Assignment – 20

Name - Md. Firoze Baba
firozebaba68@gmail.com

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.

Program:

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

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

	public T getFirst() {
		return first;
	}

	public U getSecond() {
		return second;
	}

	public GenericsAndTypeSafety<U, T> reverse() {
		return new GenericsAndTypeSafety<>(second, first);
	}

	public static void main(String[] args) {
		GenericsAndTypeSafety<String, Integer> pair = new
GenericsAndTypeSafety<>("Hello", 123);
		System.out.println("Original Pair: " + pair.getFirst() +
", " + pair.getSecond());

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

Output:

```
Original Pair: Hello, 123
Reversed Pair: 123, Hello
```

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.

Program:

```
import java.util.Arrays;

public class GenericClasses_Methods {

	public static <T> void swap(T[] arr,int x,int y) {
		try {
			if(x < 0 || x>=arr.length || y<0 || y>=arr.length)
{
				System.out.println("Invalid indices
					provided.");
			}
		} catch (IllegalArgumentException e) {
			e.printStackTrace();
		}
		T temp=arr[x];
		arr[x]=arr[y];
		arr[y]=temp;
	}

	public static void main(String[] args) {
		Integer[] iarr= {4,3,6,2,1,5};
		System.out.println("Original Array:
				"+Arrays.toString(iarr));
		swap(iarr, 4, 5);
		System.out.println("Array after swapping: "
				+Arrays.toString(iarr));
		String[] sarr=

			{"Firoze","Bob","pavan","manohar","Prakash"};
		System.out.println("Original Array: "
				+Arrays.toString(sarr));
		swap(sarr, 0, 3);
		System.out.println("Array after swapping: "
				+Arrays.toString(sarr));

		Character[] carr= {'a','m','m','u','b'};
		System.out.println("Original Array: "
				+Arrays.toString(carr));
		swap(carr, 1, 4);
		System.out.println("Array after swapping: "
				+Arrays.toString(carr));
	}
}
```

Output:

```
Original Array: [4, 3, 6, 2, 1, 5]
Array after swapping: [4, 3, 6, 2, 5, 1]
Original Array: [Firoze, Bob, pavan, manohar, Prakash]
Array after swapping: [manohar, Bob, pavan, Firoze, Prakash]
Original Array: [a, m, m, u, b]
Array after swapping: [a, b, m, u, m]
```

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

Program:

```
import java.lang.reflect.Field;

public class ReflectionAPI {

	private String privatename;

	public ReflectionAPI(String privatename) {
		this.privatename = privatename;
	}

	public String getPrivatename() {
		return privatename;
	}

	public void setPrivatename(String privatename) {
		this.privatename = privatename;
	}

	public static void main(String[] args) {

		ReflectionAPI rapi = new ReflectionAPI("Firoze");

		try {
			Field f = ReflectionAPI.class
					.getDeclaredField("privatename");
			f.setAccessible(true);
			System.out.println("Name before modification: "
					+ f.get(rapi));
			f.set(rapi, "Firoze Baba");
			System.out.println("Name after modification: "
					+ f.get(rapi));
		} catch (SecurityException | IllegalArgumentException |
				IllegalAccessException e) {
			e.printStackTrace();
		} catch (NoSuchFieldException e) {
			e.printStackTrace();
		}
	}
}
```

Output:

```
Name before modification: Firoze
Name after modification: Firoze Baba
```

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..

Program:

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

public class LamdaExpressions {
	private String name;
	private int age;

	public LamdaExpressions(String name, int age) {
		this.name = name;
		this.age = age;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public static void main(String[] args) {
		List<LamdaExpressions> list = new ArrayList<>();
		list.add(new LamdaExpressions("Firoze", 24));
		list.add(new LamdaExpressions("Prakash", 14));
		list.add(new LamdaExpressions("Bob", 18));
		list.add(new LamdaExpressions("manohar", 23));
		Comparator<LamdaExpressions> compare =
			Comparator.comparingInt(LamdaExpressions::getAge);
		list.sort(compare);
		System.out.println("Data sorted by age: ");
		for (LamdaExpressions x : list) {
			System.out.println(x.getName()+" : " +
x.getAge());
		}
	}
}
```

Output:

```
Data sorted by age:
Prakash: 14
Bob: 18
manohar : 23
Firoze: 24
```

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.

Program:

```
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;

public class FunctionalInterface {

	public static void process(Person p,Predicate<Person>
				predicate, Consumer<Person> consumer,
				Function<Person, String> function,
				Supplier<Person> supplier)
	{
		if (predicate.test(p)) {
			consumer.accept(p);
			System.out.println("Transfored name: "
					+ function.apply(p));
		} else {
			System.out.println("Predicate condition not met
					of person: " + p);
			System.out.println("New person created: "
					+ supplier.get());
		}
	}

	public static void main(String[] args) {
		Person person = new Person("Firoze", 24);

		Predicate<Person> ap = p -> p.getAge() > 26;

		Consumer<Person> consumer = p ->
				System.out.println("Person details: " + p);

		Function<Person, String> function = p ->
					p.getName().toUpperCase();

		Supplier<Person> supplier = () -> new Person("Teja",
23);

		process(person, ap, consumer, function, supplier);
	}
}
```

Output:

```
Predicate condition not met of person: Person [name=Firoze, | age=24]
New person created: Person [name=Bob, | age=23]
```