CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE, INDIA

MAI419-3 – JAVA PROGRAMMING

UNIT-3

By

Dr.Thirunavukkarasu, MCA.,M.Phil.,SET.,PhD &
Dr.Sridevi

Unit-3 **Teaching Hours: 15**

GENERICS, LAMBDA, AND THE COLLECTIONS FRAMEWORK

Generics

Generics Concept - General Form of a Generic Class – Bounded Types – Generic Class Hierarchy - Generic Interfaces – Restrictions in Generics.

Lambda Expression

Introduction to Lambda expression- Block Lambda Expressions - Generic Functional Interfaces - Passing lambda expressions as arguments - Lambda expressions and exceptions- Lambda expressions and variable capture.

The Collections Framework

The Collections Overview – Collection Interface – List Interface – Set Interface – SortedSet Interface – Queue Interface - ArrayList Class – LinkedList Class – HashSet Class – Using an Iterator – The For Each Statement. Working with maps – The map interfaces, the map classes. Comparators- the collection algorithms

Lab Exercises:

5. Implement the concept of Generics and lambda expressions
6 . Implement the concept of a collection framework

- **Generics** in Java allow classes, interfaces, and methods to operate on **types specified at runtime**, providing **type safety** and reducing the need for explicit casting.

- *Generics were added by JDK 5. Source code using generics cannot be compiled by earlier versions of **javac**.*

Why Generics?

Before generics, Java used Object type, which caused:
Runtime ClassCastException
Lack of compile-time type checking

Advantages of Generics

Type safety (errors caught at compile time)
Code reusability
Eliminates casting
Improves **readability and** maintainability

Runtime ClassCastException

- **ClassCastException** is a **runtime exception** that occurs when you try to **convert (cast) an object to a class type that it does NOT belong to**.
- It happens **at runtime**, not at compile time.
- Object obj = "Java";
- Integer num = (Integer) obj; // Runtime error
- Compiles successfully but Fails at runtime with ClassCastException

Reason:
obj actually holds a String, not an Integer.

Common Scenario (Inheritance)

```
class Animal
{}
class Dog extends Animal
 {}
Animal a = new Dog();
Dog d = (Dog) a;   // allowed
```

But the following conversion is not allowed

```
Animal a = new Animal();
Dog d = (Dog) a;   //  ClassCastException
```

Reason:

An Animal is not necessarily a Dog

General Form of a Generic Class

```
class Box<T>
{
   T value;
void set(T value)
{       this.value = value;    }
 T get()
{       return value;
 }}
Box<Integer> b = new Box<>();b.set(10);
```

- Here, **T** is the name of a *type parameter.* This name is used as a placeholder for the actual type that will be passed to when an object is created.

- **T** is contained within **< >**. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets.

[Program for Generics class]

Bounded Types

- In Java generics, **bounded types** restrict the range of types that can be passed as type arguments to a generic class, method, or interface.

- Instead of allowing **any data type**, bounded types ensure that the type parameter:

- Belongs to a **specific class**
- Or implements **specific interfaces**

Why Bounded Types are Needed

- To **access methods** of a specific class or interface

- To provide **stronger type safety**

- To enforce **logical constraints** on generic types

Advantages of Bounded Types

- Restricts invalid data types
- Enables use of specific methods
- Improves code reliability
- Reduces runtime errors

- Bounded types limit the type parameter range
- Upper bounds use extends
- Multiple bounds are allowed with rules
- Commonly used with Number, Comparable, and collections

General Syntax

<T extends ClassName>

- T → Type parameter
- extends → Keyword used for both classes and interfaces
- ClassName → Upper bound

Note: The keyword extends is used even when bounding by an **interface**.

Upper Bounded Types

An upper bounded type restricts the type parameter to a specific class or its subclasses.

Example Using Number Class

```
//Integer, Double, Float → subclasses of Number
String → not related to Number
class Test<T extends Number>
{
 T num;
Test(T num)
{       this.num = num;
}
double getSquare()
{ return num.doubleValue() * num.doubleValue(); //convert integer, float into double
  }}

Test<Integer> t1 = new Test<>(10);
Test<Double> t2 = new Test<>(5.5);

Test<String> t3 = new Test<>("Hello"); // Error
```

Bounded Types with Interfaces

- When we use **interfaces as bounds**, we tell Java:
- “Only types that **implement this interface** are allowed.”

```
class ClassName<T extends InterfaceName>
{    // code}
```

T can be **any class that implements InterfaceName**

Why use Bounded Types with Interfaces?

- Ensures **type safety**
- Allows calling **interface methods**
- Avoids runtime errors
- Useful for **generic algorithms**

Single Interface Bound

```
interface Printable {
void print();}
class Report implements Printable {
public void print() {
System.out.println("Printing report");    }}

class Printer<T extends Printable> {
T obj;    Printer(T obj) {
 this.obj = obj;    }
 void execute() {
obj.print(); // Safe    }}
//Works because Report implements Printable
Printer<Report> p = new Printer<>(new Report());p.execute();
Printer<String> p = new Printer<>("Hello"); // Compile-time error
```

Multiple Bounded Types

A generic type can be bounded by:

- One **class**
- Multiple **interfaces**

<T extends ClassName & Interface1 & Interface2>

Rules

- Only **one class** is allowed
- The class must be **first**
- Any number of interfaces can follow

```
class Sample<T extends Number & Comparable<T>>
 {    T value;
 Sample(T value)
{        this.value = value;
}
void show() {
System.out.println("Value: " + value);
  }}
```

Bounded Types in Generic Methods

Generic Method with Bounded Type

```
class Utility {
static <T extends Number> void printValue(T value)
{        System.out.println(value);
 }}
Utility.printValue(100);Utility.printValue(12.5);
```

Wildcard

In Java, a **wildcard** is represented by the **question mark (?)** and is used in **generics** to mean **“an unknown type”**.used in Method parameters
And cannot be extend and more flexible than type parameter.

Without wildcards:

List<Object> list = new ArrayList<String>(); // ❌ Compile-time error

With wildcard:

List<?> list = new ArrayList<String>(); // Valid

3 types of wild card

Unbounded Wildcard, upper bound wild card, lower bound wild card

Unbounded Wildcard

List<?> list;

- Accepts **any type we can read** values as Object
- You **cannot add** elements (except null)

Bounded Wildcards

- Bounded wildcards use ? (**ternary operator / means unknown type)** instead of a type parameter.

- Wildcards improve **polymorphism support**

Upper Bounded Wildcard (? extends)

List<? extends Number>

Accepts a list of:

- Number
- Any subclass of Number

```
void display(List<? extends Number> list)
{
   for (Number n : list)
{       System.out.println(n);    }}

List<Integer> li = new ArrayList<>();
List<Double> ld = new ArrayList<>();
display(li);
display(ld);
```

Lower Bounded Wildcard (? super)

List<? super Integer>
Accepts a list of:

- Integer
- Any superclass of Integer **(number, object)**

```
void addNumbers(List<? super Integer> list)
 {
 list.add(10);
 list.add(20);
}
```

Feature	Type Parameter	Wildcard
Syntax	<T extends Number>	? extends Number
Used in	Class / Method	Method parameters
Can add elements	Yes	No (extends)
Flexibility	Less	More

[Program for all types of wild card]

- extends keyword is used for both **classes and interfaces**
- Bounded types enable **method access** safely
- Multiple bounds must follow **class → interfaces order**
- Wildcards improve **polymorphism support**

Generic Class Hierarchy

- **Generic class hierarchy** refers to **inheritance relationships** between **generic classes** in Java.

It explains how:

- Generic classes interact with inheritance
- Type parameters behave in parent and child classes
- Java handles polymorphism with generics
- In Java, generic class hierarchy follows invariance, and wildcards are used to achieve flexible polymorphic behavior.”

Inheritance and Generics – Basic Concept

```
class A
{
}
class B extends A
{
}
```

But with generics:

```
class A<T>
{
}
class B<T> extends A<T>
{
}
```

Generic types follow invariance, meaning:

A<Integer> ≠ A<Number>

Even though:

Integer extends Number

Generic Class as Superclass / GenericsHierarchy

Generic Subclass with Fixed Type

A subclass can **specify a concrete type** for the parent's generic parameter.

```
class Parent<T>
{    T value;
}
class Child extends Parent<Integer>
{    void display()
 {        System.out.println(value);    }}
```

Parent<T> → generic

Child → non-generic

Type parameter is fixed as Integer

Generic Subclass with Additional Type Parameters

A subclass can:

- Inherit generic types
- Add **new type parameters**

```
class Parent<T>
{    T data;
}
class Child<T, U> extends Parent<T>
{
U extra;
Child(T data, U extra) {
 this.data = data;
this.extra = extra;    }
void show() {        System.out.println(data + " " + extra);    }}
```

Method Overriding in Generic Class Hierarchy

- Type substitution occurs during inheritance

- Overriding follows normal Java rules

```
class Parent<T>
{    T get()
{        return null;
}
}
class Child extends Parent<String>
{    @Override
String get()
 {        return "Hello";
  }
}
```

Advantages of Generic Class Hierarchy

- Reusability
- Compile-time type safety
- Clear inheritance structure
- Cleaner and maintainable code

Generic Interface

Generic interface in Java is an interface that is **parameterized with type variables**.

It allows methods to operate on **different data types** while maintaining **type safety**.

Why Generic Interfaces Are Needed

- Promote **code reusability**
- Provide **compile-time type checking**
- Eliminate explicit type casting
- Support **flexible API design**

General Syntax

```
interface InterfaceName<T>
{    T methodName();
}
```

Where:

- T → Type parameter
- InterfaceName → Generic interface name

Example of a Generic Interface

```
interface Data<T>
{
T getData();
}
```

This interface can work with **any data type**.

Implementing a Generic Interface

A class can implement a generic interface in **two ways**:

1 Class Remains Generic

- The implementing class passes the **same type parameter**.

```
interface Store<T>
{    void set(T value);
 T get();}
class MyStore<T> implements Store<T>
 {    T value;
public void set(T value) {        this.value = value;
 }
public T get()
 {
return value;    }} MyStore<String> s = new MyStore<>();s.set("Java");
```

2 Class Uses a Specific Type

- The implementing class fixes the type parameter.

```
class IntegerStore implements Store<Integer>
 {    Integer value;
public void set(Integer value)
{        this.value = value;    }
 public Integer get() {
  return value;    }}
```

Generic Interface with Multiple Type Parameters

```
interface Pair<K, V>
 {
K getKey();
V getValue();
}

class MyPair<K, V> implements Pair<K, V>
{
 K key;
 V value;
public K getKey()
{        return key;    }
public V getValue() {        return value;    }}
```

Generic Interface with Bounded Types

Generic interfaces can use **bounded type parameters**

```
interface Compare<T extends Comparable<T>>
 {    int compare(T o1, T o2);
}
```

Implementation

```
class CompareNumbers implements Compare<Integer>
 {    public int compare(Integer a, Integer b)
{        return a.compareTo(b);
 }}
```

Generic Interfaces and Inheritance

Interface Extending Another Generic Interface

```
interface A<T>
{    T get();
}
interface B<T> extends A<T>
{    void show();
}
```

Fixing Type During Extension

```
interface C extends A<String>
{
void display();
}
```

Using Generic Interfaces with Polymorphism

Generic interfaces support polymorphism through **reference variables**.

```
interface Printer<T>
 {    void print(T data);
}
class StringPrinter implements Printer<String>
 {    public void print(String data)
{        System.out.println(data);
}}

Printer<String> p = new StringPrinter();
p.print("Hello");
```

Generic Interfaces vs Generic Classes

Feature	Generic Interface	Generic Class
Multiple inheritance	Supported	Not supported
Implementation	implements	extends
Flexibility	More	Less
Object creation	Not allowed	Allowed

Restrictions in Generic Interfaces

Cannot use **primitive types**

Interface<int> i; // ❌

Cannot create static members using type parameter

static T data; // ❌

Type parameters are removed at runtime (**type erasure**)

Advantages of Generic Interfaces

- Strong type safety
- Better abstraction
- Reusable APIs
- Cleaner code

Real-Time Examples

- Comparable<T>
- Comparator<T>
- Iterable<T>
- List<E>
- Map<K, V>

- Generic interfaces allow type-safe method definitions

- Implementing classes may be generic or concrete

- Supports multiple type parameters and bounds

- Widely used in Java Collections Framework

Restrictions in Generics

- Java generics have some limitations due to **type erasure**.

Cannot Create Objects of Type Parameter

T obj = new T(); // Not allowed

- Cannot Create Generic Arrays

T[] arr = new T[10]; // Not allowed

No Static Members Using Type Parameter

static T data; // Not allowed

Cannot Use Primitive Types

Box<int> b; // Not allowed

Box<Integer> b; // Allowed

instanceof Cannot Be Used with Generics

- if(obj instanceof Box<String>) // ❌ Not allowed

Topic	Key Point
Generics	Provide type safety
Generic Class	Uses type parameters
Bounded Types	Restrict type using extends
Class Hierarchy	Wildcards enable flexibility
Generic Interfaces	Interfaces can be generic
Restrictions	Due to type erasure

Lambda Expression

- Lambda expressions were introduced in **Java 8** to support **functional programming** concepts.

- They allow **passing behavior as an argument** and help reduce **boilerplate code**, especially when working with collections and APIs like **Streams**.

- A **lambda expression** is a short block of code that takes in parameters and returns a value.

- Lambdas look similar to methods, but they do not need a name, and they can be written right inside a method body.

A **lambda expression** is an **anonymous function**:

- No name
- No return type declaration
- No access modifier
- Can be treated as an object

- A lambda expression is a concise way to represent an instance of a **functional interface**.

- A **functional interface** is an interface that contains **exactly one abstract method**.

Example:

- Runnable → run()
- Callable → call()
- Comparator → compare()

Syntax

parameter -> *expression*

More than one parameter, wrap them in parentheses

(*parameter1*, *parameter2*) -> *expression*

Syntax Variations

Form	Example
No parameter	() -> System.out.println("Hello")
One parameter	x -> x * 2
Multiple parameters	(a, b) -> a + b
Multiple statements	(a, b) -> { return a + b; }

Lambda Expression is used to

- Reduce **boilerplate code (***repetitive, predictable code)*

- Improve **readability**

- Enable **functional-style programming**

- Work seamlessly with **functional interfaces**

- **Functional interface** has **exactly one abstract method** (e.g., Runnable, Comparator, Callable).

- Helps to write compact, readable code by treating behavior like data—especially when working with collections and functional interfaces.

Lambda Expression vs Method

Feature	Lambda	Method
Name	Anonymous	Named
Reusability	Limited	High
Object	Yes	No
Scope	Local	Class-level

Example (Without Lambda)

```
Runnable r1 = new Runnable()
{    @Override
 public void run() {
System.out.println("Hello world one!");
}}
Thread thread1 = new Thread(r1);
thread1.start();
```

Example (With Lambda)

```
Runnable r2 = () -> System.out.println("Hello world two!");
Thread thread2 = new Thread(r2);
thread2.start();
```

Block Lambda Expressions

When a lambda body contains **multiple statements**, it must be enclosed in **curly braces {}**.

Such lambdas are called **Block Lambda Expressions**.

Syntax

```
(parameters) -> {
statement1;
statement2;
return value;
};
```

```
interface Calculator {
int add(int a, int b);
}
Calculator c = (a, b) -> {
int sum = a + b;
return sum;
};
```

- Curly braces are mandatory
- Return statement is required if method has a return type
- Useful for complex logic
- Program for Block Lambda Expressions

Generic Functional Interfaces

- A **functional interface** is an interface with **exactly one abstract method**.

- Java allows functional interfaces to be **generic**

```
@FunctionalInterface

interface MyGeneric<T> {
T process(T t);
}
```

Using Lambda with Generic Functional Interface

```
MyGeneric<Integer> square = (x) -> x * x;
System.out.println(square.process(5));

MyGeneric<String> upper = (s) -> s.toUpperCase();
System.out.println(upper.process("java"));
```

Benefits

- Type safety
- Code reusability
- Flexibility

[Program Using Lambda with Generic Functional Interface]

Passing Lambda Expressions as Arguments

Lambda expressions can be passed as **method arguments**, enabling powerful abstractions.

```
interface Operation {
int operate(int a, int b);
}

public class Test {
static int calculate(int x, int y, Operation op) {
return op.operate(x, y);
}
public static void main(String[] args) {
int result = calculate(10, 5, (a, b) -> a + b);
System.out.println(result);
}}
```

Advantages

- Clean and modular code
- Eliminates need for multiple classes
- Enhances flexibility

Lambda Expressions and Exceptions

- Lambda expressions can throw **exceptions**, but they must follow the rules of the functional interface.

Example (Unchecked Exception)

```
(a, b) -> a / b; // May throw ArithmeticException
```

Example (Checked Exception)

```
@FunctionalInterface
interface FileReaderLambda {
void read() throws IOException;
}

FileReaderLambda fr = () -> {
throw new IOException("File not found");
};
```

Program **Lambda Expressions and Exceptions**

- Lambda cannot throw broader exceptions than defined in the interface

- Checked exceptions must be declared in the functional interface

Lambda Expressions and Variable Capture

Lambda expressions can **access variables** from their enclosing scope.
This is called **variable capture**.

Types of Variables

- Instance variables
- Static variables
- Local variables (must be **effectively final**)

```
int x = 10;
Runnable r = () -> {
System.out.println(x);
};
r.run();
```

- Invalid Example

```
int x = 10;
Runnable r = () -> System.out.println(x);
x = 20; // Compilation error
```

- program

- Local variables must not change after initialization
- Instance and static variables can be modified
- Improves thread safety

Summary

- Lambda expressions provide a concise way to implement functional interfaces
- Block lambdas handle complex logic
- Generic functional interfaces enhance reusability
- Lambdas can be passed as method arguments
- Exception handling must follow interface rules
- Variable capture requires local variables to be effectively final

Advantages of Lambda Expressions

- Reduces boilerplate code
- Improves code clarity
- Encourages functional programming
- Enhances performance with streams

Limitations of Lambda Expressions

- Can only be used with **functional interfaces**
- Cannot have instance variables
- Cannot change local variables
- Debugging can be harder