Einetic Technologies Private Limited (EINETIC) Interview preparation document

Einetic Technologies Private Limited (EINETIC) is a privately held IT company based in Kolkata, West Bengal. It was incorporated on November 25, 2020, and operates primarily in the business of IT consultancy, software design, development, and professional services for enterprise systems.

Official Website

- **The official website of Einetic Technologies Private Limited is: https://www.einetic.com/.internshala+1**

Services Offered

- **Enterprise system integration**
- **Software design and development**
- **Professional IT services**
- **Web and mobile app development**
- **Digital marketing support and management for brands' online presence.**

Notable Details

- **EINETIC helps businesses manage digital transitions by creating user-friendly websites and apps, managing online stores, and optimizing digital marketing via major social platforms.**
- **The company is described as being innovative and supportive, treating clients as partners in their digital journey.**

Key Perks for Employees (per job listings)

- **Flexible work hours**
- **Certificate programs**
- **5-day work week**
- **Complimentary snacks**

Company Overview

- **Company Name: Einetic Technologies Private Limited (EINETIC)**
- **Official Website: https://www.einetic.com/**
- **Location: Lake Town, Kolkata - 89**

Salary & Benefits

- **Salary Range: 2.4 – 4.8 LPA**
- **Monthly Salary: Rs. 20,000 – 40,000 per month with incentives and performance bonus**

Expectations from Freshers

- **Skill Sets Required:**
    - **JAVA**
    - **Analytical skills**
    - **Communication**
    - **Interpersonal skills**
- **Qualification: B.E, B.Tech, M.E, M.Tech, BCA, MCA, B.Sc, M.Sc**
- **Specialization: CS/IT**
- **Year of Passing: Any**
- **Academic Percentage: Above 60% throughout the education**
- **Education Gap: Allowed**
- **Backlogs: Not allowed**

Job Description (JD) Brief

- **Position Title: Software Engineer (SE)**
- **Required Skills: JAVA, Analytical, Communication, Interpersonal**
- **Type of Submission: Soft Copy of certificates required**
- **Bond: No bond**
- **Training Period: 12 months**
- **Probation: 1 – 1.5 years**

Interview Process and Rounds

- **Type: Hybrid (Virtual + Face to Face)**
- **Venue: Lake Town, Kolkata - 89**
- **Tentative Interview Date: 10th October 2025**
- **Interview Rounds for Freshers:**
    - **Round 1: HR Round**
    - **Round 2: Technical Round (Virtual)**
    - **Round 3: Technical Round 2 (Face to Face)**

Technical Round questions

What is Java?

Java is a high-level, object-oriented, class-based programming language developed by Sun Microsystems (now owned by Oracle) in 1995.
It is widely used for:

- **Web applications**
- **Mobile applications (especially Android)**
- **Enterprise applications**
- **Big Data and distributed systems**

Java's slogan is "Write Once, Run Anywhere (WORA)", which comes from its platform independence.

What is Platform Independence?

A platform means the combination of Operating System (OS) and hardware (e.g., Windows on Intel, Linux on ARM, macOS on M1).

Most traditional programming languages (like C, C++) are platform dependent:

- **The source code is compiled directly into machine code.**
- **This machine code is specific to the OS/CPU.**
- **If you want to run it on another OS, you must recompile it.**

But in Java, things work differently 👇

How Java achieves Platform Independence?

Java uses a two-step execution model:

1. **Compilation**
    - **Java source code (.java) is compiled by the Java Compiler (javac).**
    - **Instead of generating machine-specific code, it generates Bytecode (.class files).**
    - **Bytecode is a universal intermediate code that is not tied to any OS or hardware.**
2. **Execution**
    - **The Java Virtual Machine (JVM) runs the Bytecode.**
    - **The JVM is platform-specific (Windows JVM, Linux JVM, Mac JVM, etc.).**
    - **But the Bytecode is the same across all platforms.**

- JVM translates Bytecode into machine code for the host OS/CPU using Just-In-Time (JIT) compilation.

Diagram: Platform Independence

```
Java Program (Hello.java)
     |
     | javac
     v
Bytecode (Hello.class)
     |
     | JVM (platform-specific)
     v
Machine Code (runs on OS/CPU)
```

So:

- You write the code once in Java.
- It gets compiled into Bytecode.
- That Bytecode can run anywhere there is a JVM.

👉 That's why Java is called platform independent at the bytecode level.

Youtube : https://youtu.be/wsAbhPQKHvo?si=37K5dOqI8MRIUTDw

JVM — Java Virtual Machine

What it is: an *abstract execution engine* that runs Java bytecode.
Role: loads .class files (bytecode), verifies them, manages memory, and executes code (interpretation + JIT).
Main components:

- Class Loader Subsystem — finds and loads class files.
- Bytecode Verifier — checks code for security/format correctness.
- Runtime Data Areas — Heap, Method Area, Stacks, PC Registers, Native Method Stack.
- Execution Engine — Interpreter + JIT (Just-In-Time) compiler that turns hot bytecode into native machine code.
- Garbage Collector (GC) — reclaims memory.
- Native Interface (JNI) — lets JVM call native libraries.

Key point: JVM is *platform-specific* (there are Windows/Linux/macOS JVM implementations). It's the piece that makes bytecode runnable on a particular OS/CPU.

JRE — Java Runtime Environment

**What it is: the runtime package that provides everything needed to *run* Java programs.
Contains:**

- **A JVM implementation (platform-specific).**
- **Core class libraries (the Java API: java.lang, java.util, etc.).**
- **Supporting files and the java launcher.**

Used by: end users or production servers that only need to *execute* Java apps (not develop them).

Key point: JRE = JVM + standard libraries. JRE itself is platform-specific (because the JVM inside it is native).

JDK — Java Development Kit

**What it is: the full kit for developing Java applications.
Contains:**

- **Everything in the JRE (so you can run apps).**
- **Development tools: javac (compiler), jar, javadoc, javap, jdb, jshell, keytool, jlink, and others.**
- **Source files for core libraries (in many distributions).**

Used by: developers who write, compile, debug, and package Java applications.

Key point: JDK ⟶ JRE ⟶ JVM (JDK includes a JRE so you can compile + run locally).

How they work together (simple flow)

1. **You write Hello.java.**
2. **javac Hello.java (tool from JDK) produces Hello.class (bytecode).**
3. **java Hello (launcher in JRE) starts the JVM, which loads Hello.class, verifies and executes it.**

ASCII diagram:

```
[Source: Hello.java] --javac (JDK)--> [Bytecode: Hello.class]

        |

        +--> (distribute .class or .jar to any machine)

[Target machine] --java (JRE)--> JVM -> executes bytecode on that OS/CPU
```

Youtube : https://youtu.be/eaAqwTdUAAo?si=ZO-Cndy3mW50RQSP

The 3 Main OOP Principles (Java)

Java is based on Object-Oriented Programming (OOP), which models software as a collection of objects that interact.
The three most fundamental principles you'll always see are:

- **Inheritance**
- **Encapsulation**
- **Polymorphism**

(It is also mention Abstraction at the end since some interviews consider it the "fourth pillar.")

---

1. Inheritance

Definition:
Inheritance allows a class (child/subclass) to reuse properties and behavior from another class (parent/superclass).

👉 **It promotes code reusability and forms an "is-a" relationship.**

Example:

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

// Child class inherits Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // inherited from Animal
        d.bark();  // Dog-specific method
    }
}
```

Output:

```
This animal eats food.
Dog barks.
```

👉 Here, Dog inherits the eat() method from Animal.

Key points about Inheritance in Java:

- Supports single inheritance (one class extends only one class).
- Achieves multiple inheritance via interfaces.
- Improves code reusability, hierarchy, and readability.

---

2. Encapsulation

Definition:
Encapsulation is wrapping data (fields) and methods together in a single unit (class) and restricting direct access to some components.

👉 It hides implementation details and provides access via getters and setters (controlled access).

Example:

```
class BankAccount {
    // private data - hidden from outside
    private double balance;

    // public method to access balance
    public double getBalance() {
        return balance;
    }

    // public method to modify balance
    public void deposit(double amount) {
        if(amount > 0) {
            balance += amount;
        }
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        acc.deposit(5000);
        System.out.println("Balance: " + acc.getBalance());
    }
}
```

👉 **Here, balance is private (cannot be accessed directly), and can only be changed using deposit() or read via getBalance() — ensuring data security.**

Key points about Encapsulation:

- **Achieved using access modifiers (private, public, protected).**
- **Protects data from unauthorized access.**
- **Provides control over how variables are accessed/modified.**

---

3. Polymorphism

Definition:
Polymorphism means "many forms" — the ability of an object to take different forms or behave differently in different situations.

There are two types in Java:

(a) Compile-time Polymorphism (Method Overloading)

- **Same method name, different parameter lists (number/type).**

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

Usage:

```
Calculator c = new Calculator();
System.out.println(c.add(2, 3));       // calls int version
System.out.println(c.add(2.5, 3.7));   // calls double version
```

(b) Runtime Polymorphism (Method Overriding)

- A subclass provides its own implementation of a method defined in the parent.
- Achieved via overriding + dynamic method dispatch.

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

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

public class Main {
   public static void main(String[] args) {
      Animal a = new Dog(); // Upcasting
      a.sound(); // calls Dog's overridden method (runtime decision)
   }
}
```

Output:

```
Dog barks
```

👉 JVM decides which version of sound() to call at runtime (not compile-time).

---

4. Abstraction (often considered 4th pillar)

Definition:
Abstraction is hiding implementation details and exposing only essential features.
Achieved by:

- Abstract classes (using abstract keyword).
- Interfaces (Java 8+ allows default & static methods).

Example:

```
abstract class Vehicle {
   abstract void start();
}
```

```
class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with key");
    }
}
```

Youtube : https://youtu.be/m_MQYyJpIjg?si=QwZ-9xm4aknm4puM

1. Classes in Java

Definition:
A class is a blueprint/template from which objects are created.
It defines attributes (fields/variables) and behaviors (methods) of an object.

👉 **Think of a class as a design (like a car blueprint) and an object as the actual car.**

Syntax:

```
class Car {
    // fields (attributes)
    String color;
    String model;

    // method (behavior)
    void drive() {
        System.out.println("Car is driving...");
    }
}
```

---

2. Objects in Java

Definition:
An object is an instance of a class.
It represents a real-world entity with state (data) and behavior (methods).

👉 **Multiple objects can be created from the same class, each with its own data.**

Example:

```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car(); // object creation
        car1.color = "Red";
        car1.model = "BMW";

        Car car2 = new Car();
        car2.color = "Blue";
        car2.model = "Audi";

        // Access methods
        car1.drive();
        System.out.println(car1.color + " " + car1.model);

        car2.drive();
        System.out.println(car2.color + " " + car2.model);
    }
}
```

Output:

```
Car is driving...
Red BMW
Car is driving...
Blue Audi
```

---

3. Constructors in Java

Definition:

A constructor is a special method that initializes objects.

- **It has the same name as the class.**
- **It does not have a return type (not even void).**
- **It is called automatically when an object is created.**

Types of Constructors

1. **Default Constructor (provided by Java if none is written)**

```
class Student {
    String name;
    int age;

    // default constructor
    Student() {
        System.out.println("Student object created");
    }
}
```

2. Parameterized Constructor (takes arguments to initialize object)

```
class Student {
    String name;
    int age;

    Student(String n, int a) {  // parameterized constructor
        name = n;
        age = a;
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Rahul", 22);
        System.out.println(s1.name + " " + s1.age);
    }
}
```

Output:

```
Rahul 22
```

3. Constructor Overloading (multiple constructors with different parameters)

```
class Student {
    String name;
    int age;

    Student() {
        name = "Unknown";
        age = 0;
    }

    Student(String n, int a) {
        name = n;
        age = a;
    }
}
```

4.Access Modifiers in Java

Access modifiers define visibility/scope of classes, methods, and variables.

Types of Access Modifiers:

1. **public – accessible from anywhere.**
2. **protected – accessible within the same package and by subclasses (even in different packages).**
3. **default (no keyword) – accessible only within the same package.**
4. **private – accessible only within the same class.**

Example:

```
class Student {
   public String name;      // accessible anywhere
   private int age;         // only within Student class
   protected String branch; // package + subclass
   String college;          // default (package level)

   public void setAge(int a) { // public method to access private field
      if (a > 0) {
         age = a;
      }
   }

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

Youtube: https://youtu.be/IUqKuGNasdM?si=_NriJ_UqFgO0il6f

What is an Exception?

- **An exception is an unexpected event that disrupts the normal flow of a program during runtime.**
- **It is an object that describes an error or unexpected situation.**
- **In Java, exceptions are represented by classes (all extend from Throwable).**

👉 **Example:**

- **Dividing by zero (ArithmeticException)**
- **Accessing invalid array index (ArrayIndexOutOfBoundsException)**
- **Opening a file that does not exist (FileNotFoundException)**

Exception Handling in Java

Definition:
Exception Handling is a mechanism to handle runtime errors so that the program continues to run without crashing.

Java provides a robust try-catch-finally-throw-throws mechanism.

Exception Handling Keywords

try

Block of code that may cause an exception.

```
try {
   int result = 10 / 0; // risky code
}
```

catch

Block that handles the exception.

```
catch (ArithmeticException e) {
   System.out.println("Division by zero is not allowed");
}
```

finally

Block that always executes (whether exception occurs or not). Commonly used for cleanup (closing files, DB connections).

```
finally {
   System.out.println("Finally block executed");
}
```

throw

Used to explicitly throw an exception.

```
throw new IllegalArgumentException("Invalid input");
```

throws

Declares exceptions that a method may throw.

```
void readFile() throws IOException {
   // file reading logic
}
```

Types of Exceptions in Java

Exceptions in Java fall into two main categories:

1. Checked Exceptions

- **Checked at compile-time.**
- **If not handled, the program won't compile.**
- **Usually caused by external resources (files, DB, network).**
- **Must be either caught or declared using throws.**

Examples:

- **IOException**
- **SQLException**
- **ClassNotFoundException**
- **FileNotFoundException**

Example code:

```
import java.io.*;

class Example {
   public static void main(String[] args) {
      try {
         FileReader fr = new FileReader("test.txt"); // file may not exist
      } catch (FileNotFoundException e) {
         System.out.println("File not found!");
      }
   }
}
```

---

2. Unchecked Exceptions (Runtime Exceptions)

- **Checked at runtime, not compile-time.**
- **Usually due to programming mistakes (bad logic, invalid inputs).**
- **Compiler doesn't force handling.**

Examples:

- **ArithmeticException**
- **NullPointerException**
- **ArrayIndexOutOfBoundsException**
- **NumberFormatException**

Example code:

```
class Example {
   public static void main(String[] args) {
      int a = 10, b = 0;
      int result = a / b; // causes ArithmeticException
   }
}
```

---

3. Errors (not Exceptions, but part of Throwable)

- **Represent serious problems beyond the application's control.**
- **Should not be caught in most cases.**
- **Examples:**
    - **OutOfMemoryError**
    - **StackOverflowError**
    - **VirtualMachineError**

Exception Hierarchy

```
           Throwable
           /     \
      Exception      Error
         |
  --------------------
  |                 |
Checked (Compile)   Unchecked (Runtime)
(IOException, etc.)  (NullPointerException, etc.)
```

Example: Full Exception Handling

```
public class ExceptionDemo {
   public static void main(String[] args) {
        try {
           int[] arr = {1, 2, 3};
           System.out.println(arr[5]); // risky code
        } catch (ArrayIndexOutOfBoundsException e) {
           System.out.println("Invalid index accessed: " + e);
        } finally {
           System.out.println("This will always run");
        }
    }
}
```

Output:

```
Invalid index accessed: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds
This will always run
```

Youtube : https://youtu.be/1XAfapkBQjk?si=jARZMHHOHfzcPw0X

Basic Difference between ArrayList and LinkedList

- **ArrayList:**
    - **Backed by a dynamic array.**
    - **Stores elements in contiguous memory locations.**
    - **Provides fast random access (get by index).**
- **LinkedList:**
    - **Backed by a doubly linked list.**
    - **Each element (called a node) stores:**
        - **The data.**
        - **A reference to the previous node.**
        - **A reference to the next node.**
    - **Provides fast insertion/deletion (when position is known).**

2. Memory Structure

- **ArrayList:**
  - **Uses a resizable array under the hood.**
  - **When it grows beyond capacity, a new larger array is created and old elements are copied.**
  - **More memory-efficient than LinkedList because it only stores data and array indexes.**
- **LinkedList:**
  - **Uses node objects.**
  - **Each node stores extra references (prev and next).**
  - **Consumes more memory than ArrayList.**

---

3. Performance Comparison

Operation	ArrayList	LinkedList
Access by index (get/set)	**O(1)** → because of direct index lookup	**O(n)** → traversal required from head/tail
Insertion (at end)	Amortized **O(1)** (sometimes resizing occurs)	**O(1)** (if inserting at end and reference is known)
Insertion (in middle/start)	**O(n)** → elements need to be shifted	**O(1)** (if you already have a reference to the node, otherwise traversal is **O(n)**)
Deletion (by index)	**O(n)** → elements need shifting	**O(1)** (if reference known, otherwise O(n) for traversal)
Search (contains)	**O(n)** → linear search	**O(n)** → linear search

---

4. When to Use What

- **Use ArrayList when:**
  - **You need fast random access.**
  - **Your application is mostly read-heavy (more searches and retrievals).**
  - **Insertion/deletion happens mostly at the end of the list.**
- **Use LinkedList when:**
  - **You need fast insertion and deletion at arbitrary positions (not just at the end).**
  - **Your application is write-heavy.**
  - **You frequently add/remove elements from the beginning or middle of the list.**

5. Example Code

```
import java.util.*;


public class ListExample {

    public static void main(String[] args) {

        // ArrayList Example

        List<String> arrayList = new ArrayList<>();

        arrayList.add("A");

        arrayList.add("B");

        arrayList.add("C");

        System.out.println("ArrayList: " + arrayList);


        // LinkedList Example

        List<String> linkedList = new LinkedList<>();

        linkedList.add("X");

        linkedList.add("Y");

        linkedList.add("Z");

        System.out.println("LinkedList: " + linkedList);


        // Access by index

        System.out.println("ArrayList get(1): " + arrayList.get(1)); // O(1)

        System.out.println("LinkedList get(1): " + linkedList.get(1)); // O(n)

    }

}
```

YOUTUBE : https://youtu.be/QWMyhFUtFHo?si=QGyTLy7hprOdrOXQ

What is a Set in Java?

- **Set is a collection that does not allow duplicate elements.**
- **Java provides HashSet, TreeSet, and LinkedHashSet implementations.**

1.HashSet

Definition:
HashSet is a collection that stores unique elements using a hash table (actually a HashMap internally).

Key Features:

- **No duplicates allowed.**
- **No ordering guaranteed (elements may appear in random order).**
- **Allows one null element.**
- **Faster operations (O(1) average for add, remove, contains) because it uses hashing.**
- **Implements: Set, Collection, Serializable, Cloneable.**

Example:

```
import java.util.HashSet;


public class HashSetExample {

    public static void main(String[] args) {

        HashSet<String> set = new HashSet<>();

        set.add("Apple");

        set.add("Banana");

        set.add("Apple"); // duplicate, ignored

        set.add(null);    // null allowed


        System.out.println(set);

    }

}
```

Output: (order may vary)

```
[null, Apple, Banana]
```

2.TreeSet

Definition:
TreeSet is a sorted set that stores unique elements in ascending order (by default). Internally, it uses a Red-Black tree (self-balancing binary search tree).

Key Features:

- **No duplicates allowed.**
- **Elements are automatically sorted (natural ordering or custom comparator).**
- **Does NOT allow null elements (throws NullPointerException if you try).**
- **Slower than HashSet (O(log n) for add, remove, contains).**
- **Implements: Set, SortedSet, NavigableSet.**

Example:

```
import java.util.TreeSet;


public class TreeSetExample {

    public static void main(String[] args) {

        TreeSet<String> set = new TreeSet<>();

        set.add("Banana");

        set.add("Apple");

        set.add("Mango");


        System.out.println(set);

    }

}
```

Output: (sorted automatically)

```
[Apple, Banana, Mango]
```

Youtube : https://youtu.be/mI4eNh5iIpE?si=D4jas7kBKvh82NtP

Multithreading

1. What is Multithreading?

Definition:
A thread is the smallest unit of execution in a program.
Multithreading means running multiple threads concurrently within a single program to perform parallel tasks.

Why use multithreading?

- **Improves CPU utilization.**
- **Allows parallel execution of tasks.**
- **Useful for responsive GUI applications, server requests, background tasks.**

Thread in Java:

- **Java provides the Thread class and Runnable interface to create threads.**

---

2. Ways to Create Threads in Java

(a) Extending the Thread class

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
        }
    }
}


public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();


        t1.start(); // start a new thread
        t2.start();
    }
}
```

Key points:

- Override run() method.
- Use start() to begin execution (do not call run() directly).

(b) Implementing the Runnable interface

```
class MyRunnable implements Runnable {
   public void run() {
      for(int i = 1; i <= 5; i++) {
         System.out.println(Thread.currentThread().getName() + " : " + i);
      }
   }
}

public class Main {
   public static void main(String[] args) {
      Thread t1 = new Thread(new MyRunnable());
      Thread t2 = new Thread(new MyRunnable());

      t1.start();
      t2.start();
   }
}
```

Advantages:

- Can extend another class (Java doesn't support multiple inheritance).
- More flexible.

---

3. Thread Lifecycle

```
New -> Runnable -> Running -> Waiting/Blocked -> Terminated
```

- New: Thread object created, not started yet.
- Runnable: Ready to run, waiting for CPU scheduling.
- Running: JVM executes run() method.
- Waiting/Blocked: Thread waits for resource or notification.
- Terminated: Thread has completed execution.

---

4. Synchronization in Java

Definition:
Synchronization is a mechanism to control access to shared resources by multiple threads to prevent data inconsistency or race conditions.

Why needed?

- **When multiple threads access and modify shared data, results may be unpredictable.**
- **Synchronization ensures only one thread executes critical code at a time.**

Example without synchronization

```
class Counter {
    int count = 0;


    void increment() {
        count++; // not thread-safe
    }
}


public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();


        Thread t1 = new Thread(() -> { for(int i=0;i<1000;i++) c.increment(); });
        Thread t2 = new Thread(() -> { for(int i=0;i<1000;i++) c.increment(); });


        t1.start(); t2.start();
        t1.join(); t2.join(); // wait for threads to finish


        System.out.println("Count: " + c.count); // may not be 2000!
    }
}
```

Problem: Race condition — both threads may read/update count simultaneously.

Solution: Synchronization

```
class Counter {
    int count = 0;
    synchronized void increment() { // critical section
        count++;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> { for(int i=0;i<1000;i++) c.increment(); });
        Thread t2 = new Thread(() -> { for(int i=0;i<1000;i++) c.increment(); });

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Count: " + c.count); // always 2000
    }
}
```

Youtube : https://youtu.be/r_MbozD32eo?si=XAjOl9H5G7tqDeXv

Java 8 features

1. Lambda Expressions

Definition:
A lambda expression is a short block of code that takes parameters and returns a value. It provides a clear and concise way to implement functional interfaces (interfaces with a single abstract method).

Syntax:

```
(parameters) -> expression

or

(parameters) -> { statements; }
```

Example 1: Simple lambda

```
// Functional interface
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greeting g = () -> System.out.println("Hello, Java 8!");
        g.sayHello();
    }
}
```

Output:

```
Hello, Java 8!
```

Example 2: Lambda with parameters

```
interface MathOperation {
    int operation(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        MathOperation add = (a, b) -> a + b;
        MathOperation multiply = (a, b) -> a * b;

        System.out.println(add.operation(5, 3));      // 8
        System.out.println(multiply.operation(5, 3));  // 15
    }
}
```

---

2. Functional Interfaces

Definition:
A functional interface is an interface with exactly one abstract method.

- **Can have default and static methods.**
- **Annotated with @FunctionalInterface (optional but recommended).**

Common Java 8 functional interfaces:

Interface	**Method Signature**	**Use Case**
Runnable	void run()	Thread execution
Comparator<T>	int compare(T o1, T o2)	Sorting

Interface	**Method Signature**	**Use Case**
Predicate<T>	boolean test(T t)	Condition check
Function<T,R>	R apply(T t)	Transform data
Consumer<T>	void accept(T t)	Process without returning
Supplier<T>	T get()	Supply object

Example using Predicate:

```
import java.util.function.Predicate;


public class Main {
   public static void main(String[] args) {
      Predicate<Integer> isEven = n -> n % 2 == 0;


      System.out.println(isEven.test(4)); // true
      System.out.println(isEven.test(7)); // false
   }
}
```

---

3. Stream API

Definition:
Stream API allows functional-style operations on collections (filter, map, reduce). Streams do not store data, but process data from a source like List, Set, etc.

Example:

```
import java.util.*;
import java.util.stream.*;


public class Main {
   public static void main(String[] args) {
      List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");


      // Filter names starting with 'A', convert to uppercase
      List<String> result = names.stream()
                          .filter(n -> n.startsWith("A"))
                          .map(String::toUpperCase)
                          .collect(Collectors.toList());


      System.out.println(result); // [ALICE]
```

```
    }
}
```

Other Stream operations:

- **forEach() – iterate elements**
- **map() – transform data**
- **filter() – select elements**
- **reduce() – combine elements**
- **sorted() – sort elements**
- **distinct() – remove duplicates**

---

4. Default and Static Methods in Interfaces

Before Java 8, interfaces could only have abstract methods.
Java 8 added:

- **Default methods – provide default implementation**
- **Static methods – can be called using interface name**

Example:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle started");
    }

    static void info() {
        System.out.println("This is a vehicle interface");
    }
}

class Car implements Vehicle {}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();          // default method
        Vehicle.info();     // static method
    }
```

}

---

5. Method References

Definition:
Shortcut for lambda expressions when you just want to refer to a method instead of writing a lambda.

Syntax examples:

- **ClassName::staticMethod**
- **instance::instanceMethod**
- **ClassName::new (constructor reference)**

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(System.out::println);
```

Youtube : https://youtube.com/playlist?list=PLsyeobzWxl7qbvNnJKjYbkTLn2w3eRy1Q&si=5WMvpkXXIT8OyxSk