

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 void setFirst(T first) {  
        this.first = first;  
    }  
  
    public U getSecond() {  
        return second;  
    }  
  
    public void setSecond(U second) {  
        this.second = second;  
    }  
}
```

```
}
```

```
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<String, Integer> pair = new Pair<>("hello", 123);  
    System.out.println(pair);  
    Pair<Integer, String> reversedPair = pair.reverse();  
    System.out.println(reversedPair);  
}  
}
```

How it works

Pair Class:

- Holds two generic types T and U.
- Constructor initializes the first and second fields.
- Getters and setters provide access to first and second.

Reverse Method:

- Creates a new Pair with the second and first fields swapped.
- Returns this new Pair with reversed types.

Main Method:

- Demonstrates creating a Pair of String and Integer.
- Prints the original pair.
- Reverses the pair and prints the reversed pair.

output:

Pair{first=hello, second=123}

Pair{first=123, second=hello}

Original Pair:

first: "hello" (type String)

second: 123 (type Integer)

Reversed Pair:

first: 123 (type Integer)

second: "hello" (type String)

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.

Swap Method:

```
public class ArrayUtils {  
    public static <T> void swap(T[] array, int index1, int index2) {  
        T temp = array[index1];  
        array[index1] = array[index2];  
        array[index2] = temp;  
    }  
}
```

Generic Swap Method:

- `public static <T> void swap(T[] array, int index1, int index2):`
- T is a type parameter, making the method generic.
- The method swaps the elements at index1 and index2 in the array.
- The swap is done using a temporary variable temp.

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        String[] stringArray = {"apple", "banana", "cherry", "date"};
```

```
        System.out.println("Before swap: " + java.util.Arrays.toString(stringArray));
```

```
        ArrayUtils.swap(stringArray, 1, 3);
```

```
        System.out.println("After swap: " + java.util.Arrays.toString(stringArray));
```

```
        Integer[] intArray = {10, 20, 30, 40};
```

```
        System.out.println("Before swap: " + java.util.Arrays.toString(intArray));
```

```
        ArrayUtils.swap(intArray, 0, 2);
```

```
        System.out.println("After swap: " + java.util.Arrays.toString(intArray));
```

```
        Double[] doubleArray = {1.5, 2.5, 3.5, 4.5};
```

```
        System.out.println("Before swap: " + java.util.Arrays.toString(doubleArray));
```

```
        ArrayUtils.swap(doubleArray, 2, 3);
```

```
        System.out.println("After swap: " + java.util.Arrays.toString(doubleArray));
```

```
        Character[] charArray = {'A', 'B', 'C', 'D'};
```

```
        System.out.println("Before swap: " + java.util.Arrays.toString(charArray));
```

```
        ArrayUtils.swap(charArray, 1, 2);
```

```
        System.out.println("After swap: " + java.util.Arrays.toString(charArray));
```

```
    }
```

```
}
```

Output:

Before swap: [apple, banana, cherry, date]

After swap: [apple, date, cherry, banana]

Before swap: [10, 20, 30, 40]

After swap: [30, 20, 10, 40]

Before swap: [1.5, 2.5, 3.5, 4.5]

After swap: [1.5, 2.5, 4.5, 3.5]

Before swap: [A, B, C, D]

After swap: [A, C, B, D]

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

Person Class:

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person() {  
        this.name = "Unknown";  
        this.age = 0;  
    }  
  
    public Person(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;  
}
```

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

```
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 {  
    // Load the Person class  
    Class<?> personClass = Class.forName("Person");  
  
    // Inspect constructors  
    Constructor<?>[] constructors = personClass.getDeclaredConstructors();  
    System.out.println("Constructors:");  
    for (Constructor<?> constructor : constructors) {  
        System.out.println(" " + constructor);  
    }  
  
    // Inspect fields  
    Field[] fields = personClass.getDeclaredFields();  
    System.out.println("\nFields:");  
    for (Field field : fields) {  
        System.out.println(" " + field);  
    }  
  
    // Inspect methods  
    Method[] methods = personClass.getDeclaredMethods();  
    System.out.println("\nMethods:");  
    for (Method method : methods) {  
        System.out.println(" " + method);  
    }  
  
    // Create an instance of Person using the default constructor  
    Object personInstance = personClass.getDeclaredConstructor().newInstance();  
  
    // Access and modify the private 'name' field
```

```

Field nameField = personClass.getDeclaredField("name");
nameField.setAccessible(true); // Make the private field accessible
nameField.set(personInstance, "John Doe");

// Access and modify the private 'age' field
Field ageField = personClass.getDeclaredField("age");
ageField.setAccessible(true);
ageField.set(personInstance, 30);

System.out.println("\nModified Person instance:");
System.out.println(personInstance);

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

```

Output:

After modifying the private fields, the Person instance is Person{name='John Doe', age=30}.

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..

Person Class:

```

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 void setName(String name) {  
    this.name = name;  
}
```

```
public int getAge() {  
    return age;  
}
```

```
public void setAge(int age) {  
    this.age = age;  
}
```

@Override

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

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

```

import java.util.List;

public class Main {

    public static void main(String[] args) {

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

        persons.add(new Person("Alice", 30));
        persons.add(new Person("Bob", 25));
        persons.add(new Person("Charlie", 35));
        persons.add(new Person("Diana", 20));

        // Sort persons by age using Comparator with lambda expression
        persons.sort(Comparator.comparingInt(Person::getAge));

        //persons.sort((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));

        System.out.println("Sorted Persons by Age:");
        persons.forEach(System.out::println);
    }
}

```

Explanation

Sorting with Comparator Using Lambda Expression:

- **Comparator.comparingInt(Person::getAge)** creates a Comparator that compares Person objects by their age.
- This lambda expression is equivalent to **Comparator<Person> comparator = (p1, p2) -> Integer.compare(p1.getAge(), p2.getAge());**.
- It compares the ages of two Person objects p1 and p2 by invoking their getAge() methods and comparing the results.

Lambda Expression:

- **Comparator.comparingInt(Person::getAge)** is a lambda expression that specifies how to compare two Person objects based on their ages.
- It is a shorthand for writing a Comparator implementation by directly referencing the **getAge()** method of the Person class.
- The sorted persons list is printed using **persons.forEach(System.out::println)**, which prints each Person object in the sorted list.

Output:

Sorted Persons by Age:

Person{name='Diana', 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.

Person Class:

```
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 + "'}";
}
}
```

Person Class:

- Represents a Person with name and age attributes.
- Provides getters and setters for name and age.
- Overrides toString() method for better string representation.

```
import java.util.function.Consumer;
```

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

public class Main {

    // Method that accepts functions to operate on a Person object
    public static void operateOnPerson(
        Supplier<Person> personSupplier,
        Consumer<Person> personConsumer,
        Predicate<Person> personPredicate,
        Function<Person, String> personNameFunction,
        Function<Person, Integer> personAgeFunction) {

        // Get a new Person object from the supplier
        Person person = personSupplier.get();

        // Print the person
        System.out.println("Original Person: " + person);

        // Check if the person meets the predicate condition
        if (personPredicate.test(person)) {
            // Apply the function to get the person's name and age
            String name = personNameFunction.apply(person);
            int age = personAgeFunction.apply(person);
            System.out.println("Person's Name: " + name);
            System.out.println("Person's Age: " + age);

            // Modify the person using the consumer
```

```

        personConsumer.accept(person);

        System.out.println("Modified Person: " + person);
    } else {
        System.out.println("Predicate condition not met.");
    }
}

```

```

public static void main(String[] args) {
    // Create a new Person using a Supplier
    Supplier<Person> personSupplier = () -> new Person("Ram", 45);

    // Define a Consumer to change the name and age of a Person
    Consumer<Person> personNameAndAgeConsumer = person -> {
        person.setName("Sita");
        person.setAge(30);
    };

    // Define a Predicate to check if a Person is older than 25
    Predicate<Person> personAgePredicate = person -> person.getAge() > 25;

    // Define a Function to get the name of a Person
    Function<Person, String> personNameFunction = Person::getName;

    // Define a Function to get the age of a Person
    Function<Person, Integer> personAgeFunction = Person::getAge;

    // Use the operateOnPerson method with the defined functions
    operateOnPerson(
        personSupplier,

```

```
        personNameAndAgeConsumer,  
        personAgePredicate,  
        personNameFunction,  
        personAgeFunction  
    );  
}  
}
```

Functional Interfaces Usage:

- Supplier<Person> (personSupplier) provides a new Person object.
- Consumer<Person> (personNameConsumer) changes the name of a Person.
- Predicate<Person> (personAgePredicate) checks if a Person is older than 25.
- Function<Person, String> (personNameFunction) retrieves the name of a Person.

operateOnPerson Method:

- Accepts the functional interfaces as parameters.
- Gets a new Person object from the Supplier.
- Prints the original Person object.
- Checks if the Person meets the predicate condition.
- If the predicate condition is met, applies the function to get the person's name, modifies the person using the consumer, and prints the modified person.
- If the predicate condition is not met, prints a message indicating that the condition was not met.

Output:

Original Person: Person{name=Ram, age=45}

Person's Name: Ram

Person's Age: 45

Modified Person: Person{name=Sita, age=30}