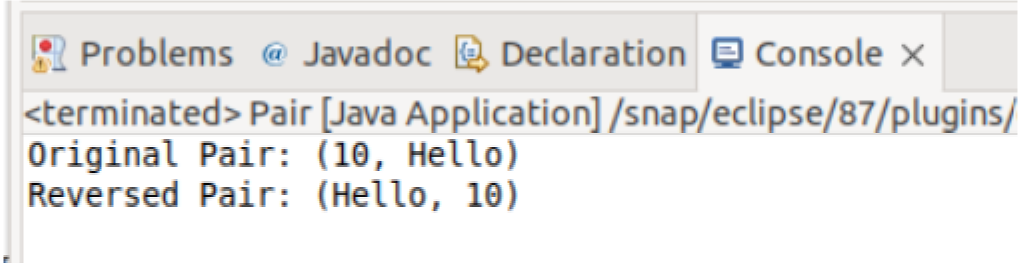
Day 6:

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.

```
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 U getSecond() {
        return second;
    }

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

    public static void main(String[] args) {
        Pair<Integer, String> pair = new Pair<>(10, "Hello");
        System.out.println("Original Pair: (" + pair.getFirst() + ", " +
pair.getSecond() + ")");
```

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

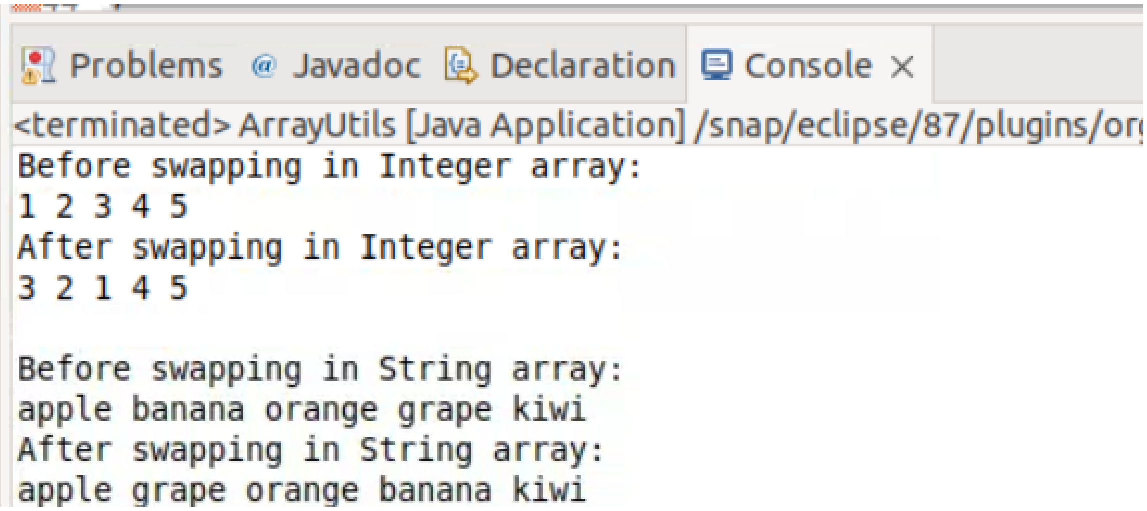


```
<terminated> Pair [Java Application] /snap/eclipse/87/plugins/
Original Pair: (10, Hello)
Reversed Pair: (Hello, 10)
```

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.

```
public class ArrayUtils {

    public static <T> void swap(T[] array, int index1, int index2) {
        if (index1 < 0 || index1 >= array.length || index2 < 0 || index2 >=
array.length) {
            throw new IllegalArgumentException("Invalid indices");
        }

        T temp = array[index1];
        array[index1] = array[index2];
        array[index2] = temp;
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4, 5};
        String[] strArray = {"apple", "banana", "orange", "grape", "kiwi"};

        // Swapping elements in Integer array
        System.out.println("Before swapping in Integer array:");
        for (Integer num : intArray) {
            System.out.print(num + " ");
        }
        System.out.println();
        swap(intArray, 0, 2);
        System.out.println("After swapping in Integer array:");
        for (Integer num : intArray) {
            System.out.print(num + " ");
        }
        System.out.println("\n");
        // Swapping elements in String array
        System.out.println("Before swapping in String array:");
        for (String fruit : strArray) {
            System.out.print(fruit + " ");
        }
        System.out.println();
        swap(strArray, 1, 3);
        System.out.println("After swapping in String array:");
        for (String fruit : strArray) {
            System.out.print(fruit + " ");
        }
    }
}
```



```
<terminated> ArrayUtils [Java Application] /snap/eclipse/87/plugins/or
Before swapping in Integer array:
1 2 3 4 5
After swapping in Integer array:
3 2 1 4 5

Before swapping in String array:
apple banana orange grape kiwi
After swapping in String array:
apple grape orange banana kiwi
```

This ArrayUtils class contains a generic swap method that takes an array and the indices of two elements to be swapped. It performs the swapping operation regardless of the type of elements in the array. In the main method, examples of swapping elements in arrays of Integer and String types are demonstrated.

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

example to demonstrate using reflection to inspect a class's methods, fields, and constructors, and then modify the access level of a private field and set its value during runtime.

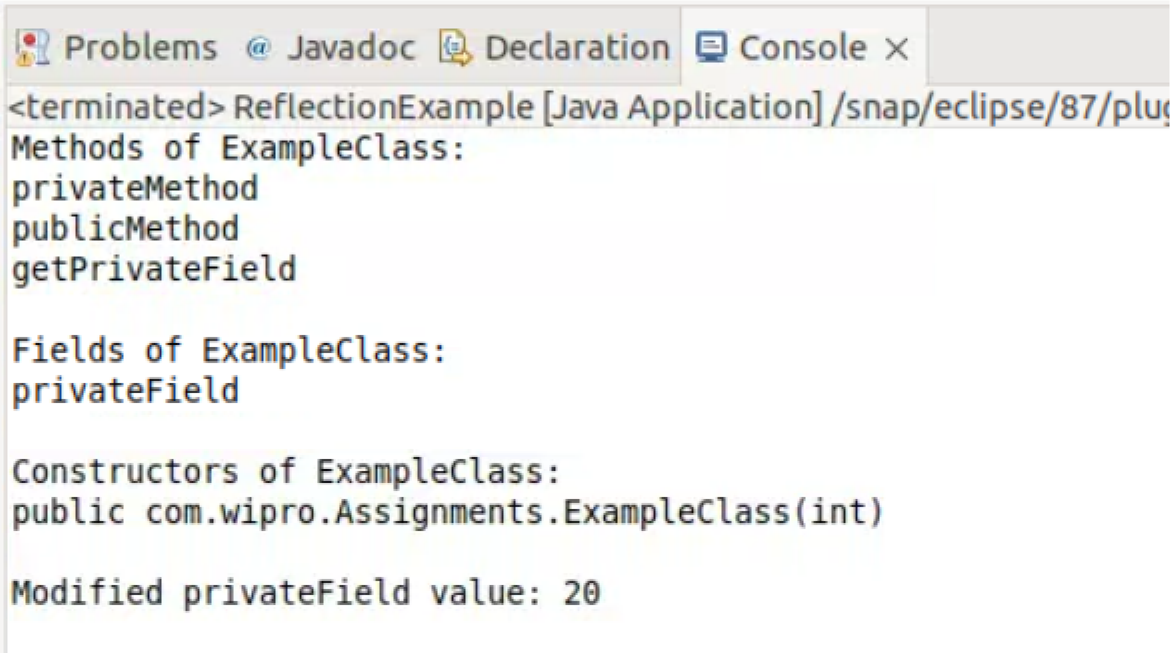
Let's say we have a class ExampleClass with a private field privateField:

```
public class ExampleClass {
    private int privateField;

    public ExampleClass(int privateField) {
        this.privateField = privateField;
    }

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

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

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

Now, we will use reflection to inspect and modify this class:

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

public class ReflectionExample {

    public static void main(String[] args) throws Exception {
        // Inspecting methods
        Method[] methods = ExampleClass.class.getDeclaredMethods();
        System.out.println("Methods of ExampleClass:");
        for (Method method : methods) {
            System.out.println(method.getName());
        }

        // Inspecting fields
        Field[] fields = ExampleClass.class.getDeclaredFields();
        System.out.println("\nFields of ExampleClass:");
        for (Field field : fields) {
            System.out.println(field.getName());
        }

        // Inspecting constructors
        Constructor[] constructors =
ExampleClass.class.getDeclaredConstructors();
        System.out.println("\nConstructors of ExampleClass:");
        for (Constructor constructor : constructors) {
            System.out.println(constructor);
        }

        // Modifying private field and setting its value during runtime
        ExampleClass instance = new ExampleClass(10);
        Field privateField =
ExampleClass.class.getDeclaredField("privateField");
```

```
        privateField.setAccessible(true); // Set accessibility of private field to
true
        privateField.setInt(instance, 20); // Set the value of private field

        // Checking the modified value
        System.out.println("\nModified privateField value: " +
instance.getPrivateField());
    }
}
```

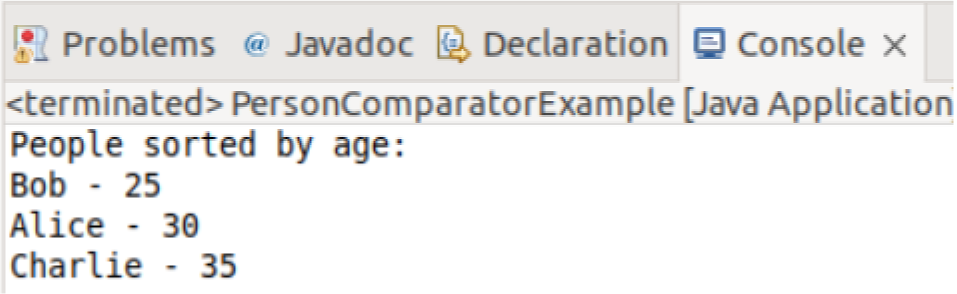


```
<terminated> ReflectionExample [Java Application] /snap/eclipse/87/plu
Methods of ExampleClass:
privateMethod
publicMethod
getPrivateField

Fields of ExampleClass:
privateField

Constructors of ExampleClass:
public com.wipro.Assignments.ExampleClass(int)

Modified privateField value: 20
```

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.

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

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 class PersonComparatorExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        // Comparator using lambda expression to sort by age
        Comparator<Person> ageComparator = (p1, p2) -> p1.getAge() -
p2.getAge();

        // Sorting the list of people by age
        people.sort(ageComparator);

        // Printing sorted list
        System.out.println("People sorted by age:");
        for (Person person : people) {
            System.out.println(person.getName() + " - " + person.getAge());
        }
    }
}
```



```
<terminated> PersonComparatorExample [Java Application
People sorted by age:
Bob - 25
Alice - 30
Charlie - 35
```

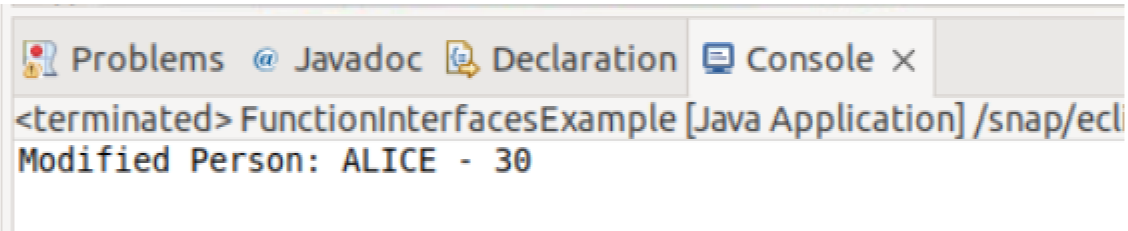
In this example, we define a Comparator<Person> using a lambda expression (p1, p2) -> p1.getAge() - p2.getAge() to compare Person objects based on their age. Then, we use the sort method of the List interface to sort the list of Person objects using this comparator. Finally, we print the sorted list of people by their age.

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.

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

class PersonDemo {
    private String name;
    private int age;

    public PersonDemo(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;
    }
```

```
}

public class FunctionInterfacesExample {

    // Method accepting functions as parameters
    public static void operateOnPerson(PersonDemo person,
                                       Predicate<PersonDemo> predicate,
                                       Function<PersonDemo, PersonDemo> function,
                                       Consumer<PersonDemo> consumer,
                                       Supplier<PersonDemo> supplier) {
        // Check condition using Predicate
        if (predicate.test(person)) {
            // Apply transformation using Function
            PersonDemo modifiedPerson = function.apply(person);
            // Perform some action using Consumer
            consumer.accept(modifiedPerson);
        } else {
            // If condition not met, create a new Person using Supplier
            PersonDemo newPerson = supplier.get();
            // Perform some action using Consumer
            consumer.accept(newPerson);
        }
    }

    public static void main(String[] args) {
        PersonDemo person = new PersonDemo("Alice", 30);

        // Example usage of operateOnPerson method
        operateOnPerson(person,
                p -> p.getAge() >= 18, // Predicate to check if age is greater
than or equal to 18
                p -> {                  // Function to transform person's name to
uppercase
                    p.setName(p.getName().toUpperCase());
                return p;
            },
            p -> System.out.println("Modified Person: " + p.getName() + " -
" + p.getAge()), // Consumer to print modified person
            () -> new PersonDemo("Unknown", 0) // Supplier to create a
new Person
        );
    }
}
```



```
<terminated> FunctionInterfacesExample [Java Application] /snap/ecli
Modified Person: ALICE - 30
```

In this example, the operateOnPerson method accepts a Person object and four functional interfaces - Predicate, Function, Consumer, and Supplier. Inside the method, it evaluates a condition using the Predicate, transforms the Person object using the Function, and performs some action on the PersonDemo object using the Consumer. Additionally, if the condition fails, it creates a new Person object using the Supplier. Finally, in the main method, we demonstrate the usage of the operateOnPerson method.