Object-Oriented Programming in Java

Lecture 6 - Abstract Elements

Emily Lucia Antosch

HAW Hamburg

14.08.2025

Contents

1. Introduction

1.1 Where Are We Currently?

- The last lecture was about inheritance
- You can now
  - create and use simple inheritance lines,
  - override methods from the base class,
  - use the `equals()` method to compare objects with each other,
  - reference objects via the respective base class
- Today we continue with **Interfaces**.

1.1 Where Are We Currently?

1. Imperative Concepts
2. Classes and Objects
3. Class Library
4. Inheritance
5. **Interfaces**
6. Graphical User Interfaces
7. Exception Handling
8. Input and Output
9. Multithreading (Parallel Computing)

1.2 The Goal of This Chapter

- You model common properties of classes by extending classes with common interfaces (in the form of abstract base classes or interfaces).
- You hide the data type of objects by referencing objects via interfaces when accessing common properties of different classes.
- You sort a collection of objects of the same data type according to arbitrary criteria

2. Abstract Elements & Methods

2.1 Abstract Elements

? Question

- Do you remember our geometric objects?
- What bothers you about the current structure of our classes?
- What doesn't make sense or is "ugly"?

2.1 Abstract Elements

? Question

- Do you remember our geometric objects?
- What bothers you about the current structure of our classes?
- What doesn't make sense or is "ugly"?

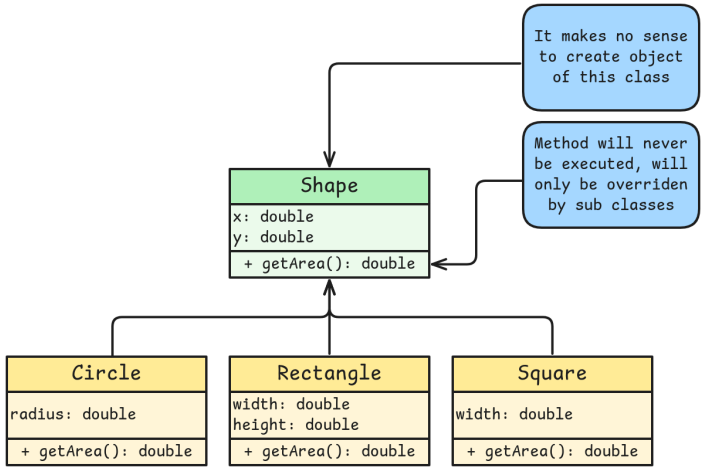


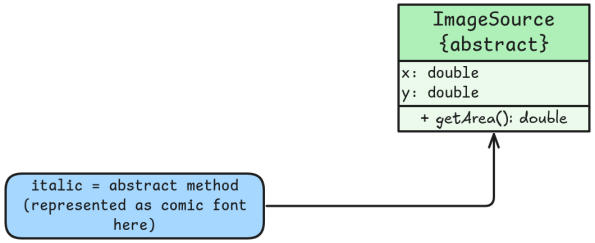
Figure 1: Overriding the method from Shape

2.1 Abstract Elements

- Class becomes abstract class through keyword abstract
- Effect: No objects of the class can be created.
- Instead:
  - Derive class and extend in (concrete = non-abstract) subclasses
  - Create objects of the subclasses

Java

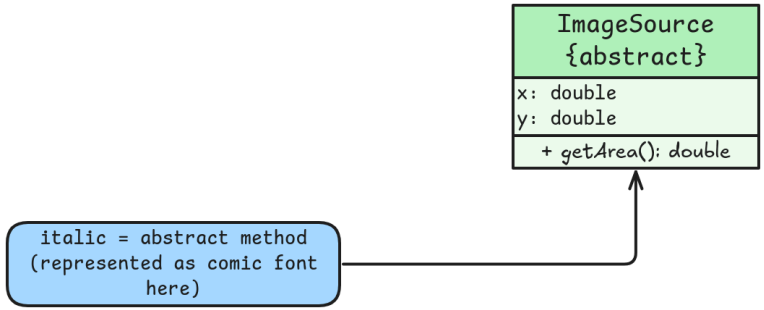
```java
public abstract class A {
    // ...
}

public class B extends A {
    // ...
}

public static void main(String[] args) {
    A abstractObj = new A();
    B concreteObj = new B();
}
```



2.1 Abstract Elements

- Method becomes abstract method through the keyword abstract
- Abstract method contains only the declaration, but no implementation

```java
public abstract class ImageSource {
        String name;

        public abstract Image getNextImage();
    }
```

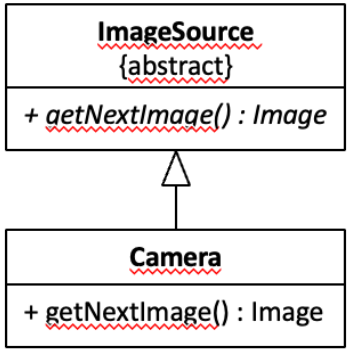


Memorize

- Abstract methods cannot be called (no implementation exists!)
- Instead specifies which methods subclasses must have

2.1 Abstract Elements

- Classes with abstract methods must be abstract.
- Otherwise, non-implemented methods could be called for objects.
- Inheritance:
  - Abstract methods are inherited.
  - Subclasses remain abstract as long as not all abstract methods are implemented




```java
public abstract class ImageSource {
    String name;

    public abstract Image getNextImage();
}

public class Camera extends ImageSource {
    public Image getNextImage() {
        // ...
    }
}
```

2.1 Abstract Elements

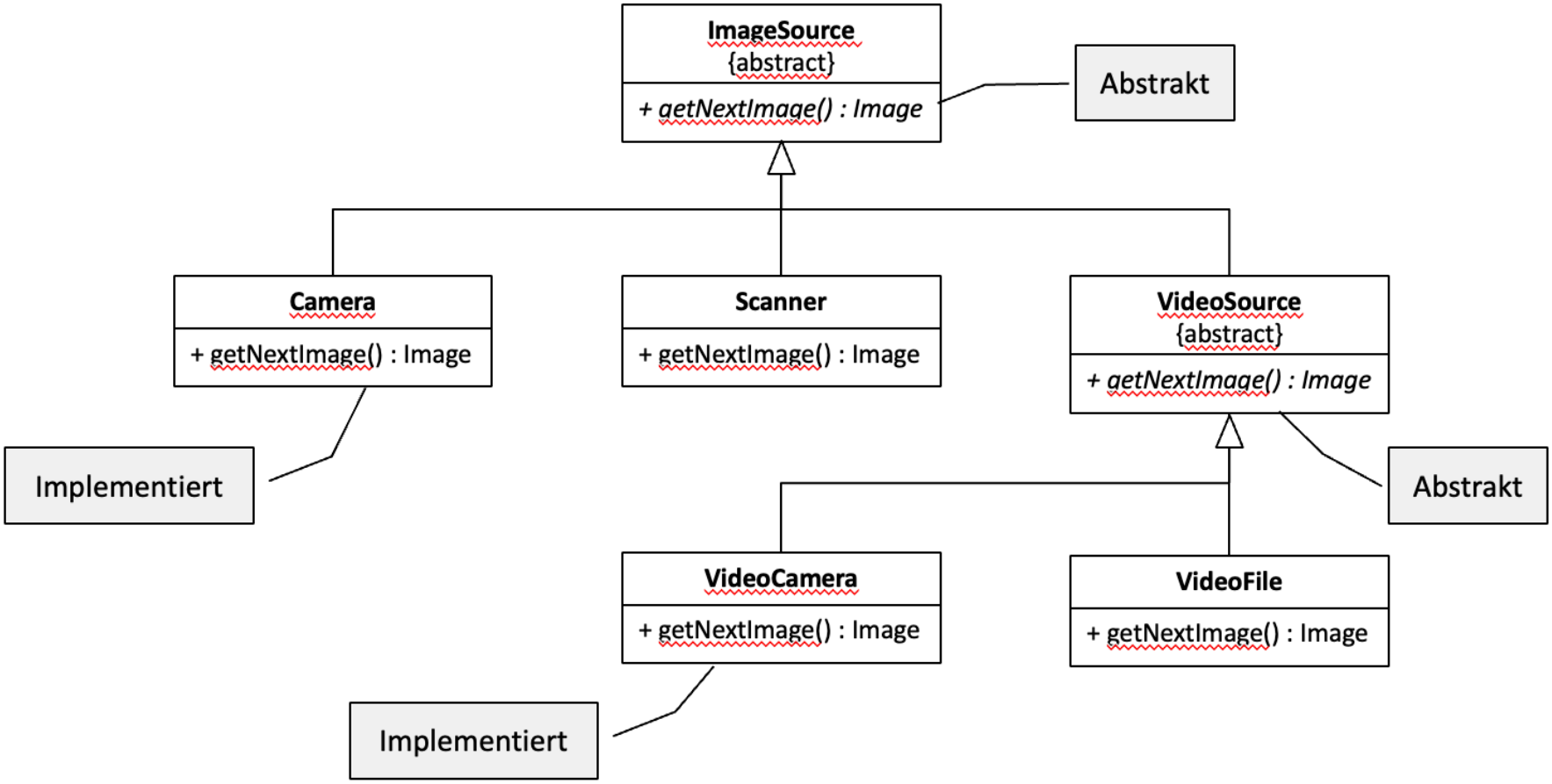
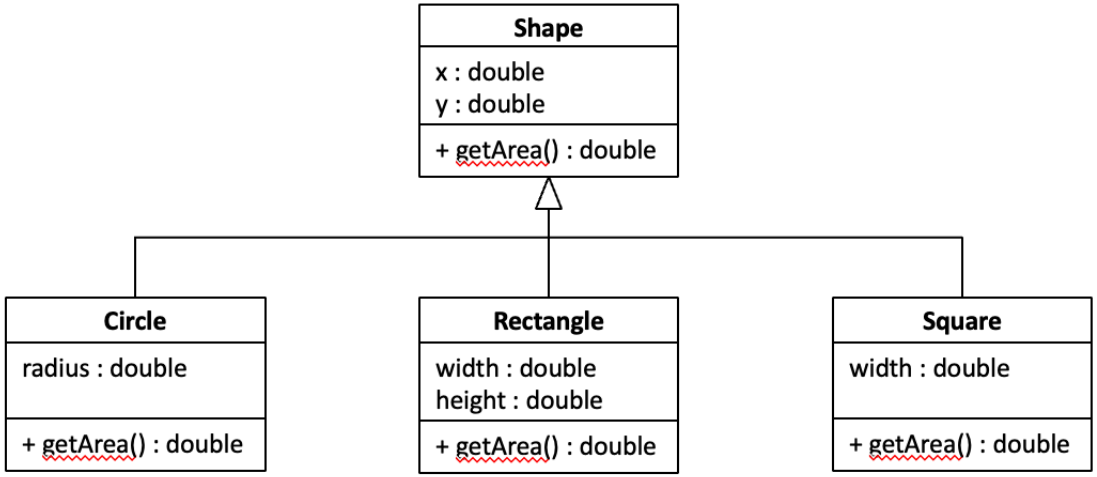


Figure 5: Large overview of abstract classes and methods

2.1 Abstract Elements

Task 1

- Now improve the structure of the class hierarchy.
- Use abstract elements for this.



2.1 Abstract Elements

- No objects of class Shape, but only of concrete geometric shapes
- All classes for geometric shapes have getArea().
- Implementation depends on the type of geometric shape

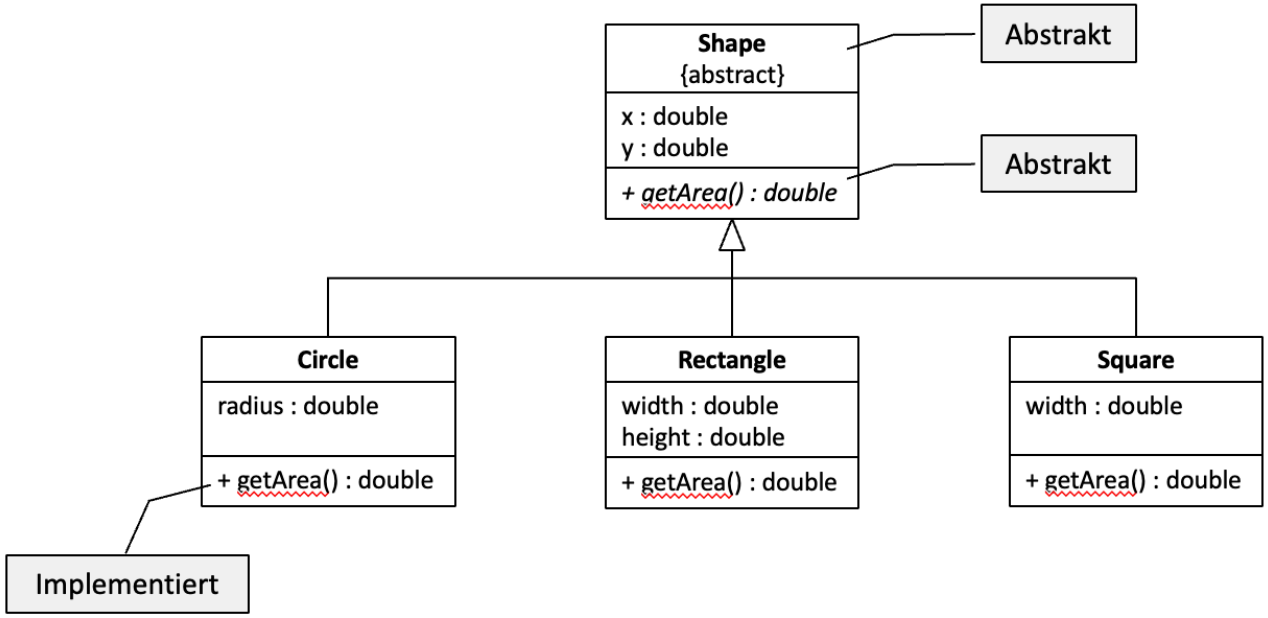


Figure 7: Abstract and implemented methods

3. Interfaces

3.1 Interfaces

- Classes (as a reminder):
  - Concrete classes cannot contain abstract methods.
  - Abstract classes can additionally contain abstract methods.
- Basic idea of an interface:
  - Declares only abstract methods
  - Therefore specifies which methods a class must implement
  - Contains no variables (No object creatable: no constructors needed)
  - Often describe properties (e.g. Comparable, Cloneable, Scalable, …)

3.1 Interfaces

- Visibility:
  - All methods are public abstract (even when modifiers are missing).
  - All attributes are public static final (even when modifiers were missing).
- From Java 8 also implemented methods:
  - Default methods: Comparable to conventional methods in a class
  - Static methods

3.1 Interfaces

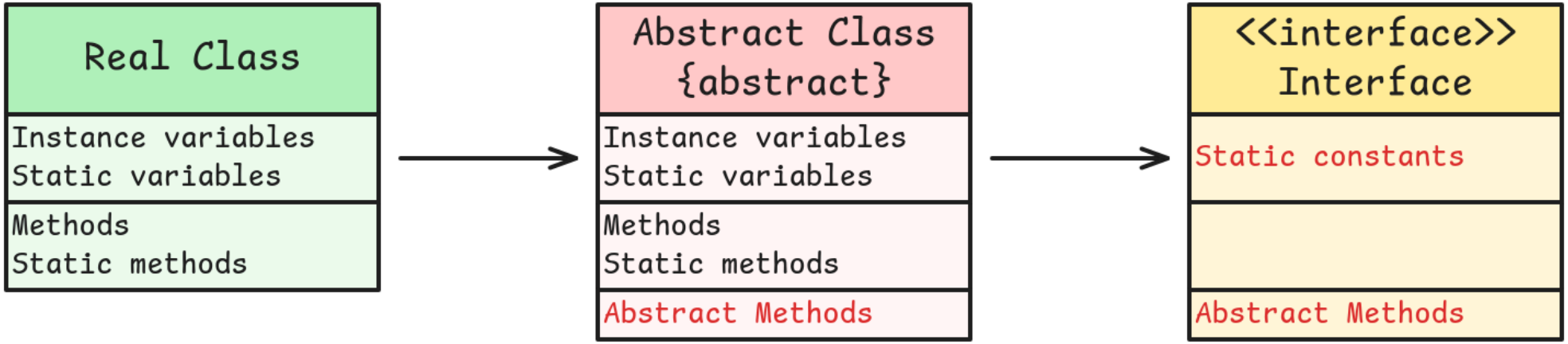
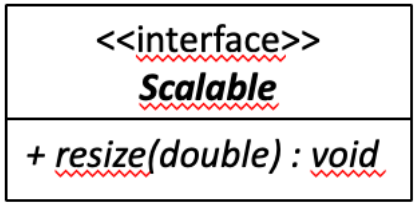


Figure 8: Gradient between Real, Abstract and Interface

3.1 Interfaces

- Declaration of an interface:

```java
Modifier interface InterfaceName {
Constants
Abstract Methods
Default methods and static methods
}
```



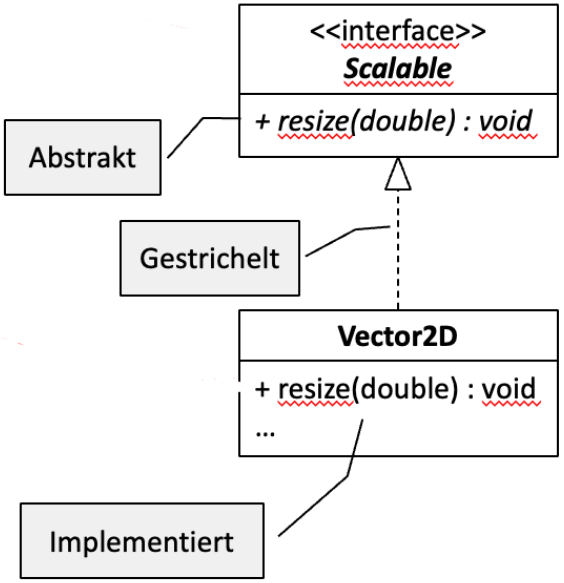
- Declare method `resize()` to change size of an object

```java
public interface Scalable {
    void resize(double factor);
}
```

3.1 Interfaces

- Classes implement interfaces via the keyword implements
- Class inherits elements of the interface and implements abstract methods

```java
public class Vector2D implements Scalable {
    private double x, y;

    public Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public void resize(double factor) {
        x *= factor;
        y *= factor;
    }
    // Additional methods ...
}
```



3.1 Interfaces

- Interface method not implemented: Method remains abstract
- Therefore the class is also abstract
- Subclasses only become concrete when all abstract methods are implemented

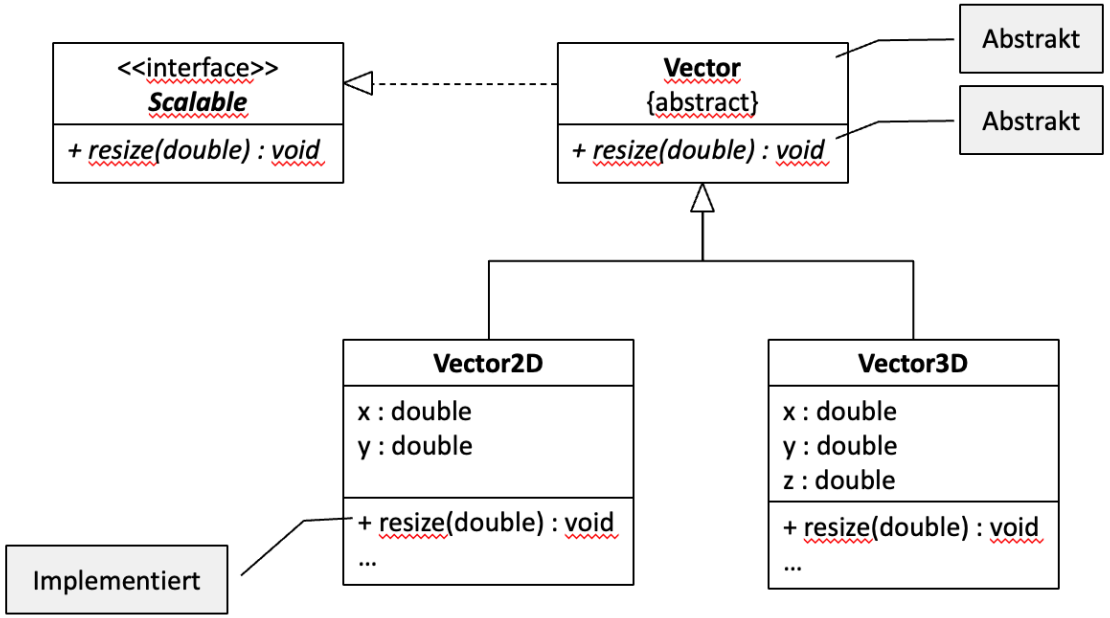
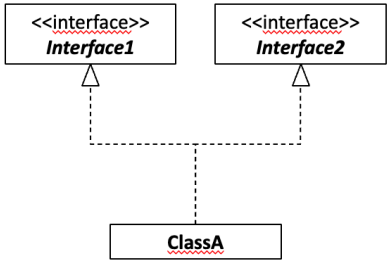


Figure 11: Abstract classes and interfaces

3.1 Interfaces

- As a reminder: Multiple inheritance for classes not allowed
- But: Implementation of any number of interfaces (separated by commas) allowed



Java

```java
interface Interface1 {
    // ...
}

interface Interface2 {
    // ...
}

class ClassA implements Interface1, Interface2 {
    // ...
}
```

3.1 Interfaces

Class GrayImage implements Scalable, Drawable and Rotateable

```java
public class GrayImage implements Scalable, Drawable, Rotateable {
    // Attributes and constructors
    // Interface methods
    // Additional methods
}
```

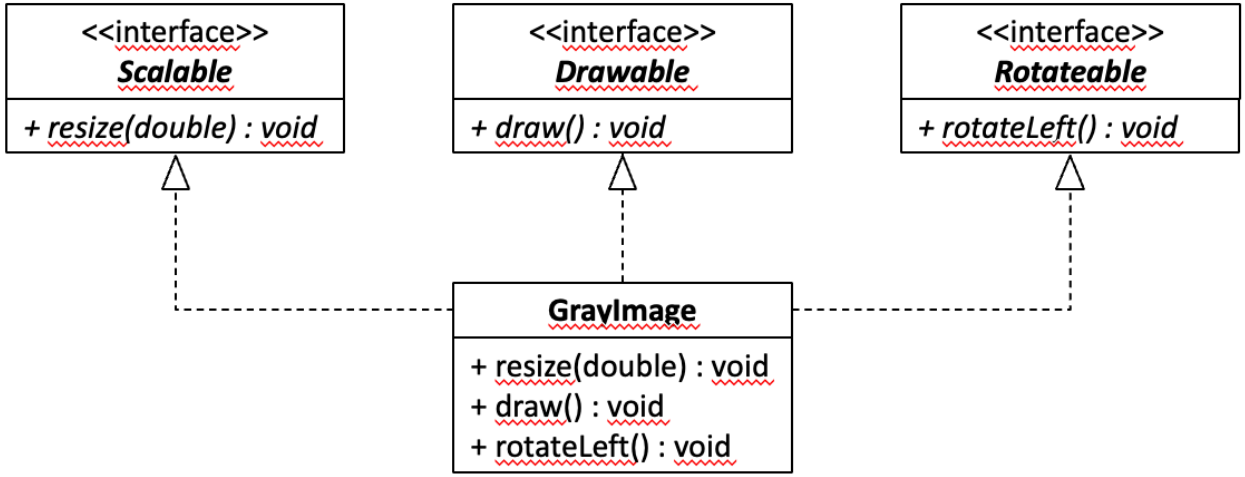
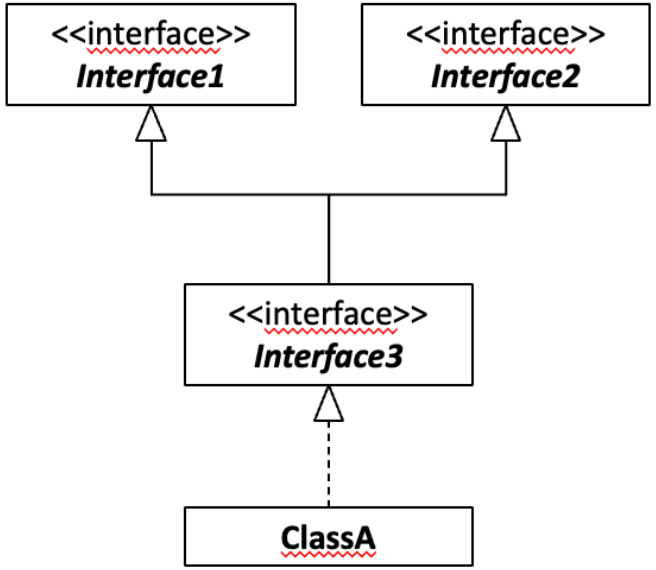


Figure 13: GrayImage-Beispiel

3.1 Interfaces

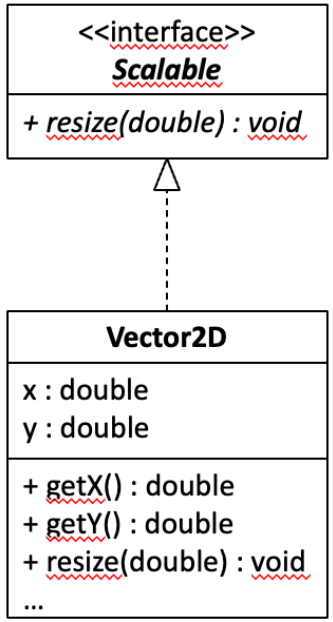
- Interfaces can be derived through extends.
- Multiple inheritance is allowed for interfaces!

```java
interface Interface1 {
    // ...
}

interface Interface2 {
    // ...
}

interface Interface3 extends Interface1, Interface2 {
    // ...
}

class ClassA implements Interface3 {
    // ...
}
```



3.1 Interfaces

- Just like with base classes:
  - Objects referenceable via data types of their implemented interfaces
  - Reference variable can only access attributes and methods of its interface



? Question

- Which accesses to attributes are allowed and which are not?

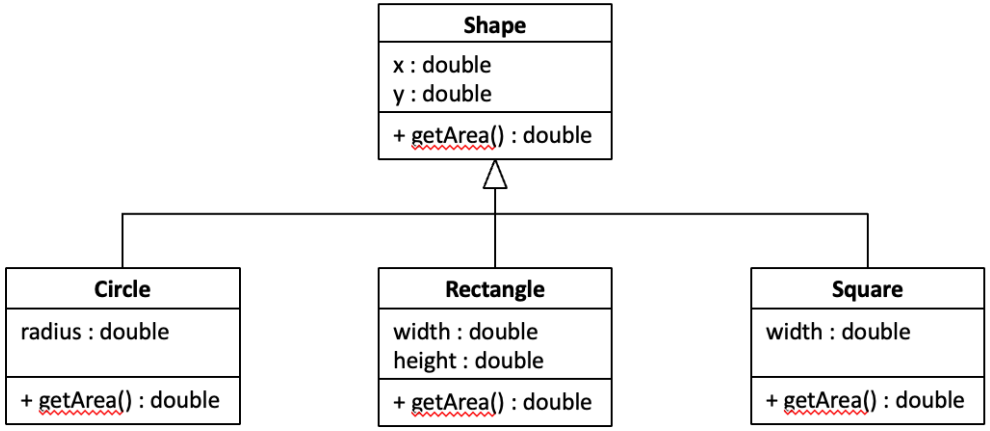
Java

```java
public static void main(String[] args) {
    Vector2D classRef = new Vector2D(1, 3);
    Scalable interRef = classRef;

    classRef.resize(1.5);
    System.out.println(classRef.getX());
    interRef.resize(1.5);
    System.out.println(interRef.getX());
}
```

3.1 Interfaces

Task 2

- Create an interface Transformable with the following methods:
  - Move
  - Rotation by 90° (one method each for rotation left and right)
  - Scale
  - Implement the interface in all classes of geometric shapes

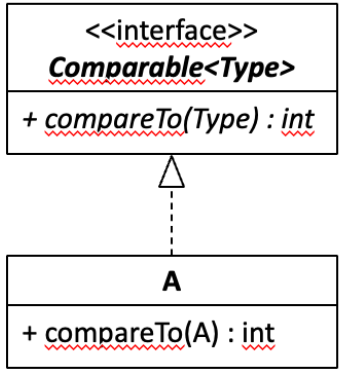


4. Comparison (Interface `Comparable`)

4.1 Interface Comparable

- Comparison of objects (Which is “larger”, which is “smaller”?)

```java
public interface Comparable<Type> {
    public int compareTo(Type other);
}
```



- Usage:
  - Implement interface in own class
  - Replace placeholder Type with own class name
  - Return value is interpreted as follows:

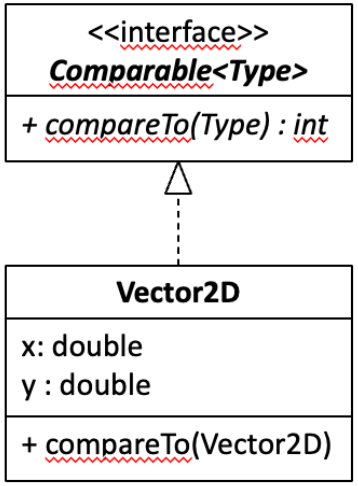
Return Value	Meaning
Negative	`this < other`
Null	`this == other`
Positive	`this > other`

Table 1: Formats and Flags

4.1 Interface Comparable

- Comparison of vectors based on magnitude:

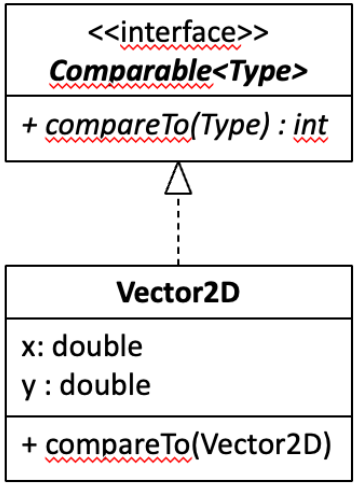
```java
public class Vector2D implements Comparable<Vector2D> {
    double x, y;

    public double getAbs() {
        return Math.sqrt(x * x + y * y);
    }

    public int compareTo(Vector2D other) {
        if (getAbs() < other.getAbs()) {
            return -1;
        } else if (getAbs() > other.getAbs()) {
            return 1;
        } else {
            return 0;
        }
    }
}
```



4.1 Interface Comparable

- Sort lists via class method `Collections.sort()`
- Prerequisite: Elements in list implement Comparable
- Method `sort()` uses the comparison method `compareTo()` pairwise

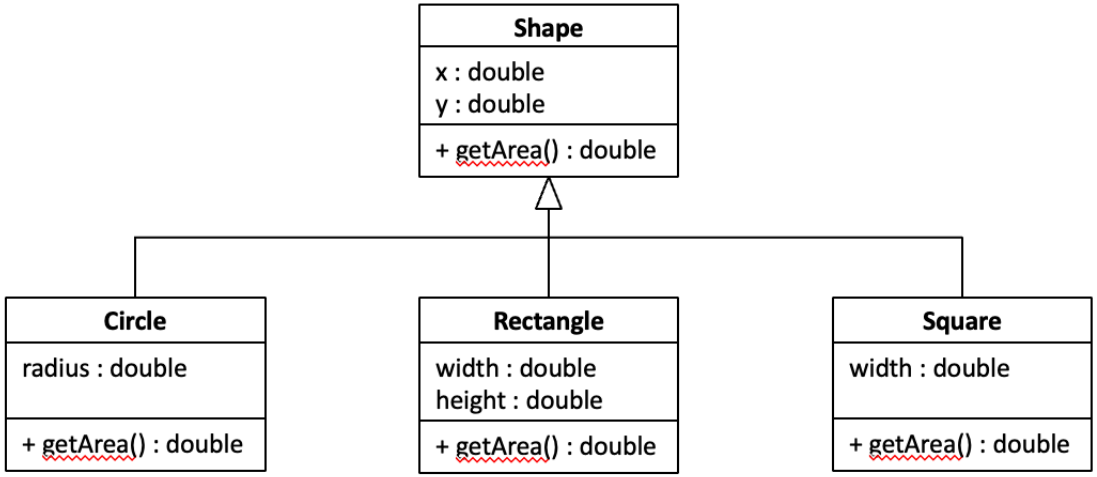
```java
public static void main(String[] args) {
    ArrayList<Vector2D> vectors = new ArrayList<Vector2D>();
    vectors.add(new Vector2D(0, 5));
    vectors.add(new Vector2D(0, -1));
    vectors.add(new Vector2D(7, 8));
    vectors.add(new Vector2D(0, 0));

    Collections.sort(vectors);
    for (Vector2D vector : vectors) {
        System.out.println(vector.getAbs());
    }
}
```



4.1 Interface `Comparable`

Task 3

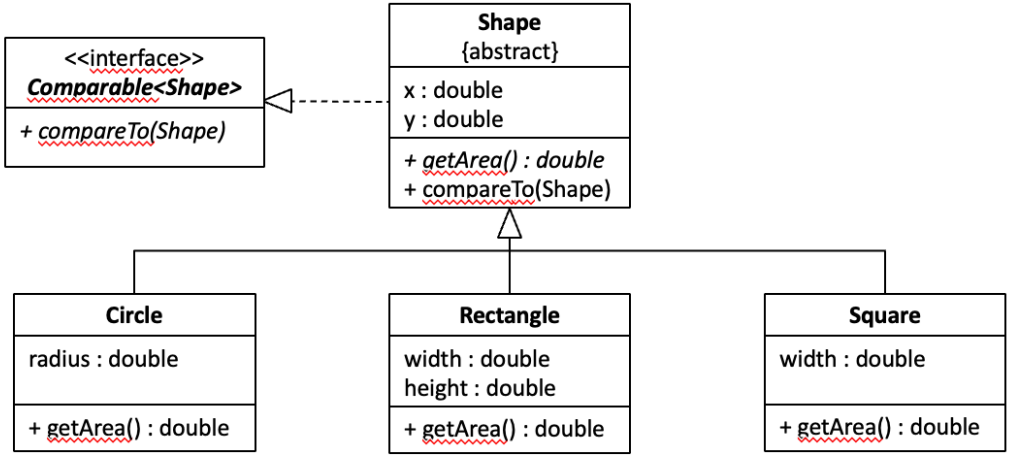
- Implement `Comparable<Type>` for geometric objects.
- Criterion for the comparison is the area of the objects.



4.1 Interface `Comparable`

! Memorize

- Only the class `Shape` must implement `Comparable`.
- The remaining classes inherit the interface and implementation.



4.1 Interface Comparable

- Implementation in Shape:

```java
public abstract class Shape implements Comparable<Shape> {
    // Attributes and other methods ...

    public int compareTo(Shape other) {
        double thisArea = getArea();
        double otherArea = other.getArea();

        if (thisArea < otherArea) {
            return -1;
        } else if (thisArea > otherArea) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

4.1 Interface Comparable

Java

```java
public static void main(String[] args) {
    ArrayList<Shape> shapes = new ArrayList<Shape>();
    shapes.add(new Circle(0.0, 0.0, 2.0));
    shapes.add(new Circle(0.0, 0.0, 1.0));
    shapes.add(new Rectangle(0.0, 0.0, 10.0, 5.0));
    shapes.add(new Square(0.0, 0.0, 0.5));

    System.out.println("Areas (unsorted):");
    for (Shape shape : shapes) {
        System.out.println(shape.getArea());
    }

    Collections.sort(shapes);
    System.out.println("\nAreas (sorted):");
    for (Shape shape : shapes) {
          System.out.println(shape.getArea());
      }
  }
```

5. License Notice

5.1 Attribution

- This work is based off of the work Prof. Dr. Marc Hensel.
- Some of the images and texts, as well as the layout were changed.
- The base material was supplied in private, therefore the link to the source cannot be shared with the audience.