**Functional Programming**

**─**

Contents

[**1. Introduction to Functional Programming** 2](#_Toc183876257)

[# Pure Functions 2](#_Toc183876258)

[**2. Functional Interface** 4](#_Toc183876259)

[**3. Lambda Expression** 5](#_Toc183876260)

[# Syntax of lambda function 6](#_Toc183876261)

[# Use cases of lambda 7](#_Toc183876262)

[**4. Predicate** 8](#_Toc183876263)

[# Use cases of Predicate 9](#_Toc183876264)

[**5. Stream** 10](#_Toc183876265)

[# Stream methods 12](#_Toc183876266)

[# Parallel Stream 14](#_Toc183876267)

[**6. Functions as First-Class Citizens** 15](#_Toc183876268)

[# Use cases 16](#_Toc183876269)

[**7. Useful functional Interfaces** 17](#_Toc183876270)

[**8. Interview Questions** 21](#_Toc183876271)

## **1. Introduction to Functional Programming**

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

Key Principles:

* **Immutability:** You focus on the transformation of data without directly modifying it. New data is produced instead.
* **First-Class Functions:** Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, or returned from other functions.
* **Declarative Style:** Focus on "what to do" rather than "how to do it." (Just like languages like SQL)
* **Side-Effect Free:** Functions should not modify any state or interact with the outside world.

Real World Example: -

Imagine you're cooking a recipe. In traditional cooking (like traditional programming), you follow a step-by-step process (making it imperative approach): get ingredients, cut the vegetables(modifying their state) , mix them, cook them, and serve.

**Functional programming** is like a different approach to cooking. Instead of following a strict recipe, you focus on the ingredients and the final dish (declarative approach). You prepare all the ingredients beforehand. You might use pre-made sauces, ready-to-eat vegetables, or even a microwave to speed things up.

Benefits: -

* Clean Code: Reduces boilerplate code by leveraging lambda expressions and streams.
* Immutability: Ensures thread safety by avoiding shared mutable state.
* Parallelism: Simplifies parallel operations using parallel streams.

### # Pure Functions

A pure function is a function where the output depends solely on the input parameters and has no side effects (e.g., no modification of global variables or I/O operations).

Characteristics of Pure Functions:

* Deterministic: Same input always produces the same output.
* No Side Effects: Does not alter the state or perform external interactions.

public class PureFunctionExample {

// Pure function  
 public static int add(int a, int b) {  
 return a + b; // Depends only on input  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(*add*(5, 3)); // Output: 8  
 }  
}

Why Pure Functions Matter:-

* Immutability: Helps maintain an unchangeable state.
* Thread Safety: Pure functions are inherently thread-safe as they don't modify shared data.
* Ease of Testing: Pure functions are predictable and easier to test.

## **2. Functional Interface**

Abstract method

An abstract method is a method that is declared without an implementation. It allows defining behaviors without specifying how they are performed.

Sometimes we may come across situation where we cannot provide implementation to methods in a class, but want to leave implementation to class that extends it. There we use abstract method.

Functional Interface

It is an interface that contains exactly one abstract method. Thus they can have only one functionality to exhibit.

**Key Points:**

* Introduced in Java 8 to facilitate functional programming.
* Lambda expressions and method references rely on functional interfaces.
* Though they may have multiple default and static methods.
* Represented with annotation @FunctionalInterface(acts as compile-time check)

@FunctionalInterface  
interface Calculator {  
 int compute(int a, int b); // Single abstract method  
}  
  
class TempDoubleCalc implements Calculator{  
 public int compute(int a, int b){  
 return 2\*(a+b);  
 }  
}  
  
public class FunctionalInterfaceExample {  
 public static void main(String[] args) {  
 TempDoubleCalc temp = new TempDoubleCalc();  
 System.*out*.println(temp.compute(2,3)); //10  
 }  
}

## **3. Lambda Expression**

A lambda expression is a concise way to represent an anonymous function. It provides an implementation of a functional interface’s single abstract method directly, without using an anonymous class.

Why Lambda Expressions?

* Conciseness: Reduce boilerplate code.
* Readability: Improves code clarity for simple operations.
* Functional Programming: Enables functional style programming in Java by working seamlessly with functional interfaces.

Ex, in previous example of functional interface, to use Calculator interface we first had to create it’s class and then use, that either could be done by anonymous class or better with lambda function

@FunctionalInterface  
interface Calculator {  
 int compute(int a, int b); // Single abstract method  
}  
  
public class FunctionalInterfaceExample {  
 public static void main(String[] args) {  
  
 //With anonymous class  
 Calculator tempC = new Calculator() {  
 @Override  
 public int compute(int a, int b) {  
 return 2\*(a+b);  
 }  
 };  
 System.*out*.println(tempC.compute(2,3)); //10

//With lambda function  
 Calculator TempDoubleCalc = (a,b)-> 2\*(a+b) ;  
 System.*out*.println(TempDoubleCalc.compute(2,3)); //10  
 }  
}

Lambdas are more concise and can only be used for single-method interfaces, while anonymous classes are verbose and can implement multiple methods.

### # Syntax of lambda function

(parameters) -> expression/body

* Parameters: The input arguments (can be zero or more).
* Arrow Token (->): Separates parameters from the body.
* Body: The code to be executed, which can be:
  + A single expression.
  + A block of code enclosed in {}.

1. Lambda with single parameter

name -> System.out.println(name);

1. Lambda with multiple parameter

(a, b) -> a + b;

1. Lambda with no parameter

() -> System.out.println("No parameters!");

1. Lambda with a Single Statement (No Braces Required):

x -> x \* x;

1. Lambda with a Block of Code

(x, y) -> {  
int result = x + y;  
 return result;  
};

When and where to use Lambda Expressions?

* Use lambdas for simple, short-lived tasks, such as filtering a collection or setting up an event handler.
* Combine lambdas with functional interfaces like Predicate, Consumer, and Function.
* Use lambdas when passing behavior as arguments.
* Avoid lambdas for **complex logic** requiring multiple lines of code.

### # Use cases of lambda

1. With for each loop

public class temp {  
 public static void main(String[] args) {  
  
 //Using with for each loop  
 List<String> names = Arrays.*asList*("Raj","Kamal","Mohan");  
 names.forEach(name -> System.*out*.println(name));  
   
 //For Each with multiple lines  
 List<Integer> nums = Arrays.*asList*(1,2,3,5,8);  
 nums.forEach(n->{  
 System.*out*.println("number is " + n);  
 if(n%2==0)  
 System.*out*.printf("%d is even \n",n);  
 });  
 }  
}

1. In comparator

public class temp {  
 public static void main(String[] args) {  
  
 List<String> names = Arrays.*asList*("Kamal", "Ritu","Raj","Mohan");  
  
 //Normal comparator  
 Comparator<String> stringComp = (String s1,String s2)  
 -> Integer.*compare*(s1.length(),s2.length());  
  
 //Sort string according to length  
 Collections.*sort*(names,stringComp);  
 System.*out*.println(names); //[Raj, Ritu, Kamal, Mohan]  
   
 List<String> names2 = Arrays.*asList*("Kamal", "Ritu","Raj","Mohan");  
 //Direct using lambda expression  
 Collections.*sort*(names2,(s1,s2) -> Integer.*compare*(s1.length(),s2.length()));  
 System.*out*.println(names2); //[Raj, Ritu, Kamal, Mohan]  
 }  
}

1. Using with other Functional Interface

public class temp {  
 public static void main(String[] args) {  
 List<String> names = Arrays.*asList*("Kamal","Raj","Mohan");  
 Consumer<String> method = n-> System.*out*.println(n);  
 names.forEach(method); //Kamal,Raj,Mohan  
 //can also do  
 names.forEach(System.*out*::println); //Kamal,Raj,Mohan  
 }  
}

## **4. Predicate**

A Predicate is a functional interface introduced in Java 8 as part of the java.util.function package. It represents a single-argument function that returns a boolean value.

**Key Characteristics:**

* Single abstract method: test(T t)
* Designed for evaluating conditions (boolean-returning logic).
* Commonly used in stream processing and filtering operations.
* Has multiple default and static methods like negate, or , and , equals.

Syntax:

Predicate<T> predicate = (T t) -> booleanExpression;

public class LearnPredicate {  
 public static void main(String[] args) {  
 Predicate<Integer> isPassed = marks -> marks>39;  
 System.*out*.println(isPassed.test(44)); //true  
 System.*out*.println(isPassed.test(12)); //false  
   
 //Example 2  
 Predicate<Integer> isEven = n->n%2==0;  
 System.*out*.println(isEven.test(34)); //true  
 System.*out*.println(isEven.test(11)); //false  
 }  
}

Can combine 2 or more predicate using and, or. And get negation of predicate condition using negate

public class LearnPredicate {  
 public static void main(String[] args) {  
 Predicate<String> startWithS = name->name.toLowerCase().charAt(0)=='s';  
 Predicate<String> endsWithA = name->name.charAt(name.length()-1)=='a';  
 Predicate<String> validName = startWithS.and(endsWithA);  
  
 System.*out*.println(validName.test("Shiva")); //true  
 System.*out*.println(validName.test("Shankar")); //false  
  
 //using or  
 Predicate<String> easyValidCondition = startWithS.or(endsWithA);  
 System.*out*.println(easyValidCondition.test("Shivam")); //true  
 System.*out*.println(easyValidCondition.test("Kanha")); //true  
 }  
}

### # Use cases of Predicate

1. Filtering collection using stream

public class LearnPredicate {  
 public static void main(String[] args) {  
 List<String> names = Arrays.*asList*("Alice", "Bob", "Charlie", "");  
 Predicate<String> isNonEmpty = str -> !str.isEmpty();  
  
 names.stream()  
 .filter(isNonEmpty)  
 .forEach(System.*out*::println); //Alice,Bob,Charlie  
 }  
}

1. Validating input

public class LearnPredicate {  
 public static void main(String[] args) {  
 Predicate<String> isEmailValid = email -> email.contains("@") && email.endsWith(".com");

System.*out*.println(isEmailValid.test("test@example.com")); // Output: true  
 System.*out*.println(isEmailValid.test("invalid-email")); // Output: false  
 }  
}

## **5. Stream**

A Stream in Java is a sequence of elements that supports functional-style operations for processing data. It is part of the Java 8 Streams API (java.util.stream package).

**Key Features:-**

* A Stream does not store data; it processes data from a source (e.g., collections, arrays).
* It does not modify data of source.
* Streams support lazy evaluation, meaning operations are executed only when necessary.
* They allow for declarative programming, focusing on "what to do" rather than "how to do it."
* Once stream is called, it cannot be used again

public class LearnStream {  
 public static void main(String args[]){  
 List<Integer> nums = Arrays.*asList*(2, 4, 6, 8);  
  
 //Creating a stream  
 Stream<Integer> numStream = nums.stream();  
  
 //Using Stream first time  
 numStream.forEach(n-> System.*out*.println(n));//2 4 6 8  
  
 //using same stream second time will give error  
 numStream.forEach(n-> System.*out*.println(n));  
 // java.lang.IllegalStateException: stream has already been operated upon or closed  
 }  
}

How streams are better than loop?

1. **Streams are declarative**: For Loop require you write detailed code to handle the logic of iteration, data processing, and aggregation. But in streams you specify what should be done, not how to do it. The internal mechanics of iteration and execution are abstracted.
2. **Laziness and Efficiency:** In a for loop, every iteration and operation is executed immediately and eagerly. This can lead to unnecessary computations, especially if intermediate results are not needed. Streams are lazy by nature. Operations like filter, map, etc., are not executed until a terminal operation (e.g., collect, forEach, count) is invoked.
3. **Functional Composition and Pipelining**: To perform multiple operations (e.g., filtering, mapping, sorting) with a for loop, you must write separate steps or nested loops. This leads to verbose and harder-to-maintain code. Streams allow **pipelining**, where multiple operations can be chained together in a single pipeline. This reduces code verbosity and improves readability.
4. **Immutable Operations:** Using a for loop often involves modifying external variables or collections, which can lead to side effects and bugs. Streams encourage **immutability** by working on a pipeline of data rather than modifying external state. Data processing is done within the pipeline, ensuring that the original collection remains unmodified.
5. **Parallelism and Performance**: By default, a for loop runs sequentially. If you want to parallelize it, you must explicitly handle multithreading, synchronization, and other complexities. Streams provide **built-in parallelism** via the parallelStream() method. You can split the workload across multiple threads without writing custom multithreading code.
6. **Built-in Optimization**: In For loop, programmer is responsible for writing the iteration logic, and any optimizations must be implemented manually. Streams are optimized internally by the JVM. For example, the stream pipeline can merge or reorder operations for better performance.
7. **Readability and Maintainability**: Streams provide a declarative syntax that is more concise and easier to read.

Stream lifecycle

1. **Source**: Streams are created from data sources like collections, arrays, or I/O channels.

Stream<Integer> stream = Arrays.*asList*(1, 2, 3).stream();

1. **Intermediate Operations**: Transform or filter the stream (e.g., filter, map). These are lazy and do not process data immediately.

* filter(): Filters elements based on a specified condition.
* map(): Transforms each element in a stream to another value.
* sorted(): Sorts the elements of a stream.

stream.filter(n -> n > 2).map(n -> n \* 2);

1. **Terminal Operations:** Produce a result or side effect (e.g., forEach, collect). These trigger execution of the stream pipeline.

* collect(): It is used to return the result of the intermediate operations performed on the stream.
* forEach(): It iterates all the elements in a stream.
* reduce(): It is used to reduce the elements of a stream to a single value.

stream.forEach(System.*out*::println);

### # Stream methods

1. **filter(predicate)** : Filters element based on condition.

public class LearnStream {  
 public static void main(String args[]){  
 List<Integer> nums = Arrays.*asList*(3, 4, 7, 6, 8);  
  
 //filter even nums  
 Stream<Integer> numStream = nums.stream();  
 Stream<Integer> evenNumStream = numStream.filter(n->n%2==0);  
 evenNumStream.forEach(System.*out*::println); //4 6 8  
 }  
}

1. **map(Function)** : Transform each element.

public class LearnStream {  
 public static void main(String args[]){  
 List<String> names = Arrays.*asList*("Ajay", "Kamal", "Mukesh", "Raju");  
  
 //get List of lengths of names  
 Stream<String> nameStream = names.stream();

//can also do nameStream.map(String::length)  
 Stream<Integer> getNamesLength = nameStream.map(n->n.length());   
 getNamesLength.forEach(System.*out*::println); //4 5 6 4  
 }  
}

1. **sorted()** : Returns a stream consisting of the elements of the stream passed, sorted according to the natural order

public class LearnStream {  
 public static void main(String args[]){  
 List<String> names = Arrays.*asList*("Kamal", "Ram", "Mukesh", "Raju");  
  
 names.stream()  
 .sorted((s1,s2)->Integer.*compare*(s1.length(),s2.length()))  
 .forEach(n-> System.*out*.println(n)); // Ram Raju Kamal Mukesh  
  
  
 List<Integer> nums = Arrays.*asList*(53,21,41,5);  
 nums.stream().sorted().forEach(System.*out*::println);//5 21 41 53  
 }  
}

1. **forEach(Consumer):** Perform an action on each item.

public class LearnStream {  
 public static void main(String args[]){  
 List<String> names = Arrays.*asList*("Kamal", "Ram", "Raju");  
  
 names.stream().forEach(a-> System.*out*.println(a));   
 //Kamal Ram Raju  
 }  
}

1. **collect(Collector) :** Collects elements into a collection or data structure, . It takes a Collector as an argument, which defines how to accumulate the elements and produce the final result.

Common collectors: Collectors.toList(), Collectors.toSet(), Collectors.toMap(), Collectors.joining()(into a String)

public class LearnStream {  
 public static void main(String args[]){  
 List<String> names = Arrays.*asList*("Kamal", "Ram", "Raju");  
  
 List<Integer> namesLength = names.stream()  
 .map(n->n.length())  
 .collect(Collectors.*toList*());  
  
 System.*out*.println(namesLength); //[5,3,4]  
  
 String combinedName = names.stream()  
 .collect(Collectors.*joining*());  
 System.*out*.println(combinedName); //KamalRamRaju  
 }  
}

1. **reduce(BinaryOperator)** : Combines elements to produce a single result.

**Syntax**: T reduce(T identity, BinaryOperator<T> accumulator);

**Identity**: The initial value for the reduction. **accumulator:** A binary operator that takes two elements of type T and returns a single element of type T

public class LearnStream {  
 public static void main(String args[]){  
 List<Integer> nums = Arrays.*asList*(3,1,4,5,6);  
 int total = nums.stream().reduce(0,(a,b)->a+b);  
 System.*out*.println(total); //19  
 }  
}

### # Parallel Stream

* A parallel stream is a stream that splits its elements into multiple substreams and processes them in parallel.
* This can significantly improve performance for operations on large data sets or computationally intensive tasks.

Syntax:

//Using parallel stream  
List<Integer> numbers = List.*of*(1, 2, 3, 4, 5);  
numbers.parallelStream().forEach(System.*out*::println);  
  
//converting a sequential stream  
numbers.stream().parallel().forEach(System.*out*::println);

How parallel streams work

1. **Data Splitting:** The data source is divided into chunks (substreams).
2. **Task Execution:** Each chunk is processed independently in separate threads using the **Fork/Join Framework**.
3. **Result Merging:** Results from all threads are combined into a final result.

List<Integer> numbers = List.*of*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
  
int sum = numbers.parallelStream()  
 .filter(n -> n % 2 == 0)  
 .reduce(0, Integer::*sum*);  
  
System.*out*.println(sum); // Output: 30

Use parallel stream for large dataset, computational intensive task where there is no dependency between the elements.

## **6. Functions as First-Class Citizens**

In programming, treating functions as first-class citizens means they can be passed as arguments, returned from other functions, stored in variables, and manipulated like any other object. In Java, this concept became fully realized with the introduction of functional programming constructs in Java 8.

**Key Features:-**

* Function can be stored in variables.
* Function can be passed as parameter to other function.
* Function can be returned from method.
* Functional Interface like Supplier, Consumer, Function and BiFunction represents function like behaviour.

Functional Interfaces for First-Class Functions

1. **Consumer**: Accepts a single input and performs an operation.

public class LearnFunction {  
 public static void main(String[] args) {  
 Consumer<String> print = System.*out*::println;  
 //abstract method is accept  
 print.accept("Hello, Consumer!"); // Output: Hello, Consumer!  
 }  
}

1. **Supplier**: Takes no arguments and supplies a result

public class LearnFunction {  
 public static void main(String[] args) {  
 Supplier<Double> random = Math::*random*; //abstract method is get  
 System.*out*.println(random.get()); // Output: Random number  
 }  
}

1. **Function**: Takes one argument and returns a result.

public class LearnFunction {  
 public static void main(String[] args) {  
 Function<String, Integer> length = String::length;  
 System.*out*.println(length.apply("Functional")); // Output: 10  
 }  
}

1. BiFunction: Takes two arguments and returns a result.

public class LearnFunction {  
 public static void main(String[] args) {  
 BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;  
 System.*out*.println(add.apply(5, 10)); // Output: 15  
 }  
}

### # Use cases

1. Assigning function to a variable

public class LearnFunction {  
 public static void main(String[] args) {  
 // Assigning a lambda function to a variable  
 Function<String, Integer> stringLength = str -> str.length();  
 System.*out*.println(stringLength.apply("Hello")); // Output: 5  
 }  
}

1. Passing function as argument

public class LearnFunction {  
 public static void main(String[] args) {  
 Predicate<Integer> isEven = number -> number % 2 == 0;  
 System.*out*.println(*checkNumber*(10, isEven)); // Output: true  
 }  
  
 static boolean checkNumber(int num, Predicate<Integer> condition) {  
 return condition.test(num);  
 }  
}

1. Returning function from method

public class LearnFunction {  
 public static void main(String[] args) {  
 Function<Integer, Integer> square = *getSquareFunction*();  
 System.*out*.println(square.apply(5)); // Output: 25  
 }  
  
 static Function<Integer, Integer> getSquareFunction() {  
 return num -> num \* num;  
 }  
}

1. Using method Reference

Method references are a concise way to refer to a method by its name instead of a lambda expression. The syntax is ClassName::methodName.

public class LearnFunction {  
 public class Main {  
 public static void main(String[] args) {  
 List<String> names = List.*of*("Alice", "Bob", "Charlie");  
 names.forEach(System.*out*::println); // Prints each name  
 }  
 }  
}

## **7. Useful functional Interfaces**

Functional interfaces in Java are interfaces with a single abstract method, also known as Single Abstract Method (SAM) interfaces. They are the foundation of Java's functional programming capabilities, enabling the use of lambda expressions and method references to write concise and readable code.­

Key functional interfaces:-

| **Interface** | **When to Use** | **How to Use** |
| --- | --- | --- |
| Runnable | For executing tasks without a return value | Pass as a lambda to a Thread or executor. |
| Callable | For tasks that return a result | Use with ExecutorService and Future. |
| Comparable | For sorting objects | Override compareTo or use in collections. |
| ActionListener | For handling UI events | Attach as listeners to Swing/AWT components. |
| Consumer | For performing actions on elements | Use with forEach or stream pipelines. |
| Supplier | For generating values on demand | Use for lazy initialization or defaults. |
| Predicate | For filtering or condition checking | Use with filter, allMatch, etc. |
| Function | For data transformation | Use in map operations in streams. |

1. Runnable

Runnable represents a task that can be executed. It does not return a value or throw checked exceptions.­­

//Syntax  
@FunctionalInterface  
public interface Runnable {  
 void run();  
}

Use Case: Used in multithreading and task execution.

public class RunnableExample {  
 public static void main(String[] args) {  
 Runnable task = () -> System.*out*.println("Task is running!");  
 new Thread(task).start(); // Output: Task is running!  
 }  
}

1. Callable

Callable is similar to Runnable but returns a result and can throw exceptions.

//Syntax  
@FunctionalInterface  
public interface Callable<V> {  
 V call() throws Exception;  
}

Use Case: Used in concurrent programming when a result is required.

public class CallableExample {  
 public static void main(String[] args) throws Exception {  
 Callable<String> task = () -> "Result from Callable";  
 ExecutorService executor = Executors.newSingleThreadExecutor();  
 Future<String> result = executor.submit(task);  
 System.*out*.println(result.get()); // Output: Result from Callable  
 executor.shutdown();  
 }  
}

1. Comparable

Defines the natural order of objects.

//Syntax  
public interface Comparable<T> {  
 int compareTo(T o);  
}

Use case : Used in sorting collections.

public class ComparableExample {  
 public static void main(String[] args) {  
 String[] names = {"Alice", "Charlie", "Bob"};  
 Arrays.sort(names); // Uses Comparable  
 System.*out*.println(Arrays.toString(names)); // Output: [Alice, Bob, Charlie]  
 }  
}

1. Consumer

Represents an operation that takes a single input and returns no result.

//Syntax  
@FunctionalInterface  
public interface Consumer<T> {  
 void accept(T t);  
}

Use Case: Used in iteration, logging, or performing actions on each element of a collection.

public class ConsumerExample {  
 public static void main(String[] args) {  
 Consumer<String> consumer = s -> System.*out*.println(s.toUpperCase());  
 consumer.accept("hello, world!"); //HELLO, WORLD!  
  
 }  
}

public class ConsumerExample {  
 public static void main(String[] args) {  
 List<Double> prices = new ArrayList<>(Arrays.*asList*(12.1,42.3,93.9));  
 Consumer<Double> applyDiscount = x->System.*out*.println(x\*0.9); //10.89, 38.07, 84.51  
 prices.forEach(applyDiscount);  
 }  
}

1. Supplier

Supplies a value without any input.

//Syntax  
@FunctionalInterface  
public interface Supplier<T> {  
 T get();  
}

Use case: Used to generate or supply values lazily.

public class SupplierExample {  
 public static void main(String[] args) {  
 Supplier<Double> randomValue = Math::*random*;  
 System.*out*.println(randomValue.get()); // Prints a random number  
 }  
}

1. Function

Represents a function that takes an input and produces a result.

//Syntax  
@FunctionalInterface  
public interface Function<T, R> {  
 R apply(T t);  
 //T: Represents the type of the input argument.  
 //R: Represents the type of the return value.  
}

Use case : used for data transformation

public class FunctionExample {  
 public static void main(String[] args) {  
 Function<String, Integer> lengthFunction = String::length;  
 System.*out*.println(lengthFunction.apply("Hello")); // Output: 5  
 }  
}

1. BiFunction

Represents a function that takes 2 inputs and produces a result.

//Syntax  
@FunctionalInterface  
public interface BiFunction<T, U, R> {  
 R apply(T t, U u);  
 //Takes two input arguments (T and U) and returns a single output (R).  
}

Use Case: used for data transformation

public class LearnFunction {  
 public class Main {  
 public static void main(String[] args) {  
 BiFunction<Integer,Integer,Integer> addNums = (x,y)-> x+y;  
 System.*out*.println(addNums.apply(2,3));  
 }  
 }  
}

## **8. Interview Questions**

**1. Can lambda expressions be used with interfaces that have more than one abstract method? Why or why not?**

No, lambda expressions cannot be used with interfaces that have more than one abstract method because they rely on functional interfaces, which are interfaces with exactly one abstract method. This single abstract method provides the functional target for the lambda expression, enabling Java to determine the intended behavior.

If an interface has multiple abstract methods, there is ambiguity about which method the lambda should implement, violating the principle of functional programming. To ensure compatibility, Java introduced the @FunctionalInterface annotation, which enforces that an interface has only one abstract method, making it suitable for use with lambda expressions.

**1. When should lambdas be avoided?**

* Avoid lambdas for complex logic requiring multiple lines of code.
* Opt for named methods when logic is reused or when lambdas become hard to read.