Department Mathematik/Informatik, Abteilung Informatik
Software & System Engineering
Weiterführende Konzepte der Programmierung, SS 2025

Niklas Bockholt, Chiara Hafner, Alicja Bezpalko,
Jiyan Alis, Dilara Günay,
Simon Wolf, Adrian Bajraktari, Dr. Mersedeh Sadeghi

Homework 8. Introduction to Generics and Collection

Registration Deadline: 08.06.2025, 23:59
Hand-in Deadline: 11.06.2025, 23:59

Exercise 1. Shape Storage

You are designing a basic shape management system for a drawing application. All shapes implement a method to calculate their area and represent themselves as a string. You are also tasked with creating a generic storage class that can store shapes, compute their total area, and import shapes from another storage if they are large enough.

1) Create an abstract class `Shape` with:
   - An abstract method `double getArea()`.
   - An abstract method `toString()`.
2) Implement two concrete shape classes:
   - `Circle`, with a radius.
   - `Rectangle`, with width and length.
3) Create a generic class `ShapeStorage<T extends Shape>` that extends `ArrayList<T>` and implements:
   - `double getTotalArea()`: returns the sum of the areas of all stored shapes.
   - `void displayAllShapes()`: prints all shapes using their `toString()`.
   - `<U extends T> void importLargeShapes(ShapeStorage<U> other, double minArea)`: adds shapes from another storage to `this` storage if their area is at least `minArea`.
4) Use the provided `main` method to test your implementation.

```
package com.example;

public class Main {
    public static void main(String[] args) {
        ShapeStorage<Shape> allShapes = new ShapeStorage<>();
        ShapeStorage<Circle> smallCircles = new ShapeStorage<>();

        allShapes.add(new Rectangle(2, 3)); // 6 area
        smallCircles.add(new Circle(1)); // ~3.14 area
        smallCircles.add(new Circle(3)); // ~28.27 area

        System.out.println("Display allShapes:");
        allShapes.displayAllShapes();

        // Only circles with area >= 10 will be imported
        allShapes.importLargeShapes(smallCircles, 10.0);

        System.out.println("Display allShapes after import:");
        allShapes.displayAllShapes();

        System.out.printf("\nTotal area: %.2f\n",
                allShapes.getTotalArea());
    }
}
```

Exercise 2. Shape Analyzer

You have already implemented a generic class `ShapeStorage<T extends Shape>` that stores shape instances. Now, your task is to implement a utility class that performs analysis on collections (e.g. `ShapeStorage`) of shapes.

1) Implement a class `ShapeAnalyzer` (non-generic) that provides the following methods:
   - `static List<Shape> filterByMinArea(Collection<? extends Shape> shapes, double minArea)`
     Returns a list of shapes whose area is greater than or equal to the specified value.
   - `static Shape findShapeWithMaxArea(Collection<? extends Shape> shapes)`
     Returns the shape with the largest area.
   - `static <T extends Shape> Map<String, List<T>> groupByType(Collection<T> shapes)`
     Groups the given shapes by their concrete type (e.g., "Circle", "Rectangle") in a `Map` as seen in the return type.
     Hint: Use `.getClass().getSimpleName()` to get class names as Strings.
2) Use the provided `main` method to test your implementation.

Additional Questions:

1) Why is it important to use the wildcard `<? extends Shape>` in the method signature instead of a non-generic parameter? Hint: Think invariance.

2) Explain the design choice of using <T extends Shape> in groupByType instead of a wildcard.

3) Explain the design choice to make the methods in ShapeAnalyzer static

```
package com.example;

import java.util.List;
import java.util.Map;
import java.util.ArrayList;

public class Main2 {
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Circle(2.0));
        shapes.add(new Rectangle(3.0, 4.0));
        shapes.add(new Circle(1.0));
        shapes.add(new Rectangle(5.0, 5.0));
        shapes.add(new Circle(3.5));

        System.out.println("All Shapes:");
        for (Shape shape : shapes) {
            System.out.println(shape);
        }

        // filter shapes with area >= 15
        System.out.println("\nShapes with area >= 15:");
        List<Shape> filtered = ShapeAnalyzer
                .filterByMinArea(shapes, 15);

        for (Shape shape : filtered) {
            System.out.println(shape);
        }

        // find shape with max area
        Shape maxShape = ShapeAnalyzer
                .findShapeWithMaxArea(shapes);
        System.out.println("\nShape with max area:");
        System.out.println(maxShape);

        // group by type
        Map<String, List<Shape>> grouped = ShapeAnalyzer
                .groupByType(shapes);

        // print
        System.out.println("\nGrouped by type:");
        for (Map.Entry<String, List<Shape>> entry : grouped.entrySet()) {
            System.out.println(entry.getKey() + ":");
            for (Shape shape : entry.getValue()) {
                System.out.println("  " + shape);
            }
```

```
        }
    }
}
```

Exercise 3. Persiting Shapes

You are enhancing the shape management system to support **file persistence** and **robust parsing**. You'll use `BufferedReader` and `BufferedWriter` for efficient file operations.

1) **Create a class `ShapeFactory` with:**

   - `public static Shape fromString(String input)`
     Reads a string like `"Circle: radius=2.0"` or `"Rectangle: width=3.0, length=4.0"` and returns the appropriate `Shape` object.
     Throws `IllegalArgumentException` for invalid input.

2) **Create a class `PersistentShapeManager` with the following methods:**

   - `public static void saveShapesToFile(Collection<? extends Shape> shapes, String filename)`
     Saves all shapes to the file, one per line, using their `toString()` format. If the file already exists, it should **append** new shapes instead of overwriting it.
     Uses `BufferedWriter` and try-with-resources. Handles and logs any file-related exceptions gracefully.
   - `public static List<Shape> loadShapesFromFile(String filename)`
     Uses `BufferedReader` to read lines from the file. Each line represents a shape.
     Delegates parsing to `ShapeFactory.fromString()`.
     Skips malformed lines, logs them, and returns only valid `Shape` objects.
   - `static void clearFile(String filename)`
     Clears the contents of the specified file (i.e., truncates it to zero length), effectively making it empty without deleting the file. This can be used before saving if overwriting is explicitly desired.

3) **Use the provided `main` method to test your implementation.**

```
package com.example;

import java.util.ArrayList;
import java.util.List;

public class Main3 {
    public static void main(String[] args) {
        String filename = "shapes.txt";

        // 1. Create shapes
        List<Shape> shapesToSave = new ArrayList<>();
        shapesToSave.add(new Circle(3.5));
        shapesToSave.add(new Rectangle(5.5, 3.3));

        // optional: Clear the file before saving to it
        PersistentShapeManager.clearFile(filename);

        // 2. Save to file
        PersistentShapeManager
                .saveShapesToFile(shapesToSave, filename);

        // 3. Load from file
        List<Shape> loadedShapes = PersistentShapeManager
                .loadShapesFromFile(filename);

        // 4. Display loaded shapes
        System.out.println("Shapes loaded from file:");
        for (Shape shape : loadedShapes) {
            System.out.println(shape);
        }
    }
}
```