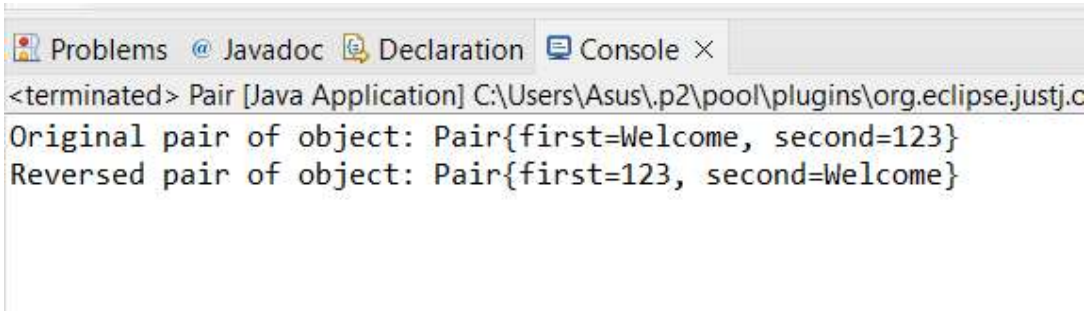
Day19

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.

```
package practice;
public class Pair<F, S> {
  private F first;
  private S second;
  public Pair(F first, S second) {
     this.first = first;
     this.second = second;
  }
  public F getFirst() {
     return first;
  }
  public void setFirst(F first) {
     this.first = first;
  }
  public S getSecond() {
     return second;
  }
  public void setSecond(S second) {
     this.second = second;
  }

  public Pair<S, F> reverse() {
     return new Pair<>(second, first);
  }
  @Override
  public String toString() {
     return "Pair{" +
           "first=" + first +
           ", second=" + second +
           '}';
  }
   public static void main(String[] args) {
     Pair<String, Integer> originalPair = new Pair<>("welcome", 123);
     System.out.println("Original pair: " + originalPair);
     Pair<Integer, String> reversedPair = originalPair.reverse();
     System.out.println("Reversed pair: " + reversedPair);
  }
}
```

Output:



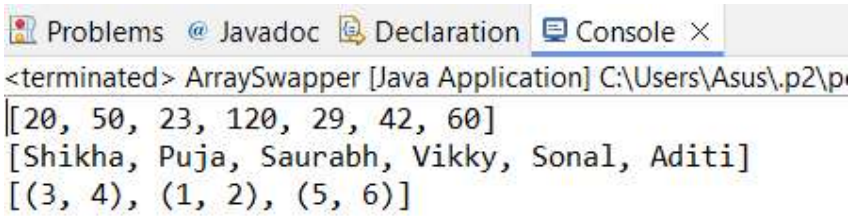
Task 2: Generic Classes and Methods

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.

```
package practice;
public class ArraySwapper {
        public static <T> void swap(T[] array, int index1, int index2) {

          if (0 <= index1 && index1 < array.length && 0 <= index2 && index2 < array.length) {
            T temp = array[index1];
            array[index1] = array[index2];
            array[index2] = temp;
          } else {
            throw new IndexOutOfBoundsException("Indices out of range");
          }
        }
        public static void main(String[] args) {

  // Demonstration with integers
          Integer[] myArray = {20,50,60,120,29, 42,23};
          swap(myArray, 2, 6);
          System.out.println(java.util.Arrays.toString(myArray));

  // Demonstration with strings
          String[] ListOfName = {"Sonal", "Puja", "Saurabh", "Vikky", "Shikha", "Aditi"};
          swap(listOfName, 0, 4);
          System.out.println(java.util.Arrays.toString(listOfName));

  // Demonstration with custom objects (assuming they have a value attribute)
          class Point {
            int x;
            int y;
            Point(int x, int y) {
              this.x = x;
              this.y = y;
            }
            @Override
            public String toString() {
              return "(" + x + ", " + y + ")";
```

```
        }
      }
      Point[] points = {new Point(1, 2), new Point(3, 4), new Point(5, 6)};
      swap(points, 0, 1);
      System.out.println(java.util.Arrays.toString(points));
    }
  }
```

Output:



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.

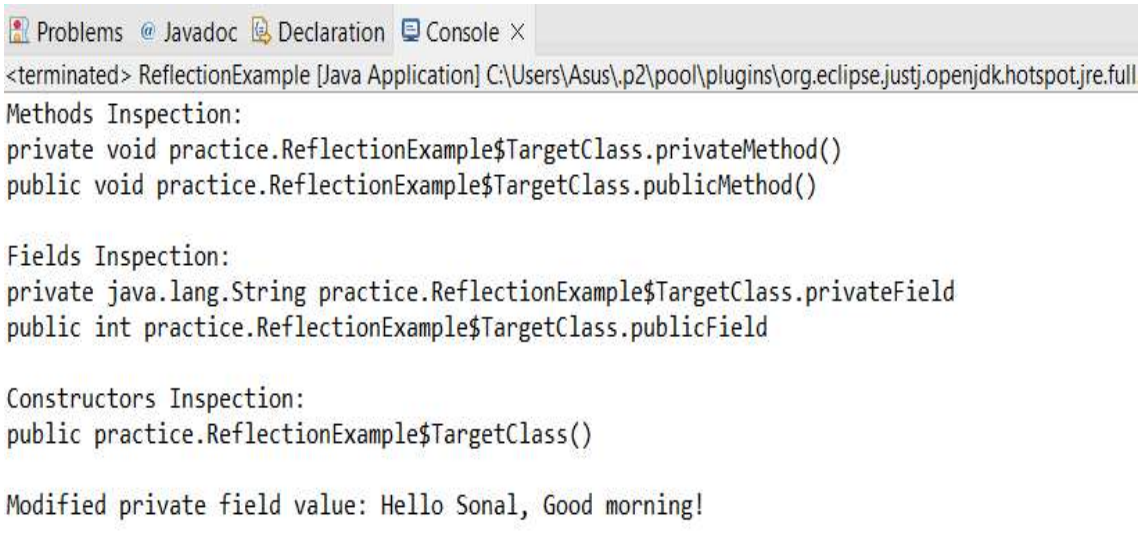
```
package practice;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectionExample {
  public static void main(String[] args) {
    try {
      // Obtain the Class object for the TargetClass
      Class<?> obj = TargetClass.class;

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

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

      // Inspect the class's constructors
      Constructor<?>[] constructors = obj.getDeclaredConstructors();
      System.out.println("\nConstructors Inspection:");
      for (Constructor<?> constructor : constructors) {
        System.out.println(constructor);
```

```
        }

        // Modify the access level of a private field and set its value
        Object instance = obj.getDeclaredConstructor().newInstance();
        Field privateField = obj.getDeclaredField("privateField Inspection");
        privateField.setAccessible(true); // Make the private field accessible

        // Set the value of the private field
        privateField.set(instance, "Hello Sonal, Good morning!");

        // Verify the value of the private field
        System.out.println("\nModified private field value: " + privateField.get(instance));
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
```

Class that is used to inspect:

```
  // inner class
  static class TargetClass {
      private String privateField = "Hello Sonal";
      public int publicField;
      public TargetClass() {
      }
      public void publicMethod() {
          System.out.println("Public method called");
      }
      private void privateMethod() {
          System.out.println("Private method called");
      }
  }
}
```

Output:



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..

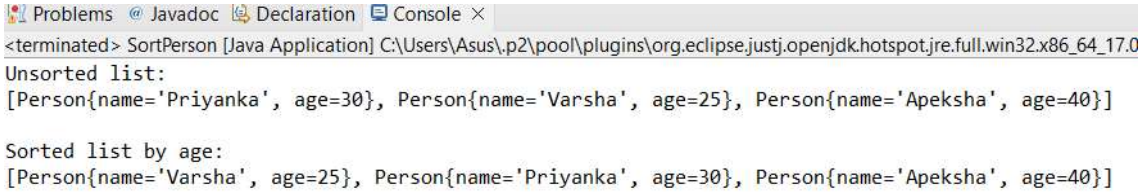
```
    package practice;
import java.util.Comparator;
public class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
     this.name = name;
     this.age = age;
  }
  public String getName() {
     return name;
  }
  public int getAge() {
     return age;
  }
   public void setName(String name) {
     this.name = name;
  }
  public void setAge(int age) {
     this.age = age;
  }

  @Override
  public String toString() {
     return "Person{name='" + name + "', age=" + age + "}";
  }
  public static final Comparator<Person> BY_AGE = (p1, p2) -> p1.getAge() - p2.getAge();
}
```

Main class

```
package practice;
        import java.util.ArrayList;
        import java.util.List;

        public class SortPerson {

           public static void main(String[] args) {

              List<Person> people = new ArrayList<>();

              people.add(new Person("Priyanka", 30));
              people.add(new Person("Varsha", 25));
              people.add(new Person("Apeksha", 40));
```

```
            System.out.println("Unsorted list:");
            System.out.println(people);  // Prints unsorted list

            // Sort the list using the Comparator
            people.sort(Person.BY_AGE);
            System.out.println("\nSorted list by age:");
            System.out.println(people);  // Prints sorted list by age
        }
    }
```

Output:



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.

```
package practice;
import java.util.Comparator;
public 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;
  }
  public void setName(String name) {
     this.name = name;
  }
  public void setAge(int age) {
     this.age = age;
  }

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

Main class:

```java
package practice;
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;
public class Main{
  public static void main(String[] args) {
     Person person = new Person("Priyanka", 30);

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

     Function<Person, String> getNameUpperCase = p -> p.getName().toUpperCase();
     System.out.println("Name in uppercase: " + operateOnPerson(person, getNameUpperCase));

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

     // Supplier example: Create a new Person object
     Supplier<Person> createNewPerson = () -> new Person("Varsha", 25);
     System.out.println("New person: " + operateOnPerson(createNewPerson));
  }

  public static boolean operateOnPerson(Person person, Predicate<Person> predicate) {
     return predicate.test(person);
  }

  public static <R> R operateOnPerson(Person person, Function<Person, R> function) {
     return function.apply(person);
  }

  public static void operateOnPerson(Person person, Consumer<Person> consumer) {
     consumer.accept(person);
  }

  public static Person operateOnPerson(Supplier<Person> supplier) {
     return supplier.get();
  }
}
```

Output: