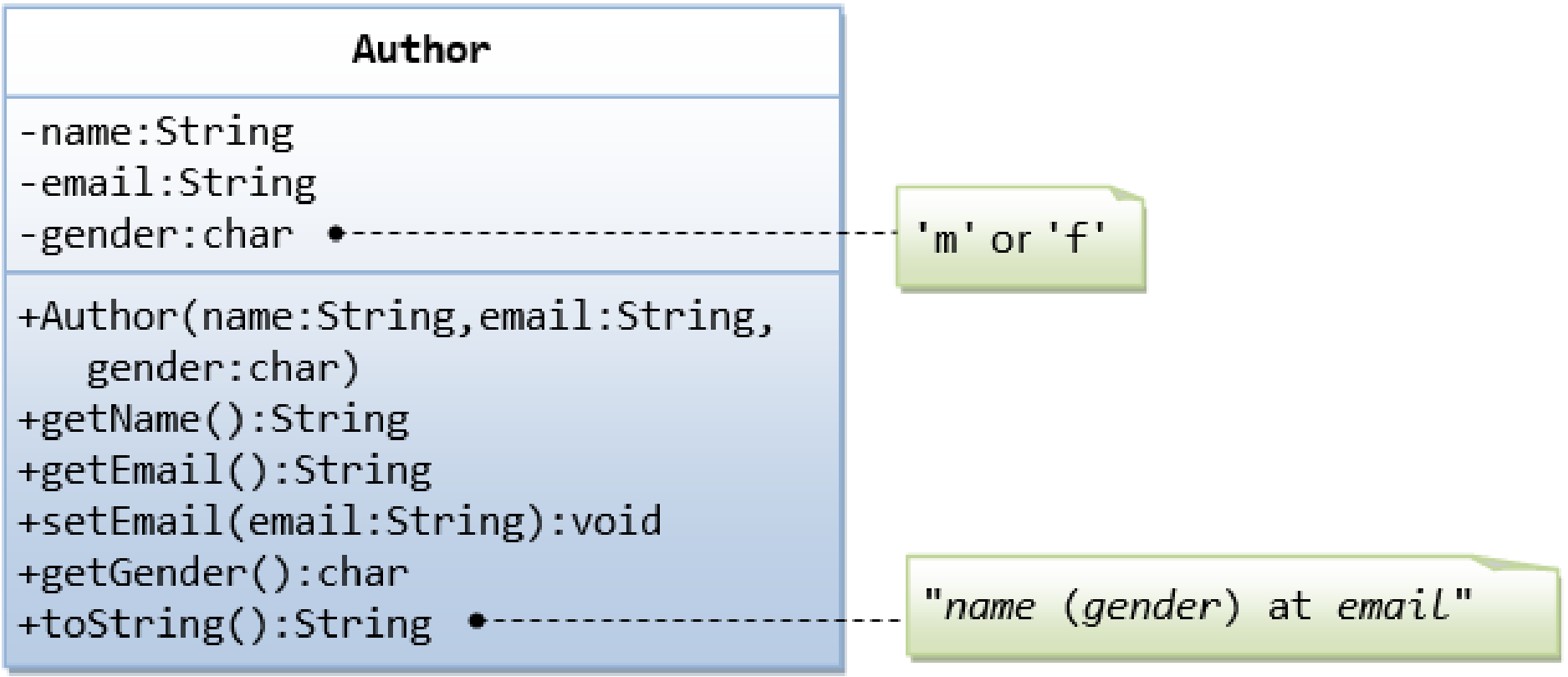
Java Programming
OOP - Composition, Inheritance & Polymorphism

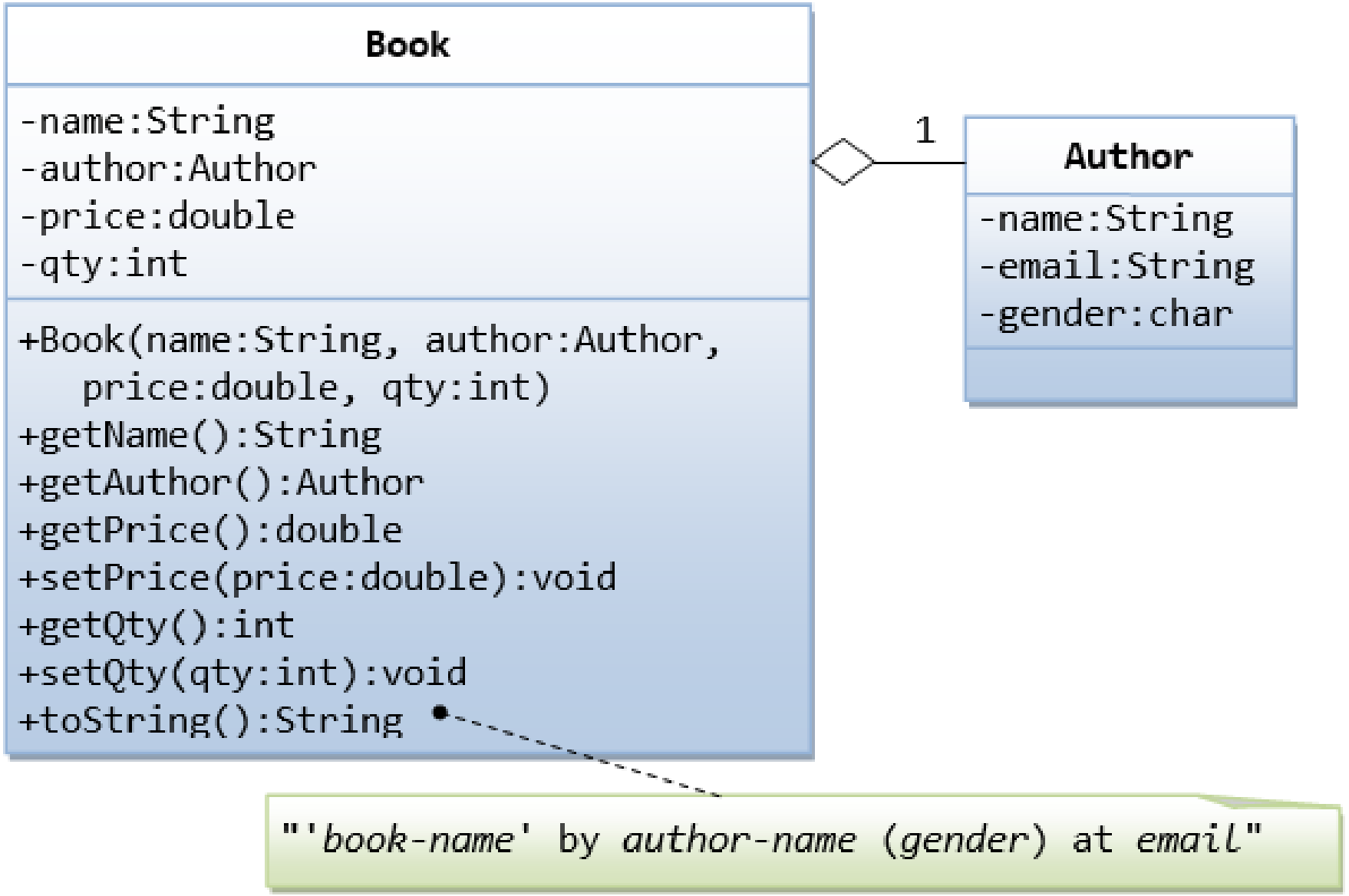
Composition

- There are two ways to reuse existing classes, namely, composition and inheritance.
- With composition (aka aggregation), you define a new class, which is composed of existing classes.
- With inheritance, you derive a new class based on an existing class, with modifications or extensions.

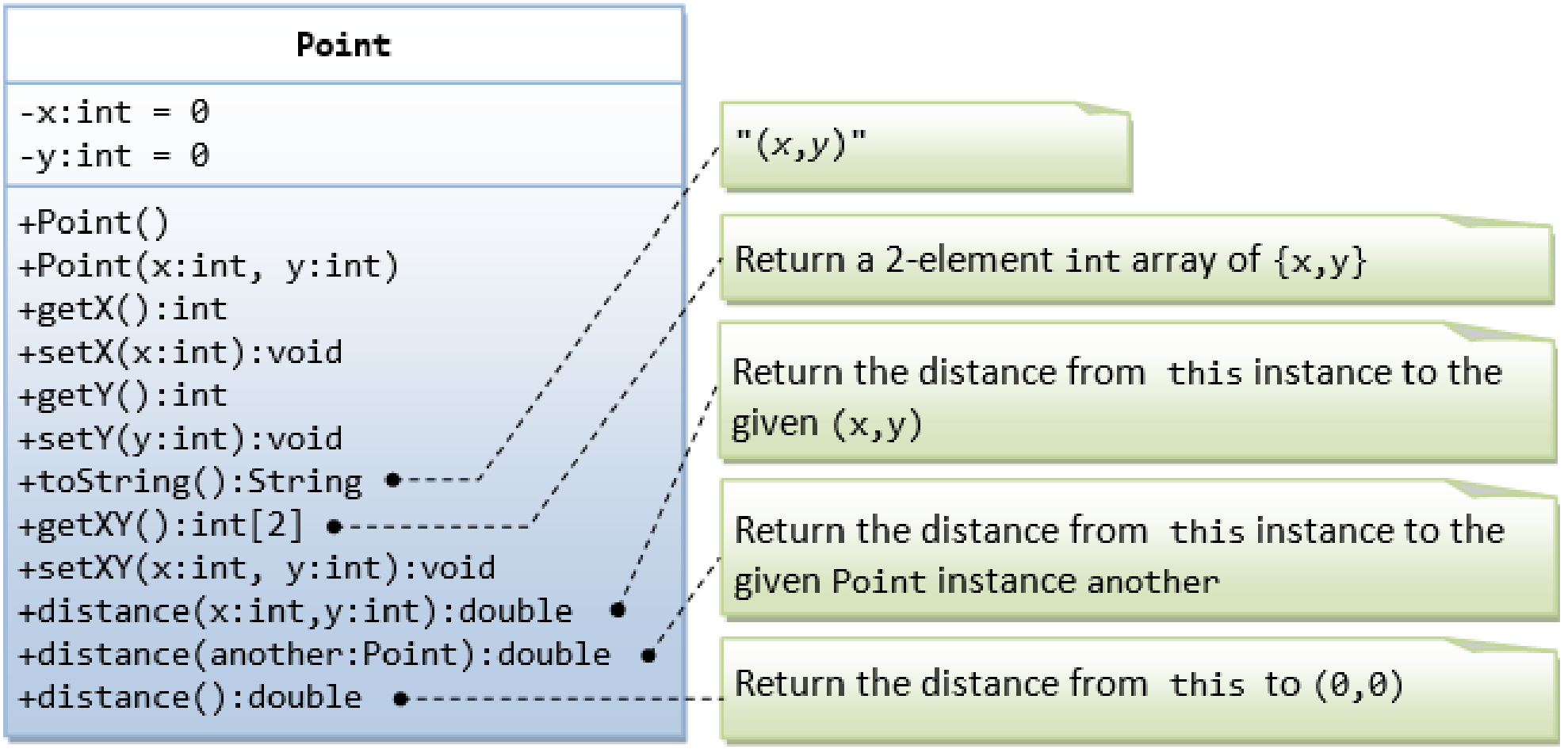
Composition - Example



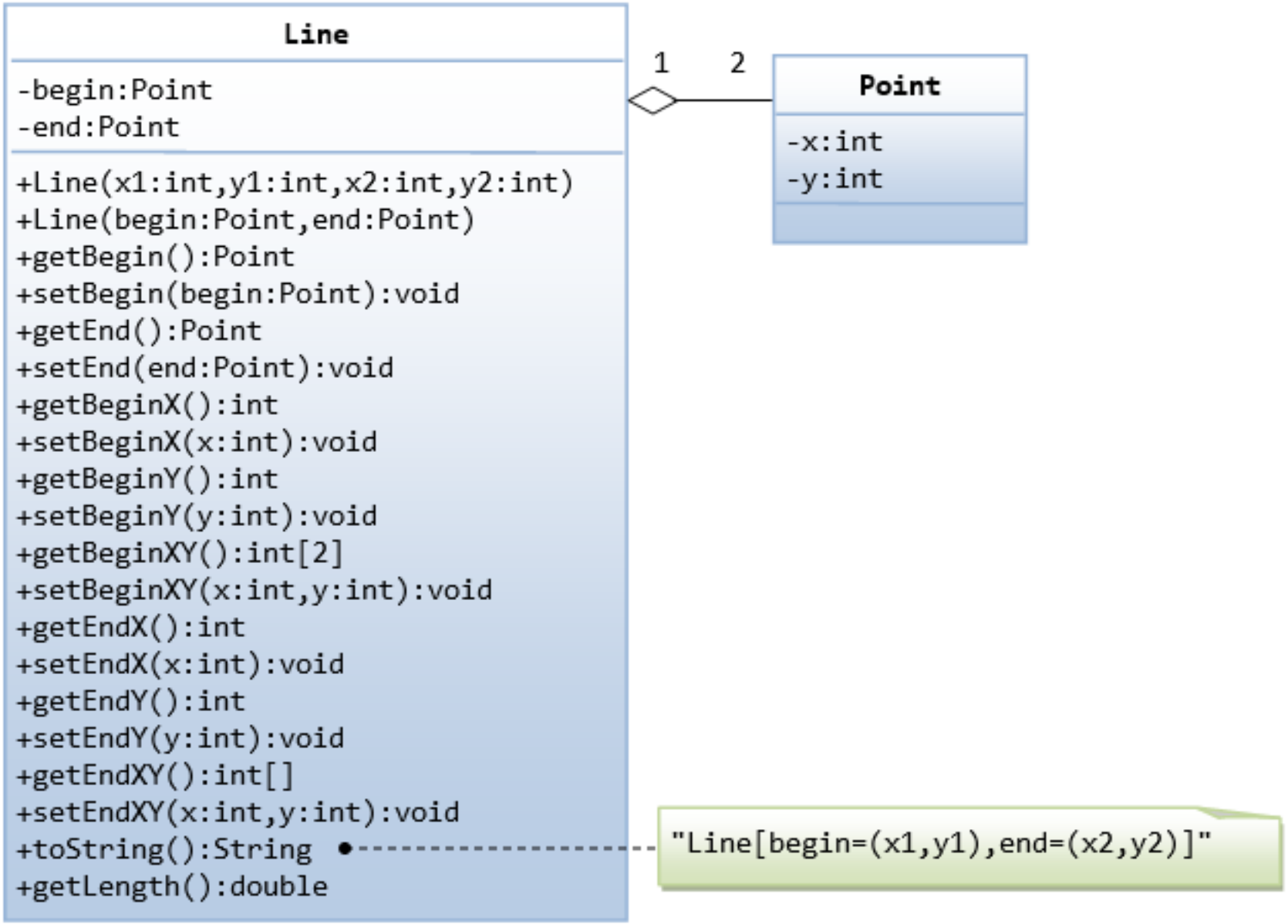
Composition - Example



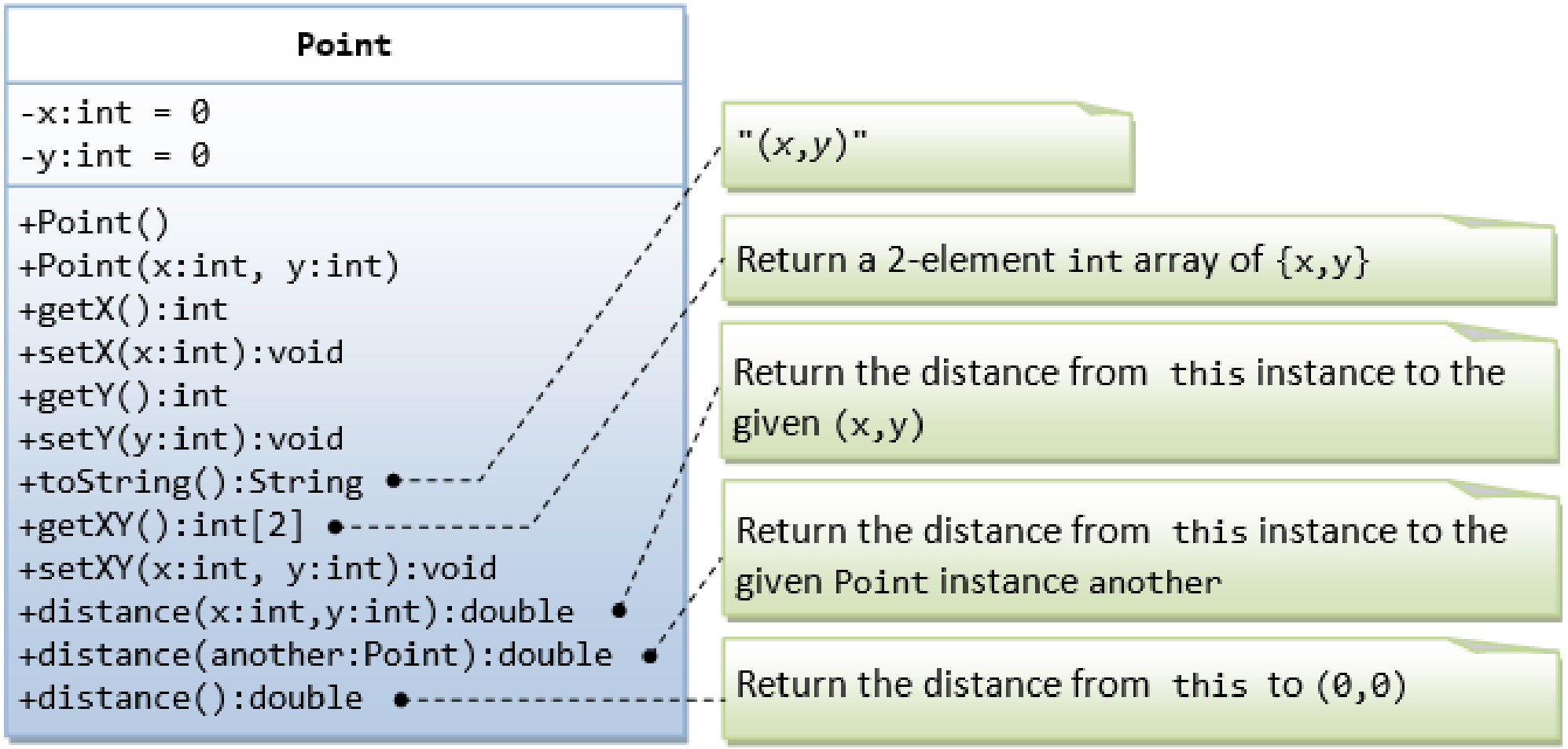
Composition – Example 2



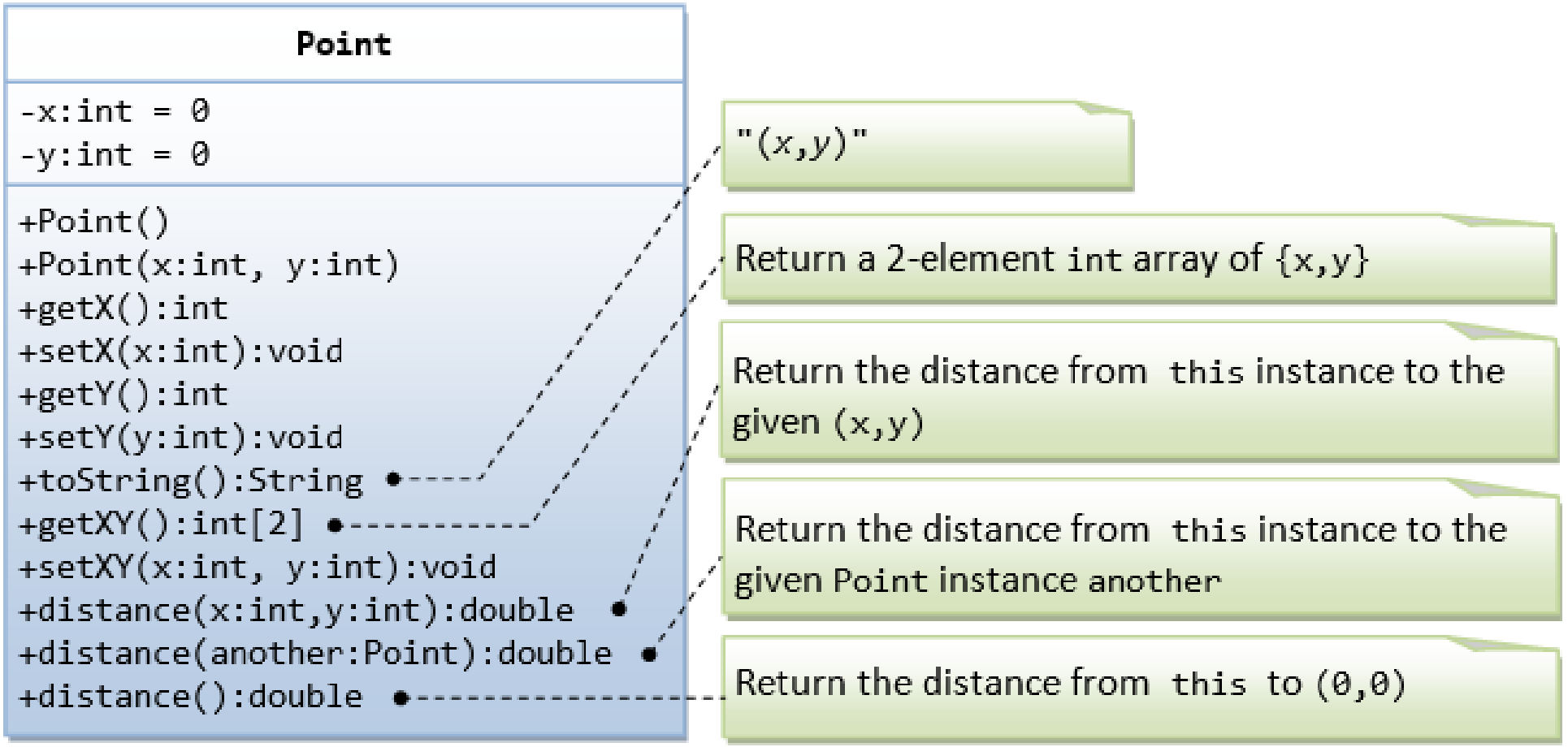
Composition – Example 2



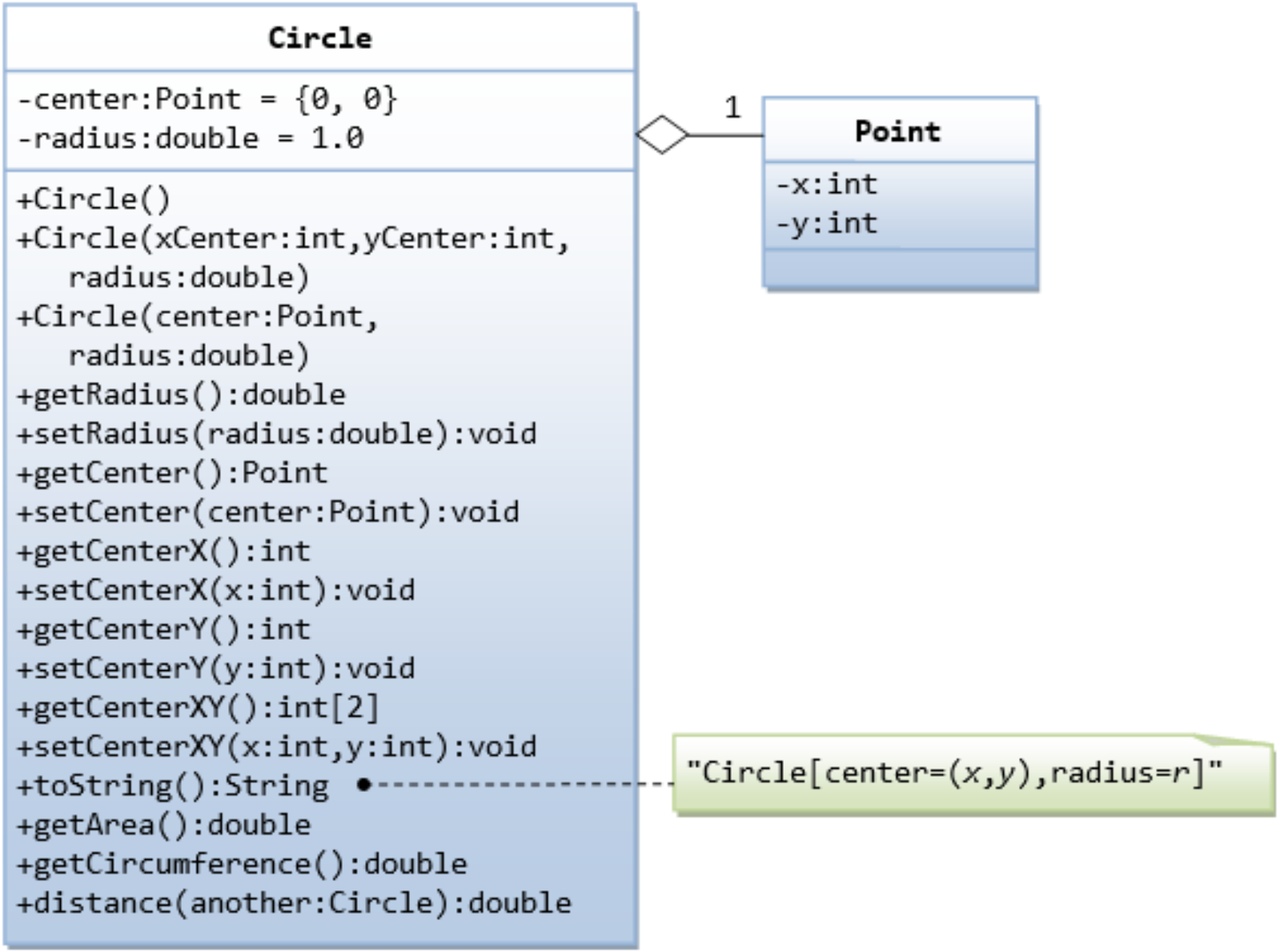
Composition – Example 2



Composition – Example 3



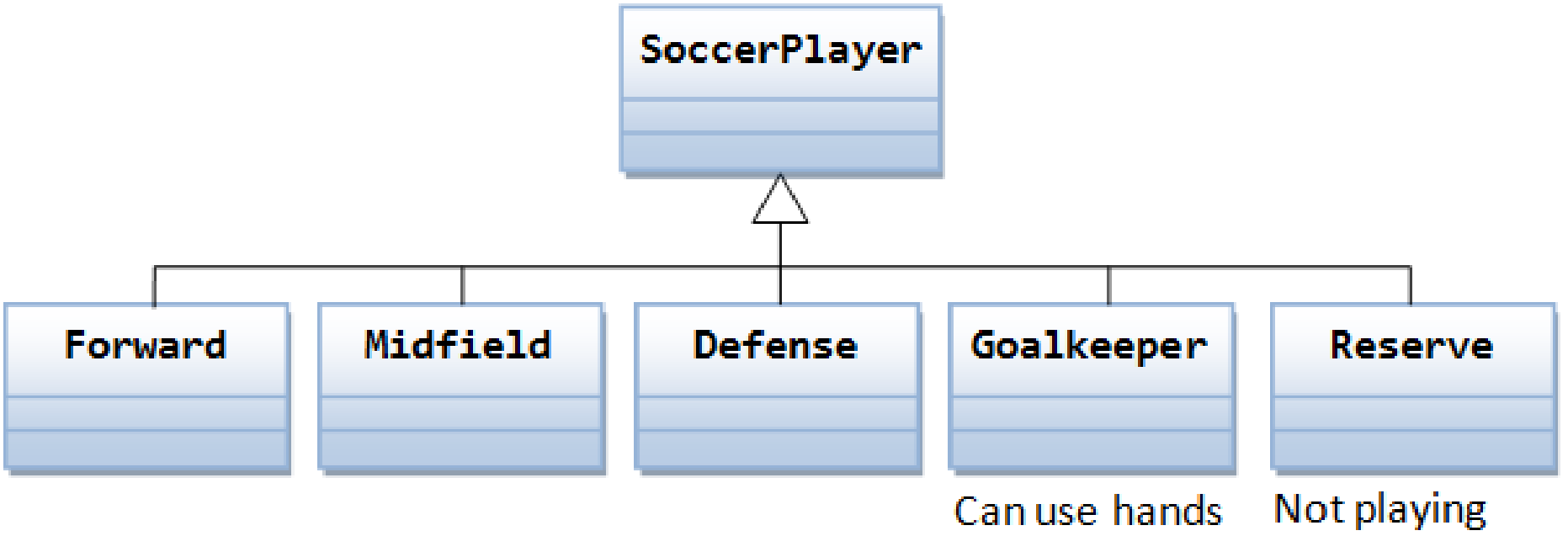
Composition – Example 3



Inheritance

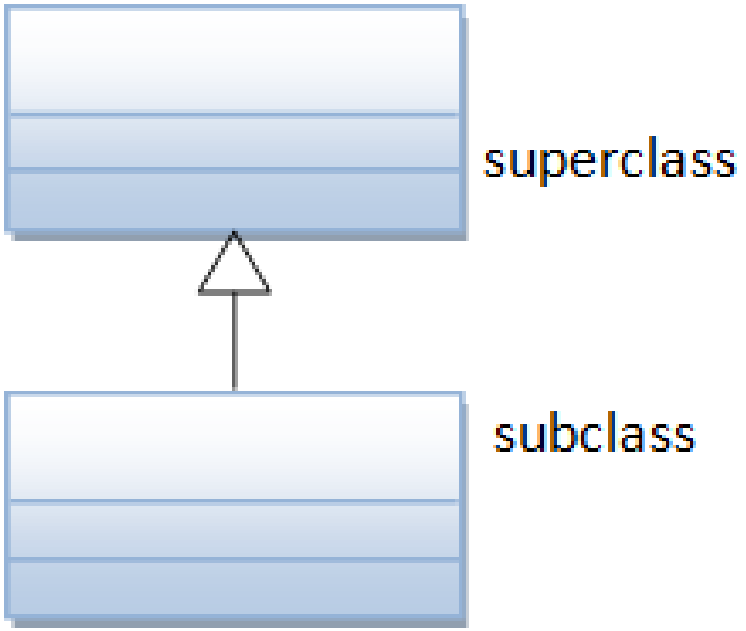
- In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy.
- The classes in the lower hierarchy inherit all the variables and method from the higher hierarchies.
- A class in the lower hierarchy is called a subclass (or derived, child, extended class).
- A class in the upper hierarchy is called a superclass (or base, parent class).
- A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.

Inheritance

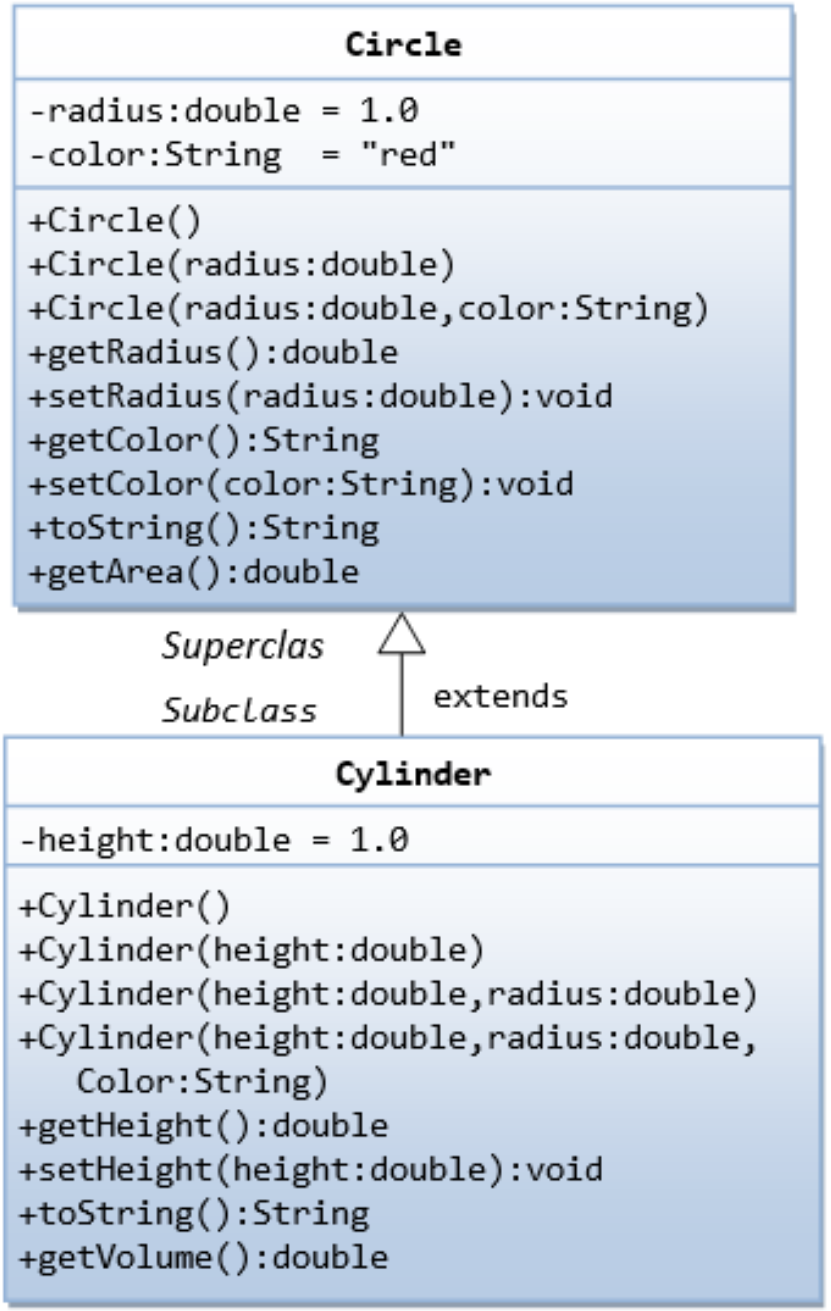


Inheritance

- UML Notation: The UML notation for inheritance is a solid line with a hollow arrowhead leading from the subclass to its superclass. By convention, superclass is drawn on top of its subclasses as shown.



Inheritance - Example



Method Overriding – Variable Hiding

- A subclass inherits all the member variables and methods from its superclasses.
- It can use the inherited methods and variables as they are.
- It may also override an inherited method by providing its own version.
- It may also hide an inherited variable by defining a variable of the same name.
- When the method is overridden, we can use super to access the inherited method.

Polymorphism

- The word "polymorphism" means "many forms".
- Each of the form has it own distinct properties.

Polymorphism - Substitutability

- A subclass possesses all the attributes and operations of its superclass.
- This means that a subclass object can do whatever its superclass can do.
- we can substitute a subclass instance when a superclass instance is expected, and everything shall work fine.

Polymorphism - Substitutability

- In our earlier example of Circle and Cylinder: Cylinder is a subclass of Circle.
- We can say that Cylinder "is-a" Circle.
- Subclass-superclass exhibits a so called "is-a" relationship.

Polymorphism - Substitutability

- // Substitute a subclass instance to a superclass reference
- Circle c1 = new Cylinder(1.1, 2.2);
- c1.getRadius(); // Invoke superclass Circle's methods
- However, you CANNOT invoke methods defined in the Cylinder class for the reference c1, e.g. c1.getHeight();

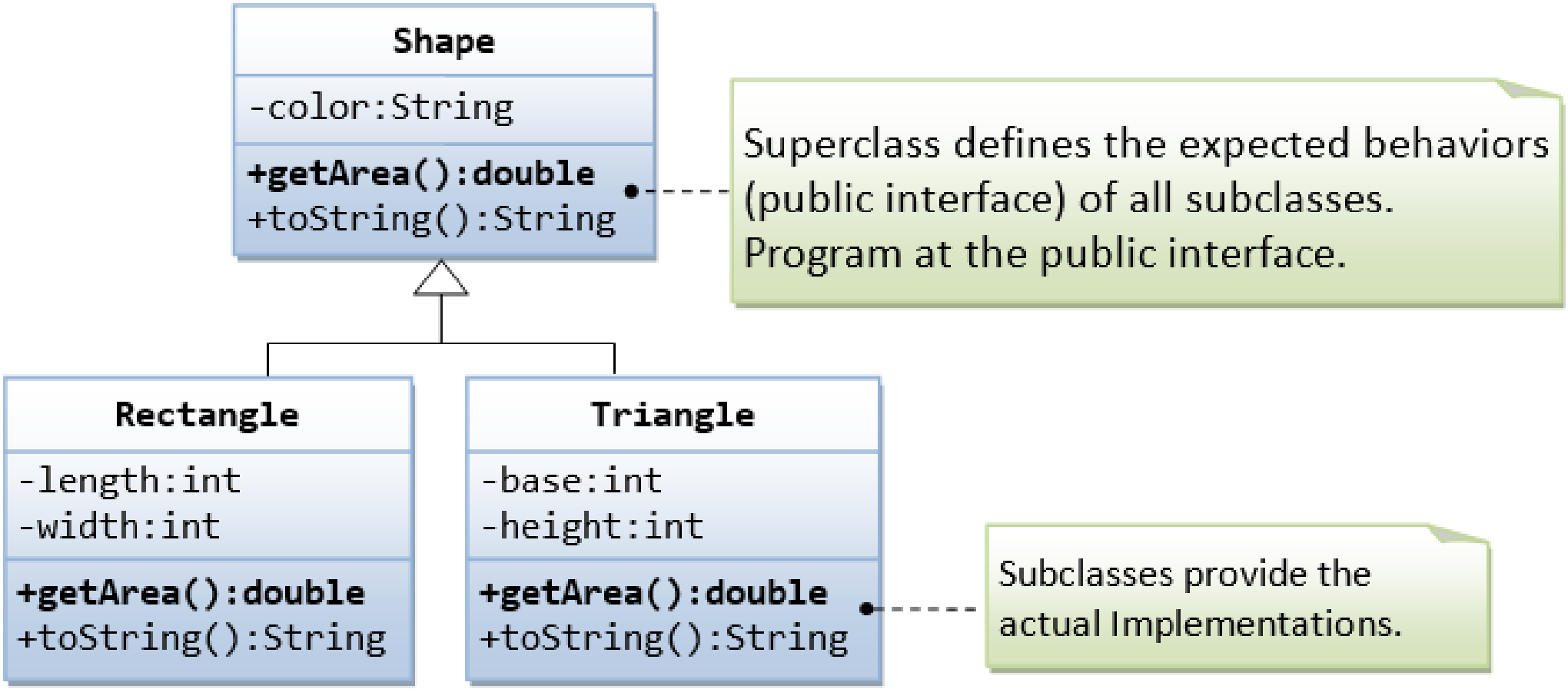
Polymorphism - Substitutability

- c1 is a reference to the Circle class, but holds an object of its subclass Cylinder.
- The reference c1, however, retains its internal identity.
- c1.toString() invokes the overridden version defined in the subclass Cylinder, instead of the version defined in Circle.
- This is because c1 is in fact holding a Cylinder object internally.

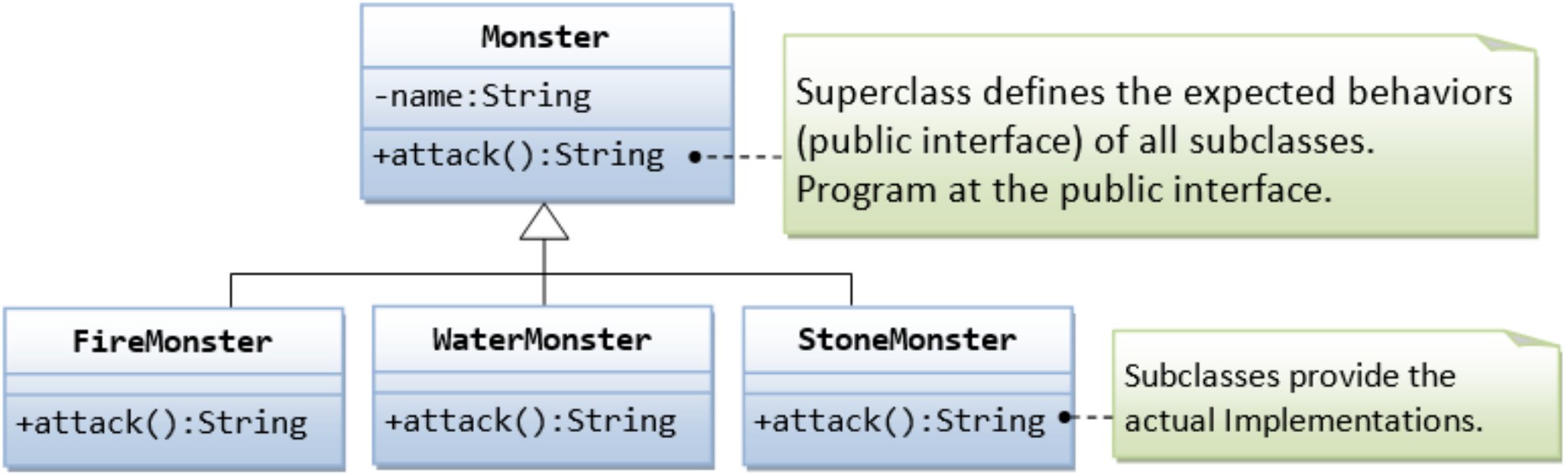
Polymorphism - Example

- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes.

Polymorphism - Example



Polymorphism - Example



Upcasting & Downcasting

- Substituting a subclass instance for its superclass is called "upcasting".
- Upcasting is always safe because a subclass instance possesses all the properties of its superclass.
- Circle c1 = new Cylinder(1.1, 2.2);
  // Compiler checks to ensure that R-value is a subclass of L-value.
- Circle c2 = new String();
  // Compilation error: incompatible types

Upcasting & Downcasting

- You can revert a substituted instance back to a subclass reference. This is called "downcasting".
- Circle c1 = new Cylinder(1.1, 2.2);
  // upcast is safe
- Cylinder cy1 = (Cylinder) c1;
  // downcast needs the casting operator

Upcasting & Downcasting

- Downcasting is not always safe, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass.
- Downcasting to an incompatible type might not give compilation error but would throw a runtime error.
- A subclass object can be substituted for its superclass, but the reverse is not true.

Abstract Classes & Interfaces

- An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method’s body).
- You use the keyword abstract to declare an abstract method.

Abstract Classes & Interfaces

- For example, in the Shape class, we can declare abstract methods getArea()

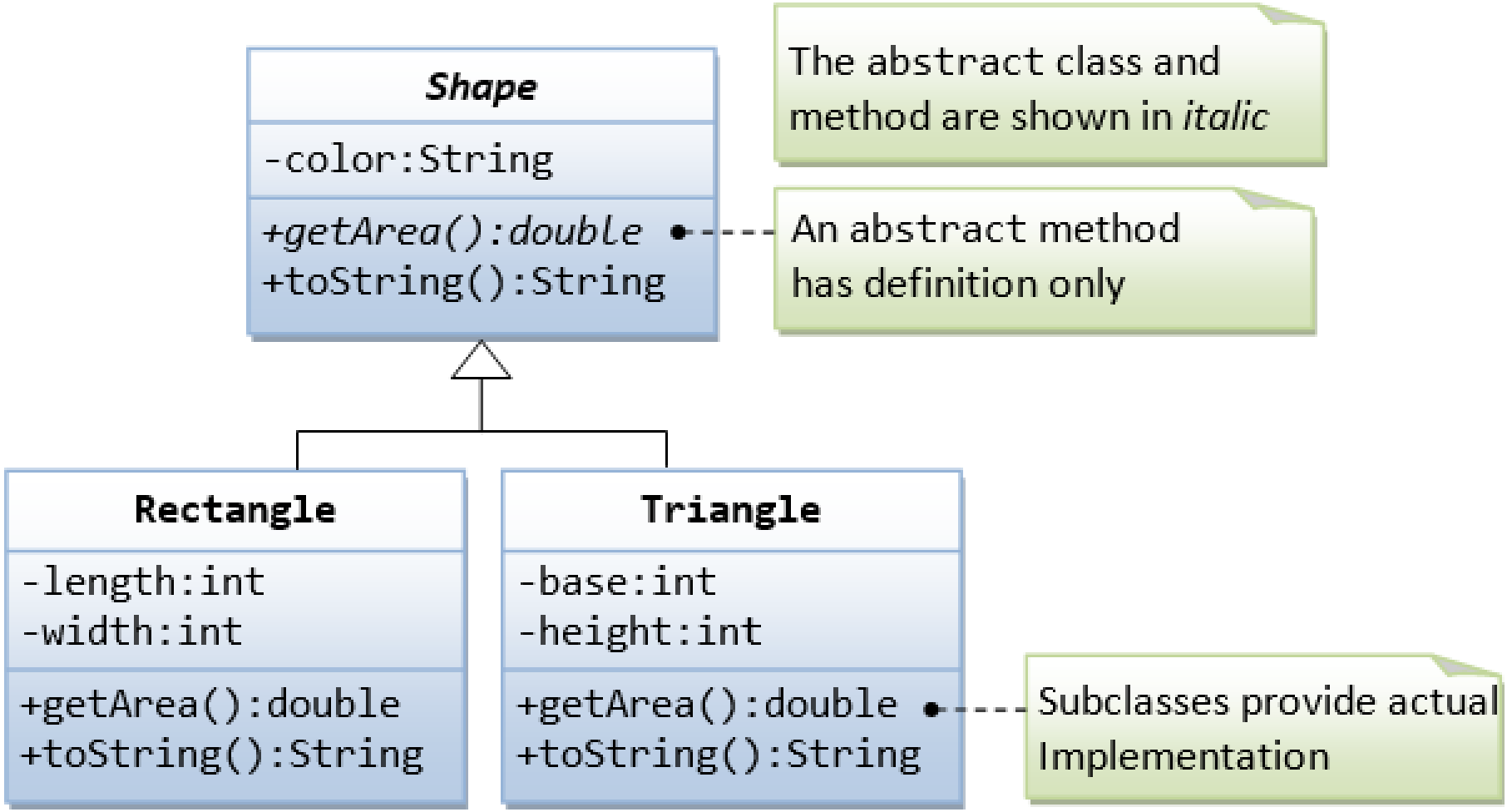
```
abstract public class Shape {
    ......
    ......
    abstract public double getArea();
    abstract public double getPerimeter();
    abstract public void draw();
}
```

Abstract Classes & Interfaces

- Implementation of these methods is NOT possible in the Shape class, as the actual shape is not yet known.
- A class containing one or more abstract methods is called an abstract class.
- An abstract class must be declared with a class-modifier abstract.
- An abstract class CANNOT be instantiated, as its definition is not complete.

Abstract Classes & Interfaces

- UML Notation: abstract class and method are shown in italic.



Abstract Class - Example

```
/*
 * The abstract superclass Monster defines the expected common
 * behaviors,
 * via abstract methods.
 */
abstract public class Monster {
   private String name;  // private instance variable

   public Monster(String name) {  // constructor
      this.name = name;
   }

   // Define common behavior for all its subclasses
   abstract public String attack();
}
```

The Java's interface

- A Java interface is a 100% abstract superclass which define a set of methods its subclasses must support.
- An interface contains only public abstract methods and possibly constants (public static final variables).

The Java's Interface

- You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes).
- The keyword public and abstract are not needed for its abstract methods as they are mandatory.
- Similar to an abstract superclass, an interface cannot be instantiated.

The Java's Interface

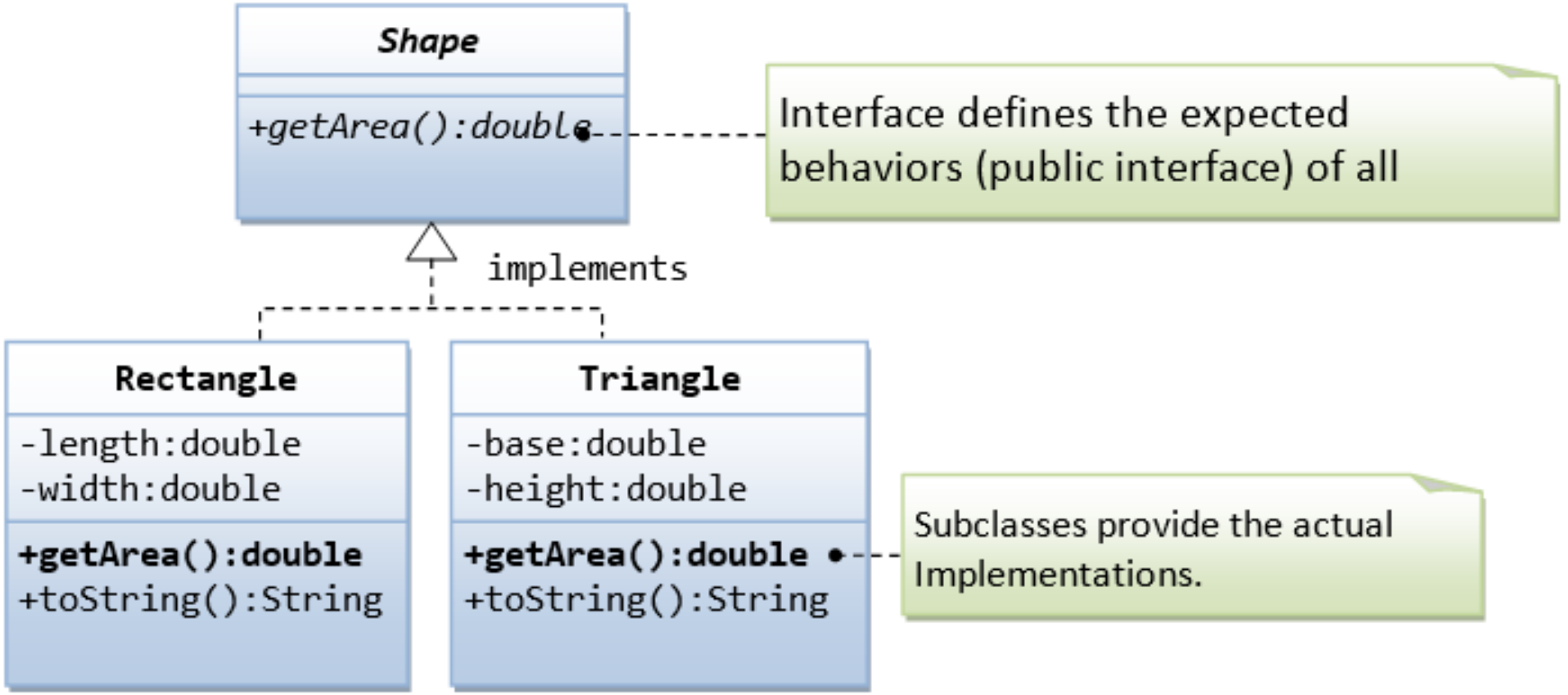
- For interface, we use the keyword "implements" to derive a subclass.
- An interface is a contract for what the classes can do. It, however, does not specify how the classes should do it.
- All methods in an interface shall be public and abstract (default).
- You cannot use other access modifier such as private, protected and default, or modifiers such as static, final.
- All fields shall be public, static and final (default).
- If a subclass implements two interfaces with conflicting constants, the compiler will flag out a compilation error.

The Java’s Interface

- Interface Naming Convention:-
- Use an adjective (typically ends with "able") consisting of one or more words.
- Each word shall be initial capitalized (camel-case).
- For example, Serializable, Extenalizable, Movable, Clonable, Runnable, etc

Interface - Example

- We can re-write the abstract superclass Shape into an interface, containing only abstract methods



Interface - Example

```
/*
 * The interface Shape specifies the behaviors
 * of this implementations subclasses.
 */
public interface Shape {  // Use keyword "interface" instead of "class"
   // List of public abstract methods to be implemented by its subclasses
   double getArea();
}
```

Interface - Example

```
// The subclass Rectangle needs to implement all the abstract methods in Shape
public class Rectangle implements Shape {  // using keyword "implements" instead of "extends"
   // Private member variables
   private int length;
   private int width;

   // Constructor
   public Rectangle(int length, int width) {
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle[length=" + length + ",width=" + width + "]";
   }

   // Need to implement all the abstract methods defined in the interface
   @Override
   public double getArea() {
      return length * width;
   }
}
```

Interface - Example

```
// The subclass Triangle need to implement all the abstract methods in Shape
public class Triangle implements Shape {
   // Private member variables
   private int base;
   private int height;

   // Constructor
   public Triangle(int base, int height) {
      this.base = base;
      this.height = height;
   }

   @Override
   public String toString() {
      return "Triangle[base=" + base + ",height=" + height + "]";
   }

   // Need to implement all the abstract methods defined in the interface
   @Override
   public double getArea() {
      return 0.5 * base * height;
   }
}
```

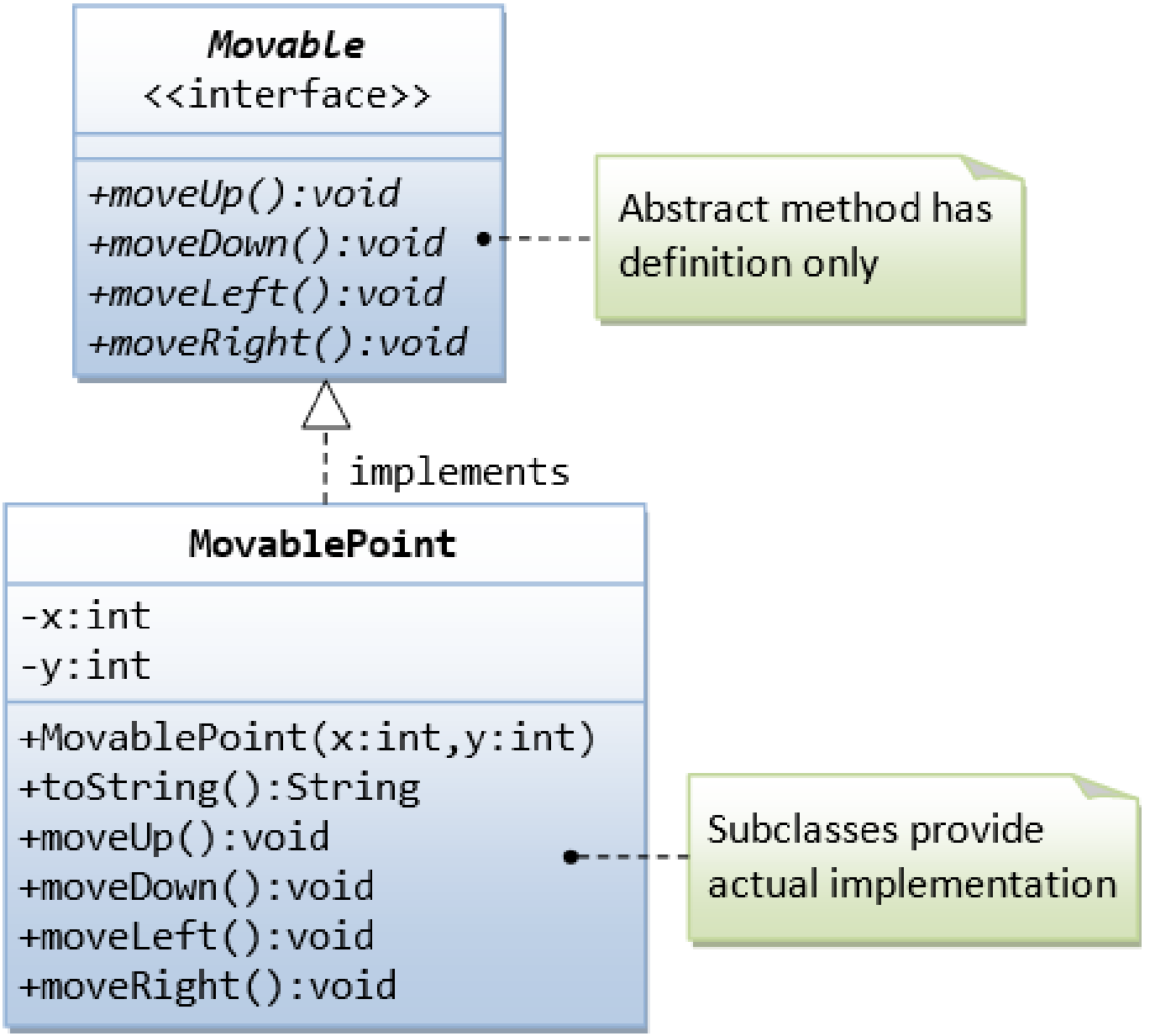
Interface - Example

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle(1, 2);  // upcast
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle(3, 4);  // upcast
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

      // Cannot create instance of an interface
      //Shape s3 = new Shape("green");   // Compilation Error!!
   }
}
```

Interface – Example 2



Interface – Example 2

```
/*
 * The Movable interface defines a list of public abstract methods
 * to be implemented by its subclasses
 */
public interface Movable {  // use keyword "interface" (instead of "class") to define an interface
   // An interface defines a list of abstract methods to be implemented by the subclasses
   public void moveUp();
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

Interface – Example 2

```
public class MovablePoint implements Movable {
   // Private member variables
   private int x, y;   // (x, y) coordinates of the point
   ….

   // Need to implement all the abstract methods defined in the interface Movable
   @Override
   public void moveUp() {
      y--;
   }
   @Override
   public void moveDown() {
      y++;
   }
   @Override
   public void moveLeft() {
      x--;
   }
   @Override
   public void moveRight() {
      x++;
   }
}
```

Interface – Example 2

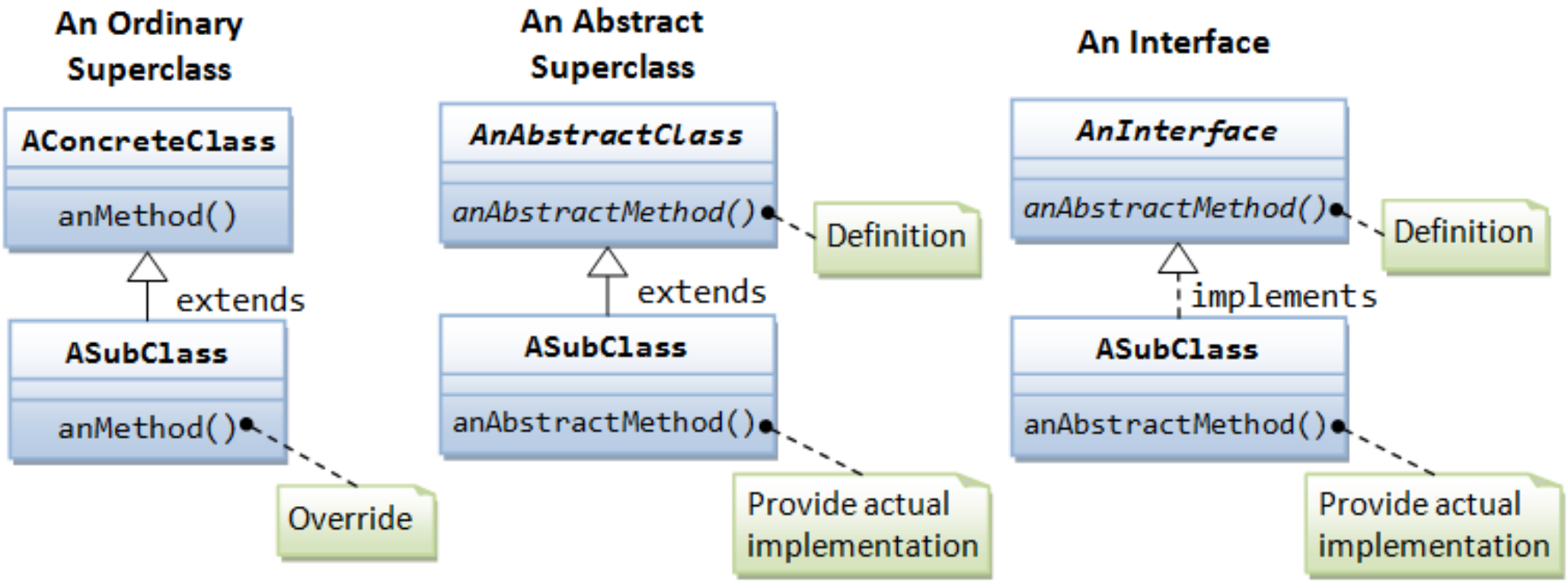
```
public class TestMovable {
   public static void main(String[] args) {
      MovablePoint p1 = new MovablePoint(1, 2);  // upcast
      System.out.println(p1);
      p1.moveDown();
      System.out.println(p1);
      p1.moveRight();
      System.out.println(p1);

      // Test Polymorphism
      Movable p2 = new MovablePoint(3, 4);  // upcast
      p2.moveUp();
      System.out.println(p2);
      MovablePoint p3 = (MovablePoint)p2;   // downcast
      System.out.println(p3);
   }
}
```

Implementing Multiple Interfaces

- Java supports only single inheritance.
- Java does not support multiple inheritance to avoid inheriting conflicting properties from multiple superclasses.
- A subclass, however, can implement more than one interfaces.
- An interface may "extends" from a super-interface.

UML Notations



"Is-a" vs. "has-a" relationships

- A subclass object processes all the data and methods from its superclass. We can say that a subclass object is-a superclass object.
- In composition, a class contains references to other classes, which is known as "has-a" relationship.

Reference

- https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html