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.

```
// Define the Pair class
public class Pair<T, U> {
    private T first;
    private U second;

    // Constructor to initialize the pair
    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    // Getter for the first element
    public T getFirst() {
        return first;
    }

    // Setter for the first element
    public void setFirst(T first) {
        this.first = first;
    }

    // Getter for the second element
    public U getSecond() {
        return second;
    }

    // Setter for the second element
    public void setSecond(U second) {
        this.second = second;
    }
```

```
    // Method to reverse the pair
    public Pair<U, T> reverse() {
        return new Pair<>(second, first);
    }

    // toString method for easy printing
    @Override
    public String toString() {
        return "Pair{" + "first=" + first + ", second=" + second + '}';
    }

    // Main method for testing the class
    public static void main(String[] args) {
        Pair<Integer, String> originalPair = new Pair<>(1, "one");
        System.out.println("Original Pair: " + originalPair);

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

Explanation

1) Pair Class Definition: The class Pair is defined with two generic types T and U.
2) Constructor: A constructor initializes the two elements of the pair.
3) Getters and Setters: Methods to get and set the values of the first and second elements.
4) reverse Method: The reverse method returns a new Pair object with the elements reversed.
5) toString Method: Overridden to provide a string representation of the pair.
6) Main Method: A main method to test the functionality of the Pair class and the reverse method.

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 {
```

```
    // Generic swap method
    public static <T> void swap(T[] array, int index1, int index2) {
        if (index1 < 0 || index1 >= array.length || index2 < 0 || index2 >= array.length) {
            throw new IndexOutOfBoundsException("Invalid indices for swapping");
        }

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

    // Demonstration
    public static void main(String[] args) {
        // Example 1: Swapping Integers
        Integer[] intArray = {1, 2, 3, 4};
        System.out.println("Before swapping integers: " + Arrays.toString(intArray));
        swap(intArray, 0, 2);  // Swap elements at index 0 and 2
        System.out.println("After swapping integers: " + Arrays.toString(intArray));

        // Example 2: Swapping Strings
        String[] strArray = {"Hello", "World", "Java"};
        System.out.println("Before swapping strings: " + Arrays.toString(strArray));
        swap(strArray, 1, 2);  // Swap elements at index 1 and 2
        System.out.println("After swapping strings: " + Arrays.toString(strArray));

        // Example 3: Swapping Custom Objects (assuming a Person class exists)
        Person[] peopleArray = {new Person("Alice"), new Person("Bob"), new Person("Charlie")};
        System.out.println("Before swapping people: " + Arrays.toString(peopleArray));
        swap(peopleArray, 0, 1);  // Swap elements at index 0 and 1
        System.out.println("After swapping people: " + Arrays.toString(peopleArray));
    }
}
```

Explanation:

Generic Method:

public static <T> void swap(T[] array, int index1, int index2):

The <T> declares a type parameter T, meaning the method can work with arrays of any type.

array is the array to swap elements within.

index1 and index2 are the indices of the elements to swap.

Input Validation:

The method checks if the provided indices are within the valid range of the array. If not, it throws an IndexOutOfBoundsException.

Swapping Logic:

It uses a temporary variable temp to hold the value of the element at index1.

The element at index1 is then overwritten with the element at index2.

Finally, the original value (stored in temp) is assigned to the element at index2.

Demonstration:

The main method showcases how to use the swap method with arrays of Integer, String, and a hypothetical Person class.

Notice how the same swap method handles different types seamlessly due to generics.

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.*;
import java.util.Arrays;

public class ReflectionDemo {

    // Sample class with a private field
    static class MyClass {
        private String privateField = "Initial Value";
```

```
    public int publicField = 10;

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

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

public static void main(String[] args) throws Exception {
    MyClass obj = new MyClass();

    // 1. Inspecting class members
    Class<?> cls = obj.getClass();
    System.out.println("Class Name: " + cls.getName());

    // Fields
    Field[] fields = cls.getDeclaredFields();
    System.out.println("Fields:");
    Arrays.stream(fields).forEach(System.out::println);

    // Methods
    Method[] methods = cls.getDeclaredMethods();
    System.out.println("Methods:");
    Arrays.stream(methods).forEach(System.out::println);

    // Constructors
    Constructor<?>[] constructors = cls.getDeclaredConstructors();
    System.out.println("Constructors:");
    Arrays.stream(constructors).forEach(System.out::println);

    // 2. Modifying a private field's value
```

```
        Field privateField = cls.getDeclaredField("privateField");
        privateField.setAccessible(true);  // Make private field accessible
        System.out.println("Original value: " + privateField.get(obj));

        privateField.set(obj, "Modified Value"); // Set new value
        System.out.println("Modified value: " + privateField.get(obj));

        // 3. Invoking a private method
        Method privateMethod = cls.getDeclaredMethod("privateMethod");
        privateMethod.setAccessible(true); // Make private method accessible
        privateMethod.invoke(obj);
    }
}
```

Explanation:

Sample Class: MyClass is a simple class with a private field, a public field, a public method, and a private method.

Get Class Object: We obtain a reference to the Class object representing MyClass using obj.getClass().

Inspecting Members: We use the Class object's methods (getDeclaredFields(), getDeclaredMethods(), getDeclaredConstructors()) to retrieve arrays of Field, Method, and Constructor objects, which provide information about the class members.

Modifying Private Field:

We get a reference to the private field using getDeclaredField().

We make it accessible by calling setAccessible(true).

We use get() to get the original value and set() to change the value of the private field.

Invoking Private Method:

Similar to field modification, we make the private method accessible.

We invoke the private method using invoke().

Task 4: Strategy

Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers"

1. SortingStrategy Interface:

```
public interface SortingStrategy {
    void sort(int[] numbers);
}
```

2. Concrete Sorting Strategies:

```
// Bubble Sort Strategy
class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        // ... (Bubble sort implementation)
    }
}

// Quick Sort Strategy
class QuickSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        // ... (Quick sort implementation)
    }
}

// Merge Sort Strategy
class MergeSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        // ... (Merge sort implementation)
```

```
    }
}
```

3. Context Class:

```
public class SortingContext {
    private SortingStrategy strategy;

    // Constructor
    public SortingContext(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    // Set a new sorting strategy at runtime
    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    // Sort the numbers using the current strategy
    public void sortNumbers(int[] numbers) {
        strategy.sort(numbers);
    }
}
```

4. Usage Example:

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 9, 1, 5, 6};

        // Using Bubble Sort
        SortingContext context = new SortingContext(new BubbleSortStrategy());
        context.sortNumbers(numbers);
        System.out.println("Sorted using Bubble Sort: " + Arrays.toString(numbers));
```

```
        // Changing to Quick Sort at runtime
        context.setStrategy(new QuickSortStrategy());
        context.sortNumbers(numbers);
        System.out.println("Sorted using Quick Sort: " + Arrays.toString(numbers));

        // Changing to Merge Sort at runtime
        context.setStrategy(new MergeSortStrategy());
        context.sortNumbers(numbers);
        System.out.println("Sorted using Merge Sort: " + Arrays.toString(numbers));
    }
}
```

How it Works:

Strategy Interface: Defines a common contract (sort) for all sorting algorithms.

Concrete Strategies: Each strategy class (e.g., BubbleSortStrategy, QuickSortStrategy) implements the SortingStrategy interface and provides its specific sorting logic.

Context Class:

Holds a reference to a SortingStrategy object.

Has a constructor to initialize the strategy.

Provides a setStrategy method to change the strategy dynamically.

Has a sortNumbers method that delegates the sorting to the currently selected strategy.

Client Code: Creates a SortingContext object, passing the initial sorting strategy. It can then call sortNumbers to sort the data and setStrategy to switch to a different algorithm at runtime.