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.

```
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<String, Integer> pair = new Pair<>("Hello", 123);
        System.out.println("Original Pair: " + pair.getFirst() + ", " + pair.getSecond());

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

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 swapElements(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};
        System.out.println("Original Integer Array: " + java.util.Arrays.toString(intArray));
        swapElements(intArray, 1, 3);
        System.out.println("Array after swapping: " + java.util.Arrays.toString(intArray));
        String[] strArray = {"apple", "banana", "orange", "grape"};
        System.out.println("\nOriginal String Array: " + java.util.Arrays.toString(strArray));
        swapElements(strArray, 0, 2);
        System.out.println("Array after swapping: " + java.util.Arrays.toString(strArray));
        Character[] charArray = {'a', 'b', 'c', 'd', 'e'};
        System.out.println("\nOriginal Character Array: " + java.util.Arrays.toString(charArray));
        swapElements(charArray, 2, 4);
        System.out.println("Array after swapping: " + java.util.Arrays.toString(charArray));
    }
}
```

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

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

class MyClass {
    private int privateField;

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

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

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

```
}

public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        Class<?> myClass = MyClass.class;
        System.out.println("Methods:");
        Method[] methods = myClass.getDeclaredMethods();
        for (Method method : methods) {
            System.out.println(method.getName());
        }
        System.out.println("\nFields:");
        Field[] fields = myClass.getDeclaredFields();
        for (Field field : fields) {
            System.out.println(field.getName());
        }
        System.out.println("\nConstructors:");
        Constructor<?>[] constructors = myClass.getDeclaredConstructors();
        for (Constructor<?> constructor : constructors) {
            System.out.println(constructor);
        }

        System.out.println("\nModifying private field:");
        MyClass obj = new MyClass(10);
        Field privateField = myClass.getDeclaredField("privateField");
        privateField.setAccessible(true); // Set accessible to true to access private field
        int value = (int) privateField.get(obj); // Get the value of private field
        System.out.println("Original value of privateField: " + value);
        privateField.set(obj, 20); // Set new value to private field
        System.out.println("Modified value of privateField: " +
obj.getClass().getDeclaredField("privateField").get(obj));
```

```
    }
}
```

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;
    }
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + 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));
        people.sort(Comparator.comparingInt(Person::getAge));
        System.out.println("Sorted list of Person objects by age:");
        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

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 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 +
                '}';
    }
}

public class FunctionInterfaceExample {
    public static void processPerson(Person person,
                                     Predicate<Person> predicate,
                                     Function<Person, String> function,
                                     Consumer<String> consumer,
                                     Supplier<String> supplier) {
        if (predicate.test(person)) {
            String result = function.apply(person);
            consumer.accept(result);
        } else {
            String defaultResult = supplier.get();
            consumer.accept(defaultResult);
        }
    }

    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        Predicate<Person> isAdult = p -> p.getAge() >= 18;
        Function<Person, String> greetingFunction = p -> "Hello, " + p.getName();
        Consumer<String> printConsumer = System.out::println;
        Supplier<String> defaultGreetingSupplier = () -> "Sorry, you are not an adult.";

        System.out.println("Processing adult person:");
        processPerson(person, isAdult, greetingFunction, printConsumer,
defaultGreetingSupplier);
        person = new Person("Bob", 15);
        System.out.println("\nProcessing underage person:");
        processPerson(person, isAdult, greetingFunction, printConsumer,
defaultGreetingSupplier);
    }
}
```