BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS F213 – Object Oriented Programming

Lab 8: Lambda expressions, Generics

Time:	2 hours
Objective:	To learn about clean coding and type safety
Concepts Covered:	Lambda Expressions, Generics
Prerequisites:	Basic knowledge of java

1. Lambda Expressions

1.1. What are lambda expressions

Lambda expressions in Java are a concise way to represent **anonymous functions**—that is, functions without a name that can be passed around as data. They are primarily used to provide implementations for **functional interfaces** (interfaces with a single abstract method).

- A lambda expression is a block of code that you can pass around to be executed later.
- It can be stored in a variable, passed as an argument, or used to define behavior in methods that expect a functional interface.

```
// Step 1: Define a functional interface
@FunctionalInterface
interface Greeting {
 void sayHello(String name);
}

// Step 2: Use a lambda expression to implement the interface
class LambdaExample {
 public static void main(String[] args) {
     Greeting greet = (name) -> System.out.println("Hello, " + name + "!");
     greet.sayHello("Alice");  // Output: Hello, Alice!
 }
}
```

1.2. When to use them

We use lambda expressions in the following cases:-

- **Using Functional Interfaces**:- When you have an interface with a single abstract method (called a functional interface), like Runnable, Comparator<T>, Consumer<T>, or your own interface.
  List of functional interface:-
  - Runnable: Void return, no parameters
  - Callable: Return value, no parameters
  - Predicate: Boolean return, single parameter
  - Function: Return value, single parameter

- Consumer: Void return, single parameter
- Supplier: Return value, no parameters
- BiFunction: Return value, two parameters
- BiConsumer: Void return, two parameters

Example:- `Runnable r = () -> System.out.println("Task running...");`

1. **Replacing Anonymous Inner Classes:-**
   Traditional:-

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});
```

   Lambda:- `button.addActionListener(e -> System.out.println("Button clicked"));`

2. **Processing Collections (Streams API)**
   Lambda expressions shine when used with Streams and Collections for operations like `map`, `filter`, `forEach`, `reduce, etc.`

   Example:-

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));
```

We **AVOID** using lambda expressions when:-

- You need more than one method in the interface
- The code block is too complex—a regular class is better for readability.
- You need access to more context, like fields or methods of the enclosing class (lambdas can capture variables, but only final or effectively final ones).

1.3. Why to use them

We use them because:-

1. Concise and Readable Code:- lambda expressions allow you to write less code. You can eliminate the need for **anonymous inner classes** when implementing interfaces like Runnable, Comparator, etc.
2. Better Functional Abstraction:- Lambda expressions allow passing **behavior** as a parameter, enabling **functional programming constructs**. This is particularly useful in APIs like the Java Collections Framework and Streams, where behavior like filtering, sorting, or transforming data is passed as an argument
3. Improved API Usability:- Lambdas make the usage of **collections** and **streams** smoother and more expressive. Without lambdas, this would require a full loop or an anonymous class.

4. Better Support for Parallel and Event-Driven Code:- Because you can treat functions as data (pass them around, assign to variables), lambdas enable clearer code for **callbacks**, **concurrent tasks**, and **event handling.**

1.4. How to use them

Different ways to use lambda expressions:-

1. Basic Syntax:-
   (parameter) -> expression
   Example:- `Runnable r = () -> System.out.println("Hello from lambda!");`
   `r.run();`

2. Single Parameter (No Parentheses) :-
   Example:- `Consumer<String> printer = s -> System.out.println(s);`
   `printer.accept("Lambda with single parameter");`

3. Multiple Parameters:-
   Example:- `BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;`
   `System.out.println(add.apply(5, 3));  // Output: 8`

4. Multiple Statements (With Curly Braces and Return):-
   Example:-

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> {
    int result = a * b;
    return result;
};
System.out.println(multiply.apply(4, 5));  // Output: 20
```

5. No Parameters:-
   Example:-
   `Supplier<String> greet = () -> "Hello, Lambda!";`
   `System.out.println(greet.get());`

6. Returning Objects:-
   Example:-
   `Supplier<List<String>> listSupplier = () -> new ArrayList<>();`
   `List<String> list = listSupplier.get();`
   `System.out.println(list);  // Output: []`

7. Lambda with Custom Functional Interface:-
   Example:-
8. Using Lambdas in Collections (like sort()):-
   Example:-

```java
interface MyComparator {
    boolean compare(int a, int b);
}

MyComparator greaterThan = (a, b) -> a > b;
System.out.println(greaterThan.compare(10, 5));
```

```java
List<String> names = Arrays.asList("Zara", "Bob", "Alex");
names.sort((a, b) -> a.compareTo(b));
System.out.println(names);  // Output: [Alex, Bob, Zara]
```

9. Lambda Replacing Anonymous Class:-
   Example:- `ActionListener listener = e -> System.out.println("Button clicked");`

1.5. Exercise

Exercise 1:- Fill in the blanks for the following code

```java
//
class LambdaPractice {
    public static void main(String[] args) {
        // 1. A lambda that prints "Hello World"
        Runnable greet = __________;
        greet.run();

        // 2. A lambda that adds two integers
        BiFunction<Integer, Integer, Integer> add = __________;
        System.out.println("5 + 3 = " + add.apply(5, 3));

        // 3. A lambda that checks if a number is even
        Predicate<Integer> isEven = __________;
        System.out.println("Is 4 even? " + isEven.test(4));

        // 4. A lambda that returns the length of a string
        Function<String, Integer> stringLength = __________;
        System.out.println("Length of 'Lambda': " + stringLength.apply("Lambda"));

        // 5. A lambda with no parameters returning a string
        Supplier<String> getMessage = __________;
        System.out.println(getMessage.get());

        // 6. Sorting a list using a lambda
        List<String> names = Arrays.asList("Charlie", "Alice", "Bob");
        names.sort(__________);
        System.out.println("Sorted names: " + names);
    }
}
```

Exercise 2:- Fill in the blanks

```java
interface StringOperation {
```

```
    String operate(String input);
}
public class StringLambdaTest {
    public static void main(String[] args) {

        // 1. Convert a string to uppercase
        StringOperation toUpperCase = __________;
        System.out.println("Uppercase: " + toUpperCase.operate("hello"));

        // 2. Reverse a string
        StringOperation reverse = __________;
        System.out.println("Reversed: " + reverse.operate("lambda"));

        // 3. Check if a string is a palindrome (return "Yes" or "No")
        StringOperation isPalindrome = __________;
        System.out.println("Is 'radar' a palindrome? " + isPalindrome.operate("radar"));
        System.out.println("Is 'hello' a palindrome? " + isPalindrome.operate("hello"));
    }
}
```

Exercise 3:- Write a Java program using lambda functions to filter and print the names of students who scored above 70 in a mathematics test. (Hint: Use import `java.util.function.Predicate`; to import the `Predicate` interface which can be used to implement the required lambda expressions.

Exercise 4:- Define a functional interface `Comparator` with a method `int compare(int a, int b)`.Write a lambda expression to compare two integers and sort an array of integers using this interface. Print the sorted array. (Hint: Implement the `Comparator` interface, provide a lambda expression for comparing numbers, and use it to sort an array.)

---

2. Generics

Generics in Java are a powerful feature that allow classes, interfaces, and methods to operate on objects of various types while providing compile-time type safety. Instead of using raw types (like Object), generics allow you to specify a placeholder for a type, which gets replaced with a real type when the code is used.

- Generics provide compile-time type checking and eliminate the need for casting.
- They allow the creation of single classes, interfaces, and methods that automatically work with any type of data.

Example:-

Without Generics:

```
ArrayList list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0); // requires casting
```

With Generics:-

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0); // no casting needed
```

Benefits:

- Type Safety: Detects errors at compile time rather than at runtime.
- Code Reusability: Write a single method/class that works with any data type.
- Elimination of Casts: No need for explicit type casting when retrieving elements.

Generics use **type parameters** like <T>, <E>, <K, V>, etc., where:

- T → Type
- E → Element
- K → Key
- V → Value

Bonus type:-

`<?>` – Unknown type

`<? extends T>` – Upper bound :- Accepts T or any of its subclasses.

`<? super T>` – Lower bound:- Accepts T or any of its superclasses.

```
void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

General Example:-

```
public class Box<T> {
 private T value;

 public void set(T value) {
     this.value = value;
 }

 public T get() {
     return value;
 }
}
public class Main {
    public static void main(String[] args) {
        // Box for String
        Box<String> stringBox = new Box<>();
        stringBox.set("Hello Generics");
        System.out.println("String value: " + stringBox.get());

        // Box for Integer
        Box<Integer> intBox = new Box<>();
```

```
        intBox.set(123);
        System.out.println("Integer value: " + intBox.get());
    }
}
```

2.1. Bounds

Bounds in Java generics are rules or constraints you place on a type parameter to limit what types can be used.
There are **two main kinds** of bounds in Java:-

1. Upper Bound:- The generic type must be a specific class or any of its subclasses.

   Example:- `<T extends Animal>`
   This means:- You can use Dog, Cat, or any class that **extends Animal** but cannot use String or Integer

2. Lower Bound:- You can only use this generic with T or any superclass of T.

   Example:- `<? super Integer>`
   This allows Integer, Number and Object but not String or Double

Why Use Bounds?

- To make sure your generic code works with **specific related types only**.
- To allow **safe reading or writing** from/to collections.
- To avoid **runtime type errors**.

Example:-

```
import java.util.*;

public class BoundsDemo {

    // Upper bound: read values safely as Number
    public static void printList(List<? extends Number> list) {
        for (Number num : list) {
            System.out.println("Read: " + num.doubleValue());
        }
    }

    // Lower bound: safely add Integer values
    public static void addIntegers(List<? super Integer> list) {
        list.add(100);
```

```
        list.add(200);
        System.out.println("Added integers to list.");
    }

    public static void main(String[] args) {
        // Example 1: Using upper bound
        List<Double> doubleList = Arrays.asList(1.1, 2.2, 3.3);
        printList(doubleList); // ✅ Allowed: Double extends Number

        // Example 2: Using lower bound
        List<Number> numberList = new ArrayList<>();
        addIntegers(numberList); // ✅ Allowed: Number is a supertype of Integer

        // Print updated list
        printList(numberList); // Using upper bound again to read added integers
    }
}
```

2.2. When to use generics

You should use **generics** in Java when you want to write code that works with **different data types** while still maintaining **type safety** and avoiding **code duplication**.

When to use them:-

1. Collections:- Always use generics with Java collections like `ArrayList`, `HashMap`, `LinkedList`, etc., to ensure type safety.
   Example`:- List<String> names = new ArrayList<>();`

2. Generic Classes:- When you want a class to work with any type (like a custom container, pair, or stack).Example:-

```
public class Box<T> {
 private T value;

 public void set(T value) {
     this.value = value;
 }

 public T get() {
     return value;
 }
}
```

3. Generic Methods:- When the method logic is independent of the type, and you want it to work on various types. Example:-

```
public <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
```

```
}
```

4. Generic Interface:- Example:-

```
interface Container<T> {
    void add(T item);
    T get();
}
```

When Not to Use Generics:

- When the type is **fixed** and known in advance (e.g., a class always processes String).
- With **primitive types** directly (use wrapper classes like Integer, Double instead).
- When working with Java's **reflection** or **serialization**, which can have issues with generic type erasure.

2.3. Exercises

Exercise 1:- Fill in the blanks in the boiler plate code

```
import java.util.*;

//[1] Generic Class with Upper Bound
class Box<__1__> {  // Fill: T extends Number
 private __2__ value; // Fill: T

 public void set(__3__ value) { // Fill: T
     this.value = value;
 }

 public __4__ get() { // Fill: T
     return value;
 }

 public void printDouble() {
     System.out.println("Double value: " + value.doubleValue());
 }
}

//[2] Generic Method
class Printer {
 public static <__5__> void printArray(__6__[] array) { // Fill: T, T
     for (__7__ item : array) { // Fill: T
         System.out.println("Item: " + item);
     }
 }
}

public class GenericsDemo {

 // [3] Upper Bound Wildcard - reading only
 public static void printNumbers(List<? __8__ Number> list) { // Fill: extends
     for (Number num : list) {
```

```
            System.out.println("Read: " + num.doubleValue());
        }
    }
    // [4] Lower Bound Wildcard - safe for writing integers
    public static void addIntegers(List<? __9__ Integer> list) { // Fill: super
        list.add(10);
        list.add(20);
        System.out.println("Added integers to list.");
    }

    public static void main(String[] args) {
        // Using Generic Class
        Box<Integer> intBox = new Box<>(); // Fill: Integer
        intBox.set(42);
        System.out.println("Box contains: " + intBox.get());
        intBox.printDouble();

        Box<Double> doubleBox = new Box<>(); // Fill: Double
        doubleBox.set(3.14);
        doubleBox.printDouble();
    }
}
```

Exercise 2:-

You are building a small library system that manages books, magazines, and other media. You need to write a generic class called `Shelf<T>` that can store any type of item. You will then implement the following:

1. Generic Class: `Shelf<T>`
   - Should store items of type T in a List<T>.
   - Should support methods:
     - `void addItem(T item)`
     - `T getItem(int index)`
     - `void printAllItems()` to print all items in the shelf.
2. Bounded Type:
   - Create a class hierarchy:
     - `abstract class Media`
     - `class Book extends Media`
     - `class Magazine extends Media`
   - Restrict the `Shelf` so that it only stores items that are subclasses of `Media`.
3. Generic Method:-

- In a utility class `LibraryUtils`, write a method:
  `public static <T extends Media> void displayMediaInfo(Shelf<T> shelf)`
  This method should print each media item's class name and content.

4. Wildcard Method:-

- Create a method to copy items from one shelf to another using wildcards
  `public static void copyShelf(Shelf<? extends Media> source, Shelf<? super Media> destination)`

Sample Output:-

```
Shelf contains:
Book: Java Basics
Book: Clean Code

Shelf contains:
Magazine: Science Weekly
Magazine: Tech Today

Displaying media info:
Item type: Book, title: Java Basics
Item type: Book, title: Clean Code

Copied items from Book shelf to Media shelf.
```

Partial skeleton for this:-

```
import java.util.*;

//Step 1: Base Media class
abstract class Media {
 protected String title;

 public Media(String title) {
     this.title = title;
 }

 // TODO: Override toString() in subclasses
}

//Step 2: Subclasses (Book, Magazine)
class Book extends Media {
 public Book(String title) {
     super(title);
 }

 // TODO: Override toString()
}

class Magazine extends Media {
 public Magazine(String title) {
```

```
     super(title);
 }

 // TODO: Override toString()
}

//Step 3: Generic Shelf class
class Shelf</* TODO: add type parameter with bound */> {
 // TODO: Store list of items
 // TODO: Method to add an item
 // TODO: Method to get an item
 // TODO: Method to print all items
}

//Step 4: Utility methods using generics
class LibraryUtils {
 // TODO: Generic method to display info of any Shelf<?>
 // e.g. displayMediaInfo(Shelf<T> shelf)

 // TODO: Wildcard method to copy Shelf<? extends Media> to Shelf<? super Media>
 // e.g. copyShelf(source, destination)
}

//Step 5: Main method to test
public class Main {
 public static void main(String[] args) {
     // TODO: Create Shelf<Book>
     // TODO: Add books
     // TODO: Create Shelf<Magazine>
     // TODO: Add magazines
     // TODO: Print items
     // TODO: Display media info
     // TODO: Copy to a mixed media shelf
 }
}
```