Name: Vaishali Ramesh Kale

Email ID: kalevaishalir16@gmail.com

Day 19:

Task 1: Generics and Type Safety Create a generic Pair class that holds two objects of different types, and write a method to return a reversed version of the pair.

Solution:::

Explanation

1. Generic Type Parameters:

- The Pair class uses two type parameters, T and U, to allow for different types of objects to be stored in the pair.

2. Constructor:

- The constructor initializes the pair with the provided values for first and second.

3. Getters and Setters:

- getFirst() and getSecond() methods return the first and second elements, respectively.
- setFirst(T first) and setSecond(U second) methods allow setting the first and second elements, respectively.

4. Reverse Method:

- The reverse method creates a new Pair object with the first and second elements swapped.

5. toString Method:

- The toString method is overridden to provide a string representation of the Pair object.

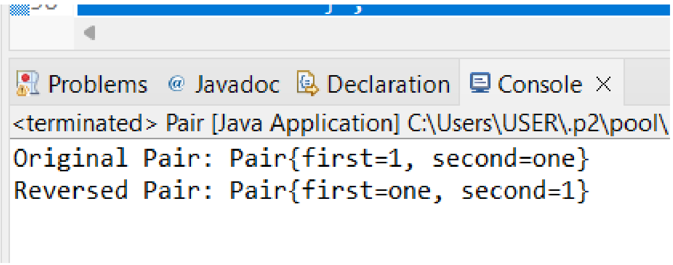
6. Main Method:

- In the main method, a Pair object is created and its reversed version is printed to demonstrate the functionality.

CODE::::

```
package com.wipro.java;

public class Pair<T, U> {
    private T first;
    private U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public void setFirst(T first) {
        this.first = first;
    }

    public U getSecond() {
        return second;
    }

    public void setSecond(U second) {
        this.second = second;
    }

    // Method to return a reversed version of the pair
    public Pair<U, T> reverse() {
        return new Pair<>(second, first);
    }

    @Override
    public String toString() {
        return "Pair{" +
                "first=" + first +
                ", second=" + second +
                '}';
    }

    public static void main(String[] args) {
        Pair<Integer, String> pair = new Pair<>(1, "one");
        System.out.println("Original Pair: " + pair);

        Pair<String, Integer> reversedPair = pair.reverse();
        System.out.println("Reversed Pair: " + reversedPair);
    }
}
```

OUTPUT::::



```
<terminated> Pair [Java Application] C:\Users\USER\.p2\pool\
Original Pair: Pair{first=1, second=one}
Reversed Pair: Pair{first=one, second=1}
```

Task 2: Generic Classes and Methods Implement a generic method that swaps the positions of two elements in an array, regardless of their type, and demonstrate its usage with different object types.

Solution:::

Explanation

1. Generic Swap Method:

- The swap method uses a type parameter <T> to allow it to swap elements of any type.
- It takes an array of type T and two indices i and j as parameters.
- The method checks if the provided indices are within the bounds of the array. If not, it throws an IndexOutOfBoundsException.
- It then swaps the elements at the specified positions using a temporary variable.

2. Main Method:

- The main method demonstrates the usage of the swap method with different types of arrays: Integer, String, and Double.

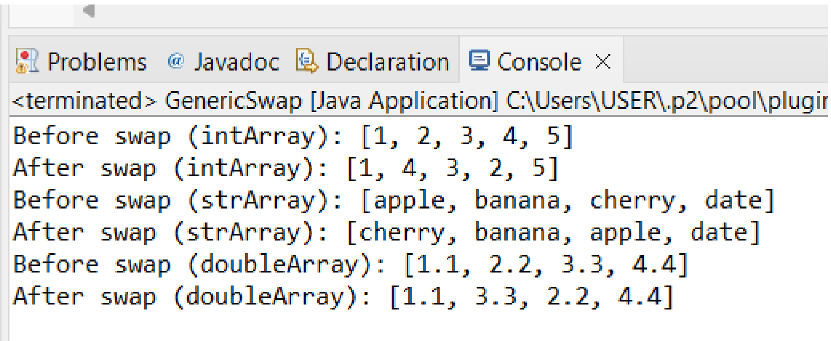
3. Print Statements:

- The state of each array is printed before and after the swap to show that the method works correctly.

CODE::::

```
package com.wipro.java;

public class GenericSwap {
  // Generic method to swap two elements in an array
  public static <T> void swap(T[] array, int i, int j) {
    if (i < 0 || i >= array.length || j < 0 || j >= array.length) {
      throw new IndexOutOfBoundsException("Index out of bounds");
    }
    T temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

  public static void main(String[] args) {
    // Example with Integer array
    Integer[] intArray = {1, 2, 3, 4, 5};
    System.out.println("Before swap (intArray): " + java.util.Arrays.toString(intArray));
    swap(intArray, 1, 3);
    System.out.println("After swap (intArray): " + java.util.Arrays.toString(intArray));

    // Example with String array
    String[] strArray = {"apple", "banana", "cherry", "date"};
    System.out.println("Before swap (strArray): " + java.util.Arrays.toString(strArray));
    swap(strArray, 0, 2);
    System.out.println("After swap (strArray): " + java.util.Arrays.toString(strArray));

    // Example with Double array
    Double[] doubleArray = {1.1, 2.2, 3.3, 4.4};
    System.out.println("Before swap (doubleArray): " +
java.util.Arrays.toString(doubleArray));
    swap(doubleArray, 1, 2);
    System.out.println("After swap (doubleArray): " +
java.util.Arrays.toString(doubleArray));
  }
}
```

OUTPUT::::



<terminated> GenericSwap [Java Application] C:\Users\USER\.p2\pool\plugir

```
Before swap (intArray): [1, 2, 3, 4, 5]
After swap (intArray): [1, 4, 3, 2, 5]
Before swap (strArray): [apple, banana, cherry, date]
After swap (strArray): [cherry, banana, apple, date]
Before swap (doubleArray): [1.1, 2.2, 3.3, 4.4]
After swap (doubleArray): [1.1, 3.3, 2.2, 4.4]
```

Task 3: Reflection API Use reflection to inspect a class's methods, fields, and constructors, and modify the access level of a private field, setting its value during runtime .

Solution:::

Explanation

1. Sample Class:

- SampleClass has a private field, a public method, a private method, and a constructor.

2. Reflection Example:

- The main method of the ReflectionExample class performs the following tasks:
- Load the Class: Uses Class.forName to get the Class object associated with SampleClass.
- Inspect Methods: Retrieves and prints all declared methods of the class.
- Inspect Fields: Retrieves and prints all declared fields of the class.
- Inspect Constructors: Retrieves and prints all declared constructors of the class.
- Create Instance: Uses the default constructor to create an instance of SampleClass.
- Modify Access Level: Uses setAccessible(true) to bypass Java's access control checks on the private field.
- Set Field Value: Changes the value of the private field using reflection.
- Verify Change: Invokes the public getPrivateField method to confirm the private field's value has been modified.

CODE::::

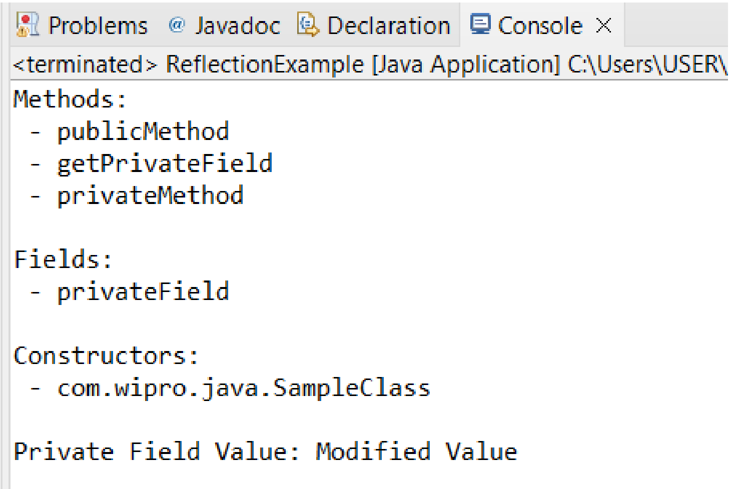
```
package com.wipro.java;

public class SampleClass {
   private String privateField;

   public SampleClass() {
      this.privateField = "Initial Value";
   }

   public void publicMethod() {
      System.out.println("Public Method");
   }

   private void privateMethod() {
      System.out.println("Private Method");
```

```
    }

    public String getPrivateField() {
        return privateField;
    }
}
```

```
package com.wipro.java;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectionExample {
    public static void main(String[] args) {
        try {
            // Get the Class object associated with SampleClass
            Class<?> clazz = Class.forName("com.wipro.java.SampleClass");

            // Inspect the class's methods
            Method[] methods = clazz.getDeclaredMethods();
            System.out.println("Methods:");
            for (Method method : methods) {
                System.out.println(" - " + method.getName());
            }

            // Inspect the class's fields
            Field[] fields = clazz.getDeclaredFields();
            System.out.println("\nFields:");
            for (Field field : fields) {
                System.out.println(" - " + field.getName());
            }

            // Inspect the class's constructors
            Constructor<?>[] constructors = clazz.getDeclaredConstructors();
            System.out.println("\nConstructors:");
            for (Constructor<?> constructor : constructors) {
                System.out.println(" - " + constructor.getName());
            }

            // Create an instance of the class using the default constructor
            Constructor<?> constructor = clazz.getDeclaredConstructor();
            constructor.setAccessible(true);
            Object instance = constructor.newInstance();

            // Modify the access level of the private field
```

```
      Field privateField = clazz.getDeclaredField("privateField");
      privateField.setAccessible(true);

      // Set the value of the private field
      privateField.set(instance, "Modified Value");

      // Verify that the value has been changed
      Method getPrivateFieldMethod = clazz.getMethod("getPrivateField");
      String fieldValue = (String) getPrivateFieldMethod.invoke(instance);
      System.out.println("\nPrivate Field Value: " + fieldValue);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

OUTPUT::::



```
<terminated> ReflectionExample [Java Application] C:\Users\USER\
Methods:
 - publicMethod
 - getPrivateField
 - privateMethod

Fields:
 - privateField

Constructors:
 - com.wipro.java.SampleClass

Private Field Value: Modified Value
```

Task 4: Lambda Expressions Implement a Comparator for a Person class using a lambda expression, and sort a list of Person objects by their age.

Solution:::

Explanation

1. Person Class:

- The Person class has two fields: name and age, with corresponding getters and a constructor. The toString method is overridden to provide a string representation of a Person object.

2. Creating a List of Person Objects:

- In the main method of PersonSortingExample, a list of Person objects is created and populated with sample data.

3. Lambda Expression for Comparator:

- A lambda expression is used to create a Comparator<Person> that compares two Person objects by their age. The expression (Person p1, Person p2) -> Integer.compare(p1.getAge(), p2.getAge()) compares the ages of two Person objects.

4. Sorting the List:

- The people.sort(byAge) method sorts the list of Person objects using the provided Comparator.

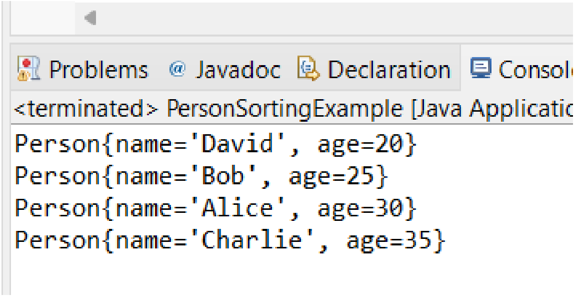
5. Printing the Sorted List:

- The forEach method is used to print each Person object in the sorted list.

CODE::::

```
package com.wipro.java;


import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;


// Define the Person class
class Person {
    private String name;
    private int age;


    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}

// Main class to demonstrate sorting using lambda expression
public class PersonSortingExample {
    public static void main(String[] args) {
        // Step 1: Create a list of Person objects
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));
        people.add(new Person("David", 20));

        // Step 2: Use a lambda expression to implement the Comparator
        Comparator<Person> byAge = (Person p1, Person p2) -> Integer.compare(p1.getAge(), p2.getAge());

        // Step 3: Sort the list using the Comparator
        people.sort(byAge);
        // Step 4: Print the sorted list

        people.forEach(person -> System.out.println(person));

    }

}
```

OUTPUT::::



```
<terminated> PersonSortingExample [Java Applicatic
Person{name='David', age=20}
Person{name='Bob', age=25}
Person{name='Alice', age=30}
Person{name='Charlie', age=35}
```

Task 5: Functional Interfaces Create a method that accepts functions as parameters using Predicate, Function, Consumer, and Supplier interfaces to operate on a Person object.

Solution:::

Explanation

1. Predicate:

- A Predicate is used to test a condition on a Person object. The isAdult predicate checks if the person is an adult.

2. Function:

- A Function is used to apply a function to a Person object and return a result. The nameLength function returns the length of the person's name.

3. Consumer:

- A Consumer is used to perform an action on a Person object. The printDetails consumer prints the details of the person.

4. Supplier:

- A Supplier is used to supply a new Person object. The newPersonSupplier creates a new person named "Alice" with age 25.

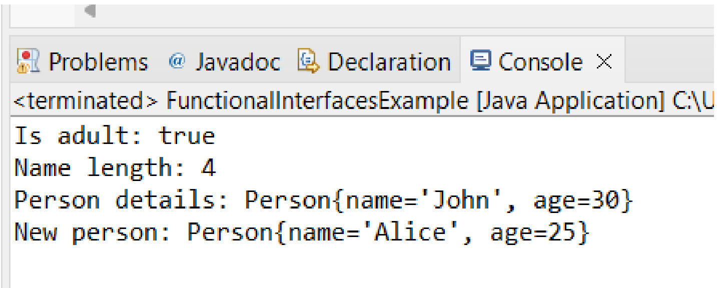
5. Helper Methods:

- testPredicate: Accepts a Predicate and a Person object, and returns the result of the predicate test.
- applyFunction: Accepts a Function and a Person object, and returns the result of applying the function.
- acceptConsumer: Accepts a Consumer and a Person object, and performs the consumer action.
- getSupplier: Accepts a Supplier and returns the supplied Person object.

CODE::::

```
package com.wipro.java;
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;

public class FunctionalInterfacesExample {
    public static void main(String[] args) {
        // Create a sample Person object
        Person person = new Person("John", 30);

        // Predicate: Check if person is an adult
        Predicate<Person> isAdult = p -> p.getAge() >= 18;
        System.out.println("Is adult: " + testPredicate(isAdult, person));

        // Function: Get person's name length
        Function<Person, Integer> nameLength = p -> p.getName().length();
        System.out.println("Name length: " + applyFunction(nameLength, person));

        // Consumer: Print person's details
        Consumer<Person> printDetails = p -> System.out.println("Person details: "
+ p);
        acceptConsumer(printDetails, person);

        // Supplier: Create a new person
        Supplier<Person> newPersonSupplier = () -> new Person("Alice", 25);
        Person newPerson = getSupplier(newPersonSupplier);
        System.out.println("New person: " + newPerson);
    }

    // Method to test Predicate
    public static boolean testPredicate(Predicate<Person> predicate, Person
person) {
        return predicate.test(person);
    }

    // Method to apply Function
    public static <R> R applyFunction(Function<Person, R> function, Person person)
{
        return function.apply(person);
    }
    // Method to accept Consumer
    public static void acceptConsumer(Consumer<Person> consumer, Person person) {
        consumer.accept(person);
    }

    // Method to get Supplier
    public static Person getSupplier(Supplier<Person> supplier) {
        return supplier.get();
    }
}
```

OUTPUT::::



<terminated> FunctionalInterfacesExample [Java Application] C:\U

```
Is adult: true
Name length: 4
Person details: Person{name='John', age=30}
New person: Person{name='Alice', age=25}
```