36. Abstract Classes and Methods

Abstract Class:
A class declared with the abstract keyword. It cannot be instantiated and may contain abstract methods (without body) and concrete methods (with body).
Example:

```
abstract class Animal {
abstract void sound ();  // abstract method
void eat()
 {
System.out.println("This animal eats food.");
}
}
```

Abstract Method:
A method without a body, declared using abstract keyword. Subclasses must override it.
Example: abstract void sound (); // No body

Use:
Abstract classes provide a base for subclassing with some shared code and some methods that must be implemented by subclasses.

37. Interfaces: Multiple Inheritance in Java

Interface:
An interface is a blueprint of a class with only **abstract methods** (by default) and **static/final variables**.
Syntax:

```
interface A
 {
void show();
}
```

Multiple Interfaces:
A class in Java can implement multiple interfaces (Java supports multiple inheritance through interfaces).
Example:

```
interface A
{
void show();
}
interface B
 {
void display();
}
class MyClass implements A, B {
public void show() {
System.out.println("Show method");
}

public void display() {
System.out.println("Display method");
}
}
```

Use:
Multiple interfaces allow a class to inherit behavior from multiple sources, avoiding diamond problem.

38. Implementing Multiple Interfaces

A class can implement multiple interfaces using a comma-separated list.

Example:

```
interface A
{
void methodA();
}

interface B
{
void methodB();
}

class MyClass implements A, B
{
public void methodA()
{
System.out.println("Method A");
}

public void methodB()
{
System.out.println("Method B");
}
}
```

Key Point:
All methods from the interfaces must be implemented in the class.

Benefit:
Achieves multiple inheritance without ambiguity.

39. Java Packages: Built-in and User-Defined Packages

Package:
A package is a group of related classes and interfaces.

1. Built-in Packages:

Provided by Java API.

Examples:

java.util (e.g., ArrayList, HashMap)

java.io (e.g., File, InputStream)

java.lang (default, e.g., String, Math)

2. User-Defined Packages:

Created by programmers to organize their code.

Example:

```
// File: MyPackage/MyClass.java
package MyPackage;

public class MyClass {
public void display() {
System.out.println("Hello from MyPackage!");
}
}
```

Using it:

```
import MyPackage.MyClass;

public class Test {
public static void main(String[] args) {
```

```
MyClass obj = new MyClass();
obj.display();
}
}
```

40. Access Modifiers: Private, Default, Protected, Public

Modifier	Access Level
Private	Accessible **only within the same class**.
default *(no modifier)*	Accessible within the **same package**.
Protected	Accessible in the **same package** and **subclasses (even in other packages)**.
Public	Accessible **from anywhere**.

Example:

```
public class Example
{
private int a;       // only in this class
int b;               // default - same package
protected int c;     // same package + subclasses
public int d;        // everywhere
}
```

41. Importing Packages and Classpath

Importing Packages:

Use import to access classes from a package.

Examples:

```
import java.util.Scanner;       // Import specific class
import java.util.*;             // Import all classes from package
```

Classpath:

The **classpath** tells Java **where to find classes and packages**.

Set using:

Command line: java -cp path MyClass

Environment variable: CLASSPATH

Example:

javac -cp . MyProgram.java

java -cp . MyProgram

42. Types of Exceptions: Checked and Unchecked

1. Checked Exceptions

- **Checked at compile-time.**
- Must be either caught or declared in the method using throws.
- Examples:
  - IOException
  - SQLException
  - FileNotFoundException

```
try
{
FileReader file = new FileReader("file.txt");
}
catch (IOException e)
{
e.printStackTrace();
}
```

2. Unchecked Exceptions

- **Checked at runtime.**
- Not required to be caught or declared.
- Examples:
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ArithmeticException

```
int a = 10 / 0; // ArithmeticException
```

43. try, catch, finally, throw, throws

1. try

- Used to wrap code that might throw an exception.

```
try {
int a = 10 / 0;
}
```

2. catch

- Catches and handles the exception thrown in try.

```
catch (ArithmeticException e)
{
System.out.println("Cannot divide by zero");
}
```

3. finally

- Always executes after try and catch blocks, used for cleanup.

```
finally
{
System.out.println("This will always run");
}
```

4. throw

- Used to explicitly throw an exception.

```
throw new ArithmeticException("Divide by zero");
```

5. throws

- Declares exceptions that a method might throw.

```
public void readFile() throws IOException
{
// code that may throw IOException
}
```

44. Custom Exception Classes

Custom Exception:

You can create your own exception by **extending** the Exception (for checked) or RuntimeException (for unchecked) class.

✅ **Steps to create a custom exception:**

```
// Custom checked exception
class MyException extends Exception {
public MyException(String message) {
super(message);
}
}
// Usage
public class Test {
public static void main(String[] args) {
try {
throw new MyException("Custom error occurred");
} catch (MyException e) {
System.out.println(e.getMessage());
}
}
}
```

🔷 **For Unchecked Exception:**

Extend RuntimeException instead of Exception.

```
class MyUncheckedException extends RuntimeException
 {
public MyUncheckedException(String msg)
 {
super(msg);
}
}
```

45. Introduction to Threads

Threads: A **thread** is a lightweight process.

- Used to **run multiple tasks simultaneously** (multithreading).
- Improves performance, especially in I/O or background tasks.

Two ways to create a thread:

1. **Extend Thread class**

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

2. **Implement Runnable interface**

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

🔁 **Thread Lifecycle:**

New → Runnable → Running → Waiting → Terminated

46. Creating Threads by Extending Thread Class or Implementing Runnable Interface

1. Extending Thread class

- Create a class that extends Thread
- Override the run() method
- Call start() to begin execution

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

public class Test {
public static void main(String[] args) {
MyThread t1 = new MyThread();
t1.start();  // starts the thread
}
}
```

✅ **2. Implementing Runnable interface**

- Create a class that implements Runnable

- Override run() method
- Pass object to a Thread and call start()

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

public class Test {
public static void main(String[] args) {
Thread t = new Thread(new MyRunnable());
t.start();  // starts the thread
}
}
```

47. Thread Life Cycle

✅ **1. New**

- Thread is created but not started.

```
Thread t = new Thread();
```

✅ **2. Runnable**

- start() is called; thread is ready to run.

✅ **3. Running**

- Thread is actively executing run () method.

✅ **4. Blocked / Waiting / Timed Waiting**

- Thread is paused or waiting for a resource/time.

✅ **5. Terminated (Dead)**

- Thread has finished execution or stopped.

48. Synchronization and Inter-thread Communication

Synchronization & Inter-thread Communication

- **Synchronization:**
  Ensures one thread accesses shared resources at a time to prevent conflicts.
  Use synchronized keyword on methods or blocks.
- **Inter-thread Communication:**
  Threads communicate using wait (), notify (), and notify All () inside synchronized context.
  - wait() — thread waits and releases lock
  - notify() — wakes one waiting thread
  - notify All() — wakes all waiting threads

49. Introduction to File I/O in Java (java.io package)

Java's **File I/O** lets you **read from** and **write to** files using classes in the java.io package.

✅ Common Classes for File I/O:

- **File**
  Represents file or directory path (no content access).
- **FileReader / FileWriter**
  For reading/writing **text files** (characters).
- **BufferedReader / BufferedWriter**
  Wrap around FileReader/FileWriter for efficient reading/writing with buffering.
- **FileInputStream / FileOutputStream**
  For reading/writing **binary files** (bytes).

✅ Basic Steps for File I/O:

1. **Open** a stream (e.g., FileReader, FileWriter).
2. **Read or write** data.
3. **Close** the stream to release resources.

Example: Reading a text file

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt")))
{
String line;
while ((line = br.readLine()) != null)
{
```

```
System.out.println(line);
}
} catch (IOException e) {
e.printStackTrace();
}
```

50. File Reader and FileWriter Classes

- **FileReader**: Used to **read characters** from a file (text files).
- **FileWriter**: Used to **write characters** to a file (text files).

Usage:

```
// Reading from a file
FileReader fr = new FileReader("input.txt");
int ch;
while ((ch = fr.read()) != -1) {
System.out.print((char) ch);
}
fr.close();
// Writing to a file
FileWriter fw = new FileWriter("output.txt");
fw.write("Hello, World!");
fw.close();
```

Key points:

- Works with characters (not bytes).
- Use close () to release resources.
- For efficient reading/writing, wrap with BufferedReader/BufferedWriter.

51. BufferedReader and Buffered Writer

- BufferedReader**:** Reads text from a character-input stream efficiently by buffering characters.
  Provides readLine () method to read a whole line.
- BufferedWrite**r:** Writes text to a character-output stream efficiently by buffering characters.
  Provides newLine() method to write a line separator.

Usage:

```
// Reading lines efficiently
BufferedReader br = new BufferedReader(new FileReader("input.txt"));
String line;
while ((line = br.readLine()) != null) {
System.out.println(line);
```

```
}
br.close();
// Writing lines efficiently
BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"));
bw.write("Hello BufferedWriter");
bw.newLine();
bw.close();
```

52. Serialization and Deserialization

- **Serialization:**
  Converts a Java object into a byte stream to save it to a file or send over a network.
- **Deserialization:**
  Converts the byte stream back into a copy of the original Java object.

Key points:

- Class must implement Serializable interface.
- Use ObjectOutputStream for serialization.
- Use ObjectInputStream for deserialization.

Example:

```
// Serialization
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data.obj"));
out.writeObject(object);
out.close();

// Deserialization
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));
Object obj = in.readObject();
in.close();
```

53. Introduction to Collections Framework

1. **Exceptions:**
   - Checked (compile-time) and Unchecked (runtime).
2. **try, catch, finally, throw, throws:**
   - Handle and declare exceptions; cleanup code in finally.
3. **Custom Exceptions:**
   - Extend Exception or RuntimeException.
4. **Threads:**
   - Run concurrent tasks; create by extending Thread or implementing Runnable.
5. **Thread Life Cycle:**
   - New → Runnable → Running → Waiting/Blocked → Terminated.

6. **Synchronization & Communication:**
    - synchronized for mutual exclusion; wait (), notify () for thread coordination.
7. **File I/O (java.io):**
    - Use streams like FileReader, FileWriter, BufferedReader, BufferedWriter to read/write files.
8. **Serialization:**
    - Convert objects to/from byte streams; class must implement Serializable.
9. **Collections Framework:**
    - Data structures like List, Set, Queue, Map for efficient object storage and access.

54. List, Set, Map, and Queue Interfaces

List: Ordered collection, allows duplicates.
Example: ArrayList, LinkedList

Set: No duplicates, unordered or sorted.
Example: HashSet, TreeSet

Map: Key-value pairs, keys unique.
Example: HashMap, TreeMap

Queue: FIFO order, for processing elements.
Example: LinkedList, PriorityQueue

55. ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

ArrayList**:** Resizable array, fast random access, allows duplicates.

LinkedList**:** Doubly linked list, fast insert/delete, allows duplicates.

HashSet**:** Stores unique elements, no order, fast lookup.

TreeSet**:** Stores unique elements in sorted order (natural or comparator).

HashMap**:** Key-value pairs, no order, fast access by key.

TreeMap**:** Key-value pairs sorted by keys.

56. Iterators and ListIterators

- **Iterator:**
  Used to traverse **any** Collection (List, Set, etc.)
  Methods: hasNext (), next (), remove ()
- **ListIterator:**
  Used to traverse **List** in both directions (forward & backward)
  Extra methods: hasPrevious (), previous (), add(), set()

57. Streams in Java (InputStream, OutputStream)

- **InputStream:**
  Reads bytes from a source (e.g., file, network).
  Example: FileInputStream
- **OutputStream:**
  Writes bytes to a destination.
  Example: FileOutputStream

Used for: Handling **binary data** (images, files).
Must **close streams** after use.

58. Reading and Writing Data Using Streams

- **Reading:** Use InputStream (e.g., FileInputStream) and read () method to read bytes.
- **Writing:** Use OutputStream (e.g., FileOutputStream) and write () method to write bytes.

Example:

```
// Reading
FileInputStream in = new FileInputStream("input.txt");
int data;
while ((data = in. read()) != -1)
{
System.out.print((char) data);
}
in.close();

// Writing
FileOutputStream out = new FileOutputStream("output.txt");
out.write("Hello”. GetBytes());
out.close();
```

59. Handling File I/O Operations

1. Open a stream (e.g., FileReader, FileWriter, FileInputStream, FileOutputStream).
2. Read from or write to the file using methods like read (), write (), or readLine ().
3. Close the stream to release resources (use try-with-resources for automatic closing).

Example:

```
try (BufferedReader br = new BufferedReader (new FileReader("file.txt")))

{

String line;
while ((line = br. readLine ()) != null)
{
System.out.println(line);
}
} catch (IOException e) {
e.printStackTrace();
}
```