Java Practical

EXPERIMENT NO. 1

AIM :

To write a Java program that demonstrates the use of wildcards (unbounded, upper bounded, and lower bounded) in Java Generics

OBJECTIVES :

1. To understand the concept of generics in Java.
2. To learn how wildcards handle unknown data types in generics.
3. To differentiate between **<?>**, **<? extends T>**, and **<? super T>**.
4. To implement wildcard methods using lists of different data types.

THEORY :

Java Generics provide type safety by allowing classes and methods to operate on specific data types while avoiding runtime type errors. However, in many situations the exact type is unknown, and this is where wildcards are useful. Wildcards increase the flexibility of generics by allowing methods to work with different data types in a controlled manner. Java supports three types of wildcards: unbounded (<?>), upper bounded (<? extends T>), and lower bounded (<? super T>). The unbounded wildcard allows any object type and is mainly used for read-only operations. Upper bounded wildcards restrict the type to a specific class or its subclasses and are useful when performing operations like numerical calculations. Lower bounded wildcards allow a class or its super classes and are mainly used for write operations. Wildcards help achieve polymorphism in generics while maintaining compile-time type safety.

There are three types of wildcards:

Wildcard	Meaning	Usage
<?>	Unbounded wildcard	Allows any data type
<? extends T>	Upper bounded wildcard	Accepts T or subclasses of T
<? super T>	Lower bounded wildcard	Accepts T or superclasses of T

Wildcards enable polymorphism in generics by providing type abstraction while maintaining type safety.

IMPLEMENTATION :

```
import java.util.*; class
WildcardDemo {
   // Method using unbounded wildcard
public static void printList(List<?> list)
 {    for (Object elem : list) {
          System.out.print(elem + " ");
       }
       System.out.println();
   }
   // Method using upper bounded wildcard     public static void
sumNumbers(List<? extends Number> list) {       double sum =
0;         for (Number num : list) {             sum +=
num.doubleValue();
       }
       System.out.println("Sum: " + sum);
   }
             // Method using lower bounded wildcard     public
          static void addIntegers(List<? super Integer> list) {
          list.add(10);     list.add(20);
       System.out.println("After adding integers: " + list); }
   public static void main (String[] args) {
       List<Integer> intList = Arrays.asList(1, 2, 3);
       List<Double> doubleList = Arrays.asList(1.5, 2.5, 3.5);
```

```
        List<Object> objList = new ArrayList<>();
System.out.println("Unbounded wildcard:");
printList(intList);         printList(doubleList);
        System.out.println("\nUpper bounded wildcard:");
sumNumbers(intList);

        sumNumbers(doubleList);
        System.out.println("\nLower bounded wildcard:");
addIntegers(objList);
    }
}
```

Output :

```
1 2 3
1.5 2.5 3.5
Sum: 6.0
Sum: 7.5
[10, 20]
```

CONCLUSION :

Wildcards in Java Generics allow functions to operate on different data types while maintaining type safety.
The experiment successfully demonstrated:

- Unbounded wildcard to accept any list.
- Upper bounded wildcard to process lists of numbers.
- Lower bounded wildcard to insert integer values into a generic list.

EXPERIMENT NO. 2

AIM :

To write a Java program that creates a list of items and traverses the list using the List Iterator interface in both forward and backward directions.

OBJECTIVES

1. To understand the use of the List interface in Java.
2. To learn how to use the ListIterator interface to traverse elements.
3. To perform both forward and backward traversal on a list.
4. To explore the hasNext(), next(), hasPrevious(), and previous() methods.

THEORY :

The List interface in Java represents an ordered collection that allows duplicate elements and positional access. To traverse a list, Java provides Iterator and ListIterator interfaces. While Iterator allows only forward traversal, ListIterator supports bidirectional traversal, making it more powerful. Using ListIterator, elements can be accessed in both forward and backward directions using methods like hasNext(), next(), hasPrevious(), and previous(). This makes it useful for applications that require navigation in both directions, such as text editors and playlists. ListIterator also allows modification of elements during traversal. Hence, ListIterator provides greater flexibility and control compared to a normal Iterator.

IMPLEMENTATION :

```
import java.util.*; public class

ListIteratorDemo {     public static void

main(String[] args) {

      // Create a list of items

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

items.add("Apple");

items.add("Banana");

items.add("Cherry");         items.add("Date");

items.add("Elderberry");

      // Create a ListIterator
```

```
        ListIterator<String> iterator = items.listIterator();
System.out.println("Forward Traversal:");          while
(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        System.out.println("\nBackward Traversal:"); while
        (iterator.hasPrevious()) {
            System.out.println(iterator.previous());
        }
    }
}
```

OUTPUT :

CONCLUSION :

The experiment successfully demonstrated the traversal of a list using the ListIterator interface.It showed that unlike the normal Iterator, the ListIterator supports bidirectional traversal, enabling forward and backward movement through the list.

EXPERIMENT NO. 3

AIM :

To write a Java program that creates a Set of strings and prints the elements using the Iterator (forward) and ListIterator (reverse).

OBJECTIVES :

- To understand Set interface and its properties.
- To traverse Set elements using an Iterator.
- To learn how to reverse elements by converting Set to List.

THEORY :

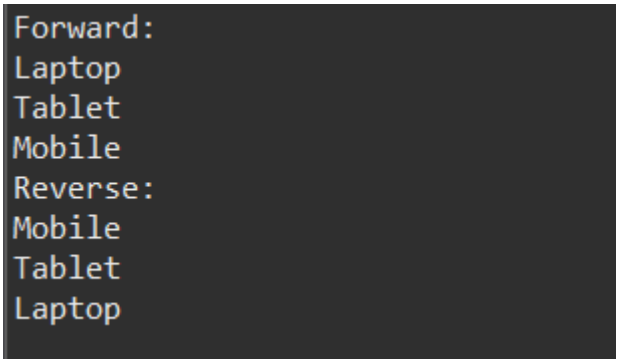
A Set in Java is a collection that does not allow duplicate elements and does not support index-based access. LinkedHashSet is commonly used when insertion order needs to be preserved. Traversing a Set can be done using the Iterator interface, which supports forward traversal only. Since Set does not support backward traversal directly, the Set is converted into a List for reverse traversal. After conversion, a ListIterator is used to traverse elements in the reverse direction. This approach demonstrates how different collection interfaces can be combined to overcome their limitations. It also highlights the flexibility of Java Collections Framework.

IMPLEMENTATION :

```
import java.util.*;  public class
SetIteratorDemo {     public static void
main(String[] args) {
      Set<String> itemSet = new LinkedHashSet<>();
itemSet.add("Laptop");         itemSet.add("Tablet");
itemSet.add("Smartphone");
itemSet.add("Smartwatch");
itemSet.add("Headphones");
      System.out.println("Forward Traversal using Iterator:");
Iterator<String> iterator = itemSet.iterator();         while
(iterator.hasNext()) {
         System.out.println(iterator.next());
      }
```

```
        List<String> itemList = new ArrayList<>(itemSet);
        ListIterator<String> listIterator = itemList.listIterator(itemList.size());
System.out.println("\nReverse Traversal using ListIterator:"); while
(listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

OUTPUT :



CONCLUSION :

The program successfully demonstrated forward traversal of a Set using Iterator and backward traversal by converting the Set into a List and using List Iterator.

EXPERIMENT NO. 4

AIM :

To write a Java program using the Map interface to add, remove and search elements based on keys.

OBJECTIVES :

- To understand the working of the Map interface.
- To perform insertion using put().
- To remove an entry using remove().
- To search items using containsKey() and get().

THEORY :

The Map interface in Java stores data in key–value pairs, where keys are unique and values may be duplicated. Unlike List and Set, Map does not extend the Collection interface. Common operations performed on a Map include insertion, deletion, and searching. The put() method is used to add elements, remove() deletes an entry based on the key, and containsKey() and get() are used to search elements. HashMap is widely used because it provides fast access and does not maintain order. Map is useful in real-world applications like student records, login systems, and product catalogs where fast data retrieval is required.

Operations in Map:

Operation	**Method**
Insert	put(key, value)
Delete	remove(key)
Search	containsKey(key), get(key)

IMPLEMENTATION :

```
import java.util.*; public class

MapOperationsDemo {     public static

void main(String[] args) {

      Map<String, String> itemMap = new HashMap<>();

itemMap.put("101", "Laptop");        itemMap.put("102",

"Smartphone");        itemMap.put("103", "Tablet");

itemMap.put("104", "Smartwatch");
```

```
System.out.println("Initial Map:");         for (Map.Entry<String,
String> entry : itemMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue()) }
        String removedItem = itemMap.remove("103");
        System.out.println("\nRemoved item with key 103: " + removedItem);
String searchKey = "102";         if (itemMap.containsKey(searchKey)) {
            System.out.println("\nItem found: " + itemMap.get(searchKey));
        } else {
            System.out.println("\nItem not found");
        }
        System.out.println("\nFinal Map:");         for
(Map.Entry<String, String> entry : itemMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

OUTPUT :

```
{101=Laptop, 102=Mobile, 103=Tablet}
Found: Laptop
```

CONCLUSION

The experiment successfully implemented adding, deleting, and searching items using the Map interface.

EXPERIMENT NO. 5

AIM:

To write a Java program that uses a lambda expression with two parameters to add numbers.

OBJECTIVES

- To understand functional interfaces.
- To implement lambda expressions with arguments.
- To perform addition using a lambda.

THEORY :

Lambda expressions were introduced in Java 8 to support functional programming. A lambda expression is an anonymous function that can be used to implement functional interfaces. A functional interface is an interface that contains only one abstract method. Lambda expressions reduce boilerplate code and make programs more readable and concise. They are commonly used in event handling, stream processing, and multithreading. The syntax of a lambda expression is simple and consists of parameters followed by an arrow (->) and an expression. Using lambda expressions improves code efficiency and flexibility.

IMPLEMENTATION :

```
@FunctionalInterface interface
AddOperation {
   int add(int a, int b);  }
public class LambdaAddDemo {
   public static void main(String[] args) {
AddOperation addition = (a, b) -> a + b;
int result = addition.add(15, 25);
      System.out.println("Sum of 15 and 25 is: " + result);
   }
}
```

OUTPUT :

CONCLUSION :

The program correctly demonstrated the use of a lambda expression with multiple parameters to add two numbers.

EXPERIMENT NO. 6

AIM :

To design and develop a JSP page for displaying a registration form.

OBJECTIVES :

- To learn how to design a form using JSP.
- To collect user input using HTML form elements.

THEORY :

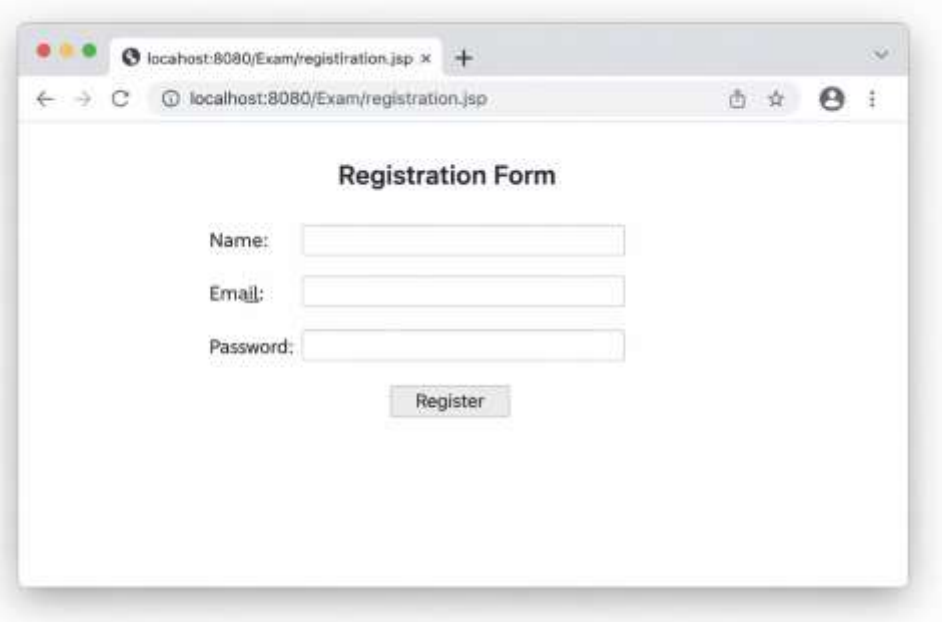
JSP (JavaServer Pages) is a server-side technology used to create dynamic web pages by combining HTML and Java code. JSP simplifies web development by allowing Java code to be embedded within HTML pages. Forms designed using JSP collect user input such as name, email, password, and other details using form elements like text boxes, radio buttons, and dropdowns. The submitted data is sent to another JSP page for processing. JSP is widely used in web applications to handle user interaction and generate dynamic content. It provides better maintainability compared to servlets.

IMPLEMENTATION

```
<html>
<head>
   <title>Registration Form</title>
</head>
<body>
<h2>Student Registration Form</h2>
<form action="submit.jsp" method="post">
   Name: <input type="text" name="name"><br><br>
   Email: <input type="email" name="email"><br><br>
Password: <input type="password" name="password"><br><br>
   Gender:
   <input type="radio" name="gender" value="Male"> Male
   <input type="radio" name="gender" value="Female"> Female <br><br>
   Course:
 <select name="course">
      <option>MCA</option>
```

```
        <option>BCA</option>
        <option>BSc IT</option>
    </select><br><br>
    <input type="submit" value="Register">
</form>
</body>
</html>
```

OUTPUT :



CONCLUSION :

The JSP program successfully displayed a registration form to collect user details.

EXPERIMENT NO. 7

AIM :

To develop a JSP application to insert, delete and display records from the StudentMaster (RollNo, Name, Semester, Course) database table.

OBJECTIVES :

- To perform database CRUD operations using JSP.
- To connect JSP with MySQL database using JDBC.
- To insert, delete and display student records.

THEORY :

JSP can interact with databases using JDBC to perform CRUD operations such as Insert, Delete, and Display. JDBC provides classes like DriverManager, Connection, PreparedStatement, and ResultSet to connect and interact with databases. In this experiment, student records are stored in a MySQL database table. PreparedStatement is used to prevent SQL injection and improve performance. JSP pages dynamically execute SQL queries and display results on the web page. This experiment demonstrates real-time database interaction in web applications.DriverManager.getConnection()

PreparedStatement

ResultSet

StudentMaster Table Schema:

Column	Data Type
RollNo	INT
Name	VARCHAR
Semester	INT
Course	VARCHAR

IMPLEMENTATION :

Database Connection File (db.jsp)

```
<%@ page import="java.sql.*" %>
<%
Connection con = null; try
{
```

```
   Class.forName("com.mysql.cj.jdbc.Driver");     con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/college", "root", ""); }
catch(Exception e) {     out.println(e);
}
%>
```

Insert Record (insert.jsp)

```
<%@ include file="db.jsp" %>
<%
String r = request.getParameter("roll");
String n = request.getParameter("name");
String s = request.getParameter("sem");
String c = request.getParameter("course");
PreparedStatement ps = con.prepareStatement("insert into StudentMaster values(?,?,?,?)");
ps.setInt(1, Integer.parseInt(r)); ps.setString(2, n); ps.setInt(3, Integer.parseInt(s));
ps.setString(4, c); ps.executeUpdate(); out.println("Record Inserted Successfully");
%>
```

Delete Record (delete.jsp)

```
<%@ include file="db.jsp" %>
<%
String r = request.getParameter("roll");
PreparedStatement ps = con.prepareStatement("delete from StudentMaster where
RollNo=?"); ps.setInt(1, Integer.parseInt(r)); int x = ps.executeUpdate();
out.println(x > 0 ? "Record Deleted Successfully" : "Record Not Found");
%>
```

Display Records (display.jsp)

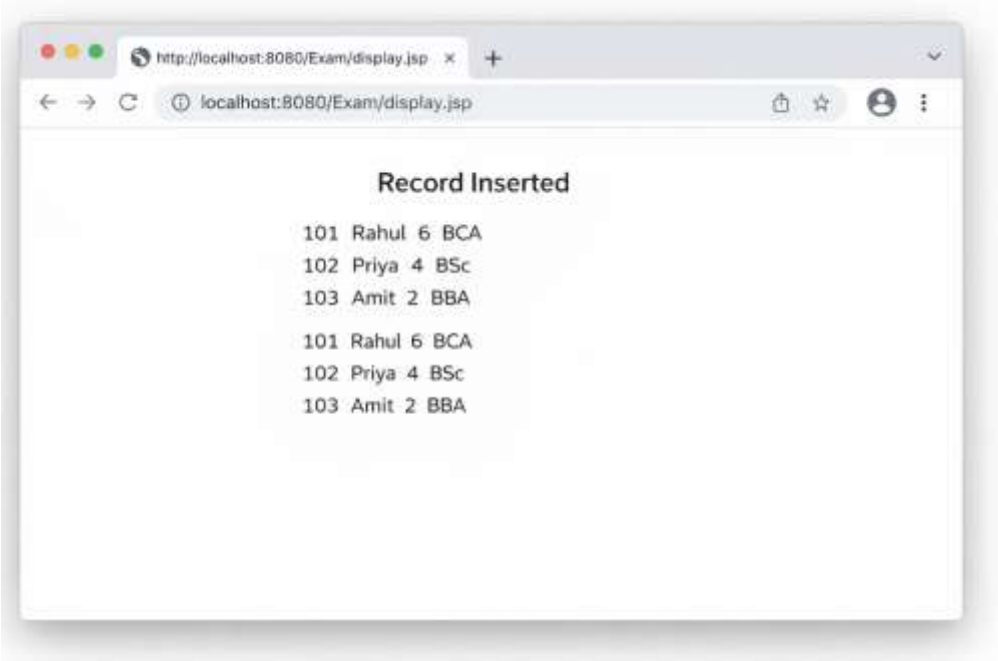
```
<%@ include file="db.jsp" %>
<%
PreparedStatement ps = con.prepareStatement("select * from StudentMaster");
ResultSet rs = ps.executeQuery(); while(rs.next()){
```

```
    out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getInt(3) + " " + rs.getString(4) +
"<br>");

}

%>
```

OUTPUT :



CONCLUSION :

The JSP application successfully performed Insert, Delete and Display operations on the StudentMaster table using JDBC.

EXPERIMENT NO. 8

AIM :

To create a Spring application that prints Hello World using dependency injection.

OBJECTIVES :

- To understand Inversion of Control container.
- To configure Spring beans and run the program.

THEORY :

Spring Framework is a powerful Java framework that supports enterprise application development. It uses the concept of Inversion of Control (IoC), where object creation and dependency management are handled by the container. Beans are configured in an XML file and loaded using ApplicationContext. Dependency Injection improves modularity, testability, and maintainability of applications. This experiment demonstrates a basic Spring application that prints a message using configured beans. It shows how Spring reduces tight coupling between classes.

IMPLEMENTATION :

hello.java public class Hello {

public void printMessage() {

System.out.println("Hello World from Spring Framework");

}

}

bean.xml

<beans>

<bean id="helloBean" class="Hello"/>

</beans> Main.java import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main { public static void

main(String[] args) {

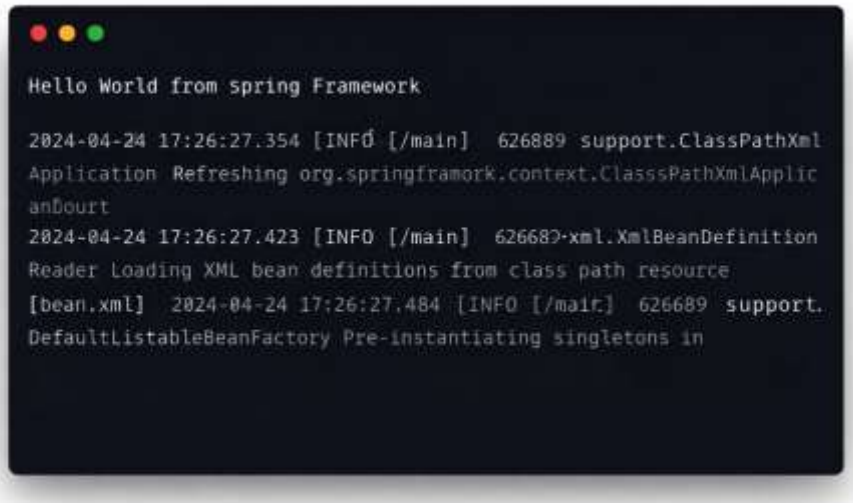
ApplicationContext ctx = new ClassPathXmlApplicationContext("bean.xml");

Hello h = (Hello) ctx.getBean("helloBean");

h.printMessage();

```
    }
}
```

OUTPUT :



CONCLUSION :

The experiment demonstrated printing a message using the Spring Framework with dependency injection.

EXPERIMENT NO. 9

AIM :

To implement and demonstrate **Autowiring** in Spring Framework.

OBJECTIVES :

- To show automatic bean dependency injection.
- To use autowire="byName" or byType.

THEORY :

Autowiring is a feature of Spring that automatically injects dependent objects into a bean. It reduces the need for explicit configuration and simplifies development. Spring supports autowiring by name, by type, and by constructor. In this experiment, autowiring is done using the byType mode, where Spring automatically matches the dependency based on class type. Autowiring improves code readability and reduces configuration errors. It is widely used in large applications where manual wiring becomes complex.

IMPLEMENTATION :

Address.java public class Address {

String city = "Mumbai"; public String

toString() { return city; }

}

Student.java public class Student { Address address; public

void setAddress(Address address) { this.address = address; } public

void display() { System.out.println("Address: " + address); }

}

bean.xml

<beans>

<bean id="address" class="Address"/>

<bean id="student" class="Student" autowire="byType"/>

</beans> **Main.java** import

org.springframework.context.*; import

org.springframework.context.support.*;

```
public class Main {     public static void
main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("bean.xml");
Student s = (Student) ctx.getBean("student");
        s.display();
    }
}
```

OUTPUT :

CONCLUSION :

Spring Autowiring was successfully demonstrated using automatic injection of dependent objects.

EXPERIMENT NO. 10

AIM :

To implement Aspect-Oriented Programming (AOP) in Spring using before advice.

OBJECTIVES :

- **To understand AOP concepts.**
- **To execute additional logic before method execution.**

THEORY :

Aspect-Oriented Programming (AOP) is used to separate cross-cutting concerns like logging, security, and transactions from business logic. Spring AOP allows developers to apply additional behavior to methods without modifying their code. Before Advice is a type of advice that executes before the target method is invoked. It is commonly used for logging, validation, and authentication. This experiment demonstrates how Spring AOP improves modularity and clean separation of concerns. AOP makes applications more maintainable and scalable.

IMPLEMENTATION

Service.java public class

Service { public void

display() {

System.out.println("Executing business logic...");

}

}

BeforeAdvice.java import

org.springframework.aop.MethodBeforeAdvice; import

java.lang.reflect.Method; public class BeforeAdviceImpl implements

MethodBeforeAdvice {

@Override public void before(Method m, Object[] args,

Object target) {

System.out.println("Before Advice executed.");

}

}

bean.xml

<beans>

```
    <bean id="service" class="Service"/>
    <bean id="beforeAdvice" class="BeforeAdviceImpl"/>
    <bean class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" value="service"/>
        <property name="interceptorNames" value="beforeAdvice"/>
    </bean>
</beans>  Main.java import
org.springframework.context.*; import
org.springframework.context.support.*; public
class Main {     public static void main(String[]
args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("bean.xml");
Service s = (Service) ctx.getBean("service");
        s.display();
    }
```

OUTPUT :

CONCLUSION :

The experiment successfully demonstrated Spring AOP Before Advice by executing additional functionality before a method call.