VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

UNIVERSITY OF INFROMATION TECHNOLOGY

Chapter 3

JAVA OOP

Lecturer: MSc. Kiet Van Nguyen

Faculty of Information Science and Engineering

University of Information Technology, VNU-HCM

Today's Objectives

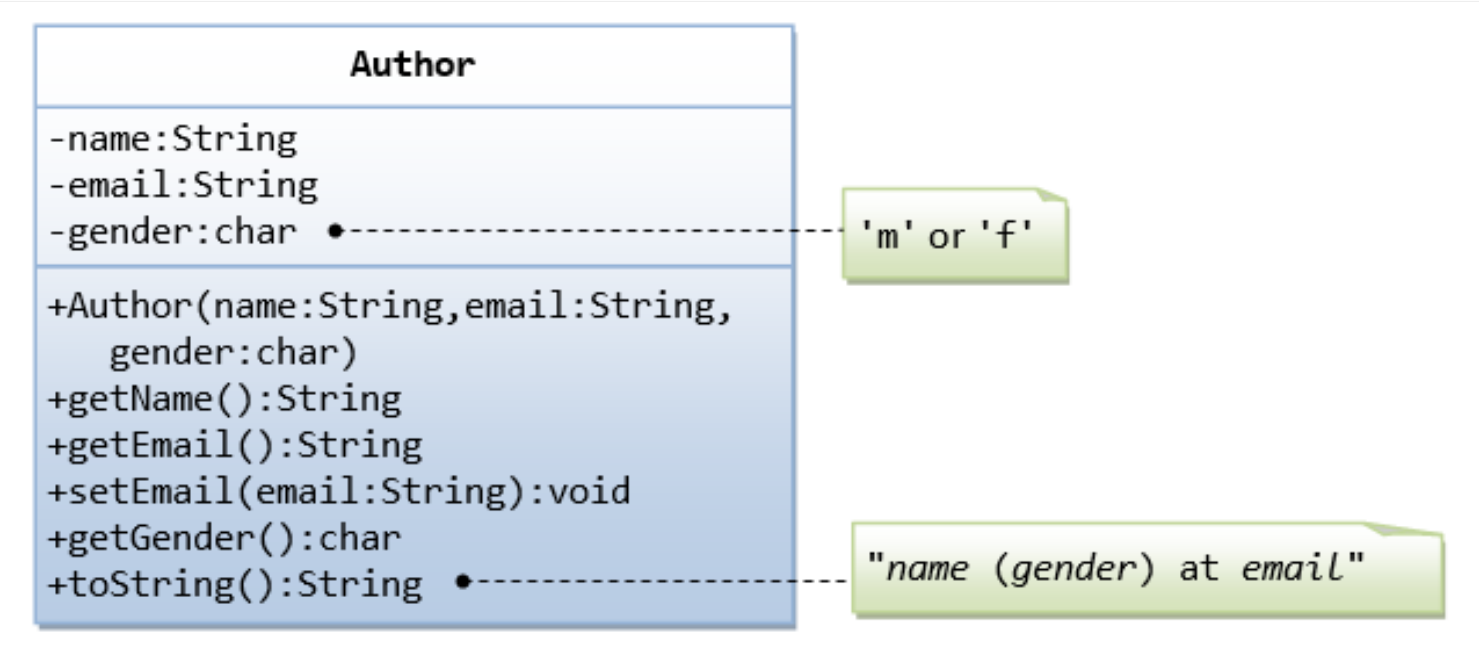
- Composition
- Inheritance
- Polymorphism

Today's Objectives

- There are two ways to ***reuse*** existing classes:
  - ✓ Composition
  - ✓ Inheritance

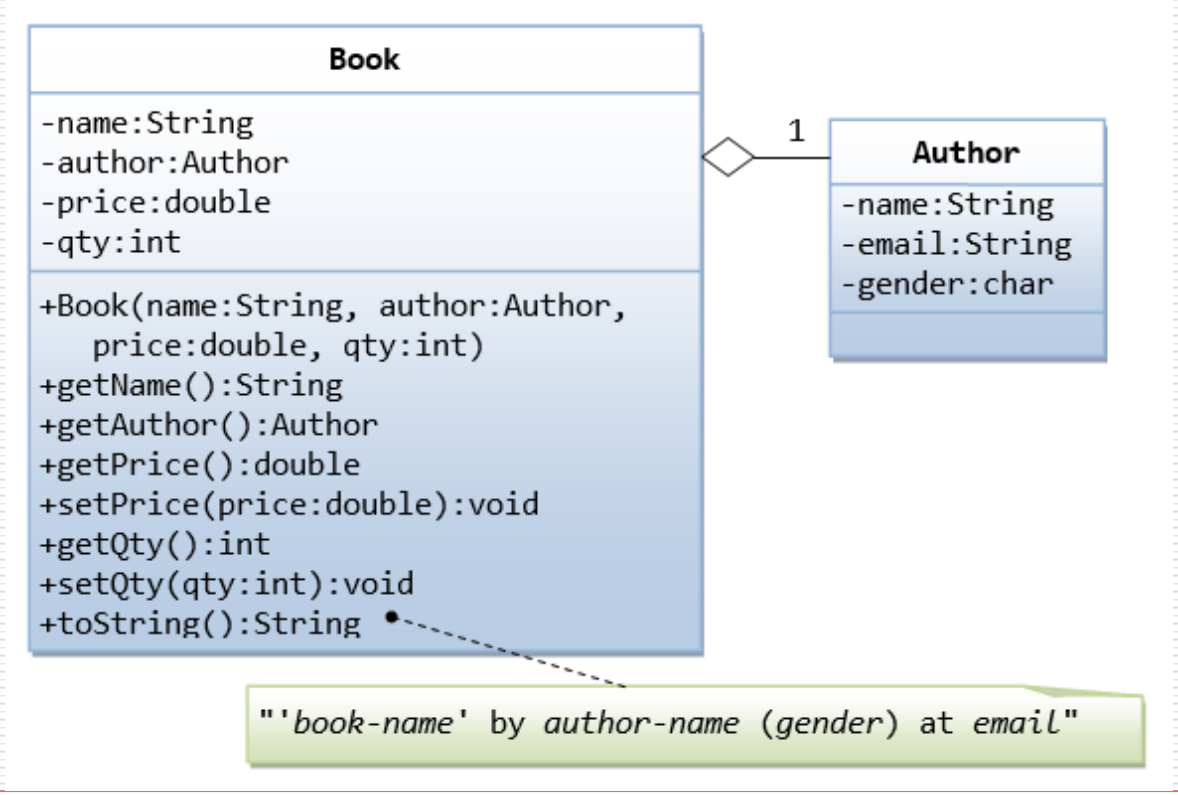
Composition

Example: The Author and Book Classes



Example: The Author and Book Classes (Cont.)

- A Book is written by one Author - Using an "Object" Member Variable



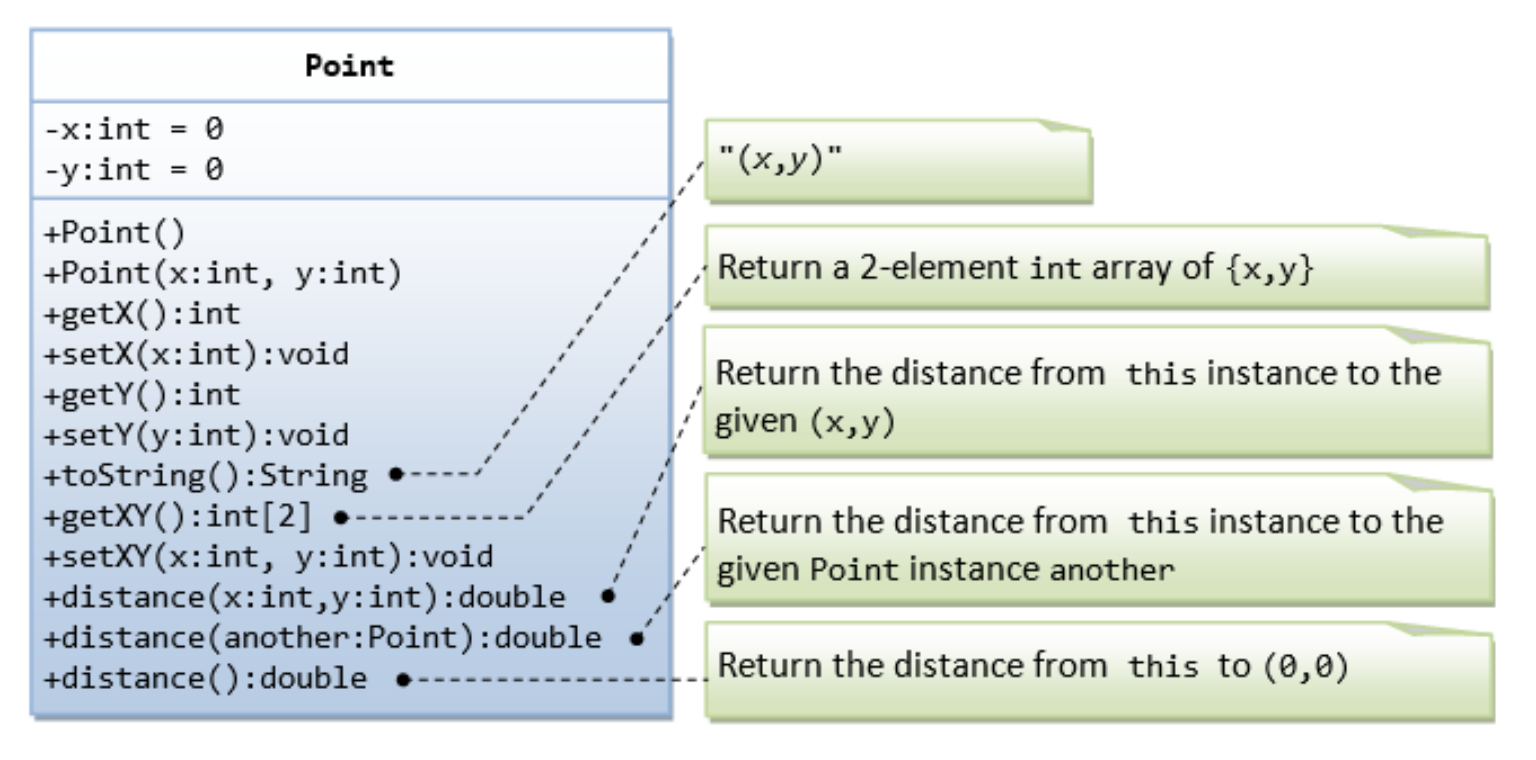
Example: The Author and Book Classes (Cont.)

```
1  /*
2   * The Author class model a book's author.
3   */
4  public class Author {
5     // The private instance variables
6     private String name;
7     private String email;
8     private char gender;   // 'm' or 'f'
9
10    // The constructor
11    public Author(String name, String email, char gender) {
12       this.name = name;
13       this.email = email;
14       this.gender = gender;
15    }
16
17    // The public getters and setters for the private instance variables.
```

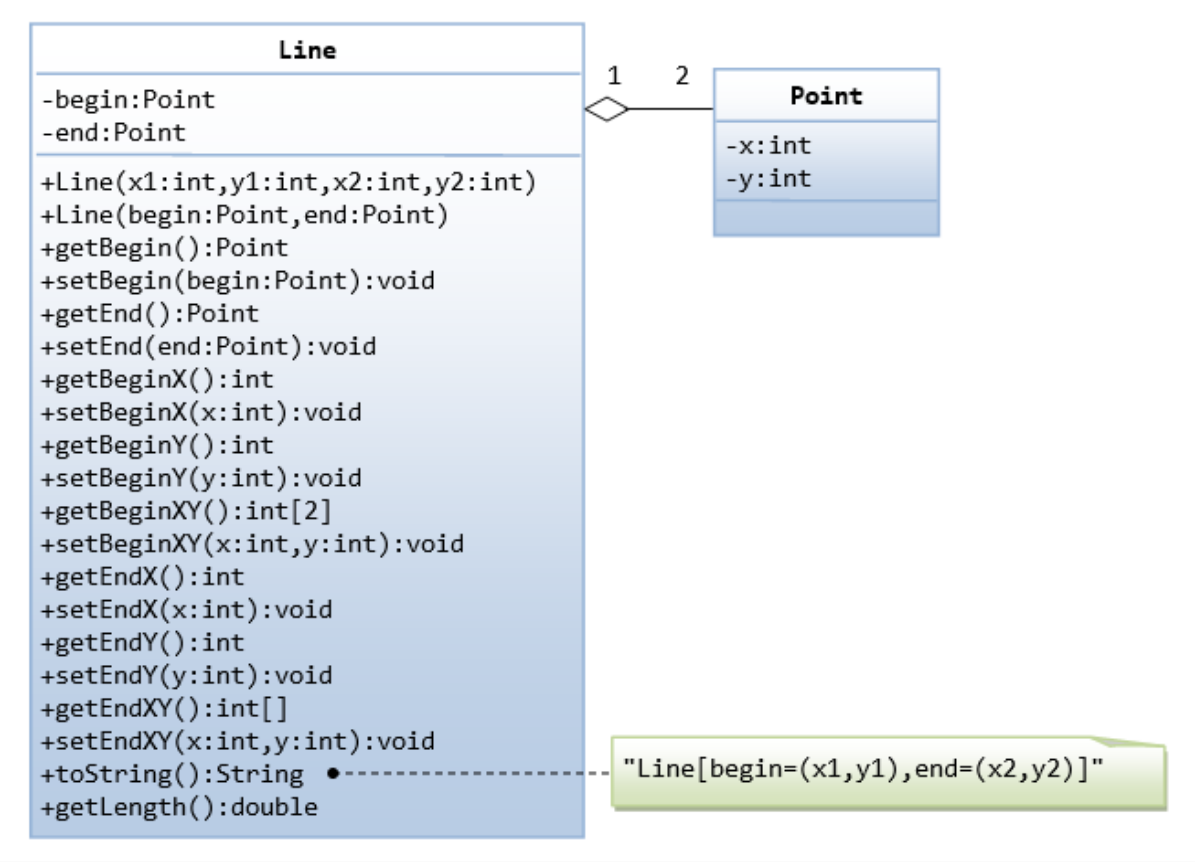
Example: The Author and Book Classes (Cont.)

```
1  /*
2   * The Book class models a book with one (and only one) author.
3   */
4  public class Book {
5     // The private instance variables
6     private String name;
7     private Author author;
8     private double price;
9     private int qty;
10
11    // Constructor
12    public Book(String name, Author author, double price, int qty) {
13       this.name = name;
14       this.author = author;
15       this.price = price;
16       this.qty = qty;
17    }
18
19    // Getters and Setters
```

Example: The Point and Line Classes



Example: The Point and Line Classes (Cont.)



Inheritance

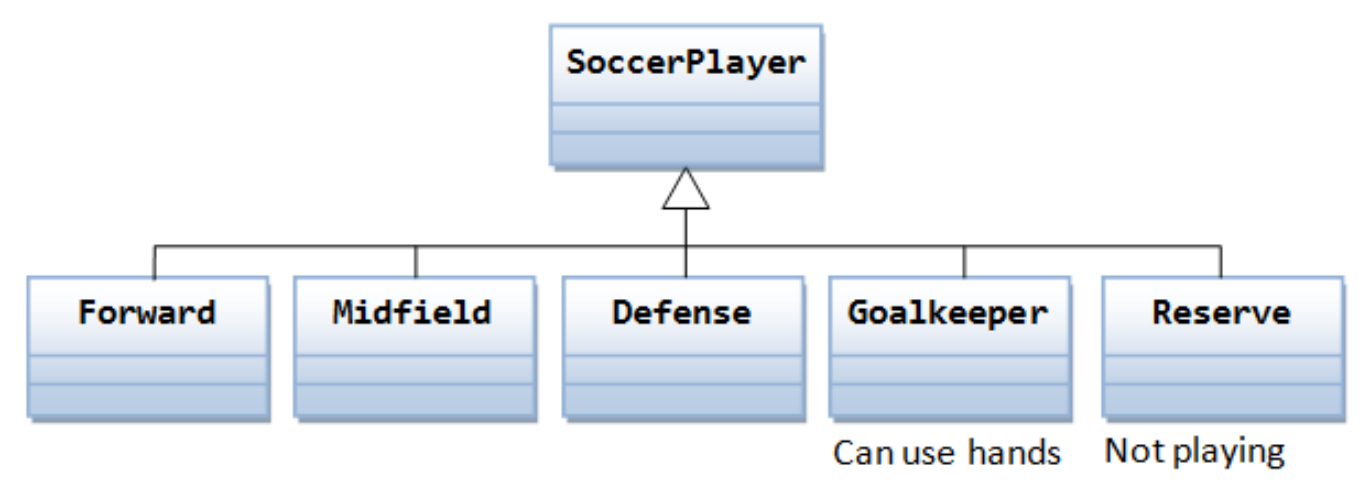
- Organize classes in *hierarchy* to *avoid duplication and reduce redundancy*.
- The classes in the lower hierarchy inherit all the variables and methods from the higher hierarchies.
- A class in the lower hierarchy is called a ***subclass*** (or *derived*, *child*, *extended class*). A class in the upper hierarchy is called a ***superclass*** (or *base*, *parent class*).

Inheritance (Cont.)

- **USE:** pulling out all the **common variables and methods** into the superclasses, and leave the specialized variables and methods in the subclasses.

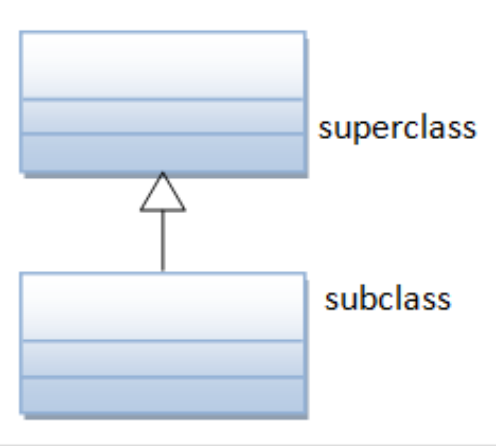
Inheritance (Cont.)

- **For example,**

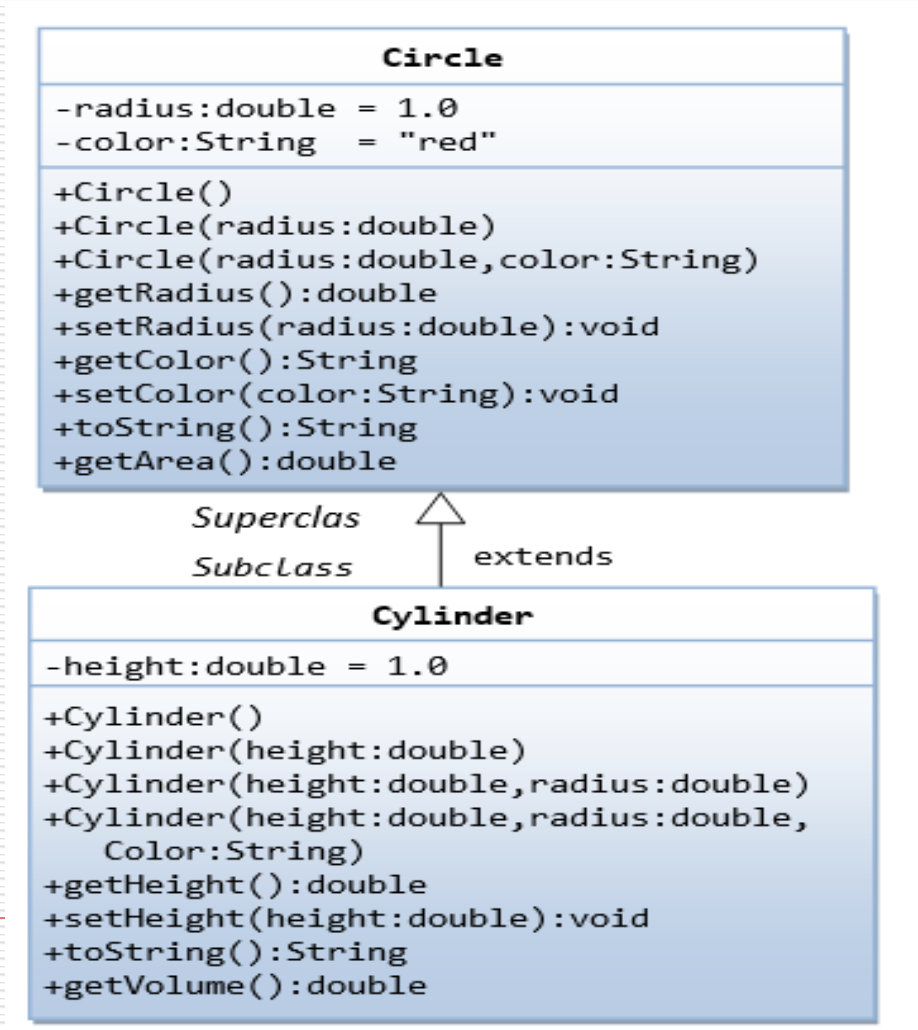


Inheritance (Cont.)

- You define a subclass using the keyword "extends", e.g.,
  - ✓ class Goalkeeper **extends** SoccerPlayer {.....}
  - ✓ class MyApplet **extends** java.applet.Applet {.....}
  - ✓ class Cylinder **extends** Circle {.....}



Example: The Circle and Cylinder Classes (Cont.)



The Circle and Cylinder Classes (Cont.)

Circle.java (Re-produced)

```
public class Circle {
   // private instance variables
   private double radius;
   private String color;

   // Constructors
   public Circle() {
      this.radius = 1.0;
      this.color = "red";
   }
   public Circle(double radius) {
      this.radius = radius;
      this.color = "red";
   }
   public Circle(double radius, String color) {
      this.radius = radius;
      this.color = color;
   }
```

The Circle and Cylinder Classes (Cont.)

```
   // Getters and Setters
   public double getRadius() {
      return this.radius;
   }
   public String getColor() {
      return this.color;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public void setColor(String color) {
      this.color = color;
   }

   // Describle itself
   public String toString() {
      return "Circle[radius=" + radius + ",color=" + color + "]";
   }

   // Return the area of this Circle
   public double getArea() {
      return radius * radius * Math.PI;
   }
}
```

The Circle and Cylinder Classes (Cont.)

Cylinder.java

```
1  /*
2   * A Cylinder is a Circle plus a height.
3   */
4  public class Cylinder extends Circle {
5     // private instance variable
6     private double height;
7
8     // Constructors
9     public Cylinder() {
10       super();  // invoke superclass' constructor Circle()
11       this.height = 1.0;
12    }
13    public Cylinder(double height) {
14       super();  // invoke superclass' constructor Circle()
15       this.height = height;
16    }
17    public Cylinder(double height, double radius) {
18       super(radius);  // invoke superclass' constructor Circle(radius)
19       this.height = height;
20    }
21    public Cylinder(double height, double radius, String color) {
22       super(radius, color);  // invoke superclass' constructor Circle(radius, color)
23       this.height = height;
24    }
```

The Circle and Cylinder Classes (Cont.)

```
26      // Getter and Setter
27      public double getHeight() {
28         return this.height;
29      }
30      public void setHeight(double height) {
31         this.height = height;
32      }
33
34      // Return the volume of this Cylinder
35      public double getVolume() {
36         return getArea()*height;   // Use Circle's getArea()
37      }
38
39      // Describible itself
40      public String toString() {
41         return "This is a Cylinder";  // to be refined later
42      }
43   }
```

The Circle and Cylinder Classes (Cont.)

A Test Drive for the `Cylinder` Class (`TestCylinder.java`)

```
1  /*
2   * A test driver for the Cylinder class.
3   */
4  public class TestCylinder {
5     public static void main(String[] args) {
6        Cylinder cy1 = new Cylinder();
7        System.out.println("Radius is " + cy1.getRadius()
8           + " Height is " + cy1.getHeight()
9           + " Color is " + cy1.getColor()
10          + " Base area is " + cy1.getArea()
11          + " Volume is " + cy1.getVolume());
12
13       Cylinder cy2 = new Cylinder(5.0, 2.0);
14       System.out.println("Radius is " + cy2.getRadius()
15          + " Height is " + cy2.getHeight()
16          + " Color is " + cy2.getColor()
17          + " Base area is " + cy2.getArea()
18          + " Volume is " + cy2.getVolume());
19    }
20 }
```

```
Radius is 1.0 Height is 1.0 Color is red Base area is 3.141592653589793 Volume is 3.141592653589793
Radius is 5.0 Height is 2.0 Color is red Base area is 78.53981633974483 Volume is 157.07963267948966
```

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 or hide an inherited variable by defining a variable of the same name.

Method Overriding & Variable Hiding (Cont.)

```
1  public class Cylinder extends Circle {
2     ......
3     // Override the getArea() method inherited from superclass Circle
4     @Override
5     public double getArea() {
6        return 2*Math.PI*getRadius()*height + 2*super.getArea();
7     }
8     // Need to change the getVolume() as well
9     public double getVolume() {
10       return super.getArea()*height;   // use superclass' getArea()
11    }
12    // Override the inherited toString()
13    @Override
14    public String toString() {
15       return "Cylinder[" + super.toString() + ",height=" + height + "]";
16    }
17 }
```

Keyword "super"

- Use the keyword **this** to refer to *this* ***instance***
- The keyword **super** refers to the **superclass**

More on Constructors

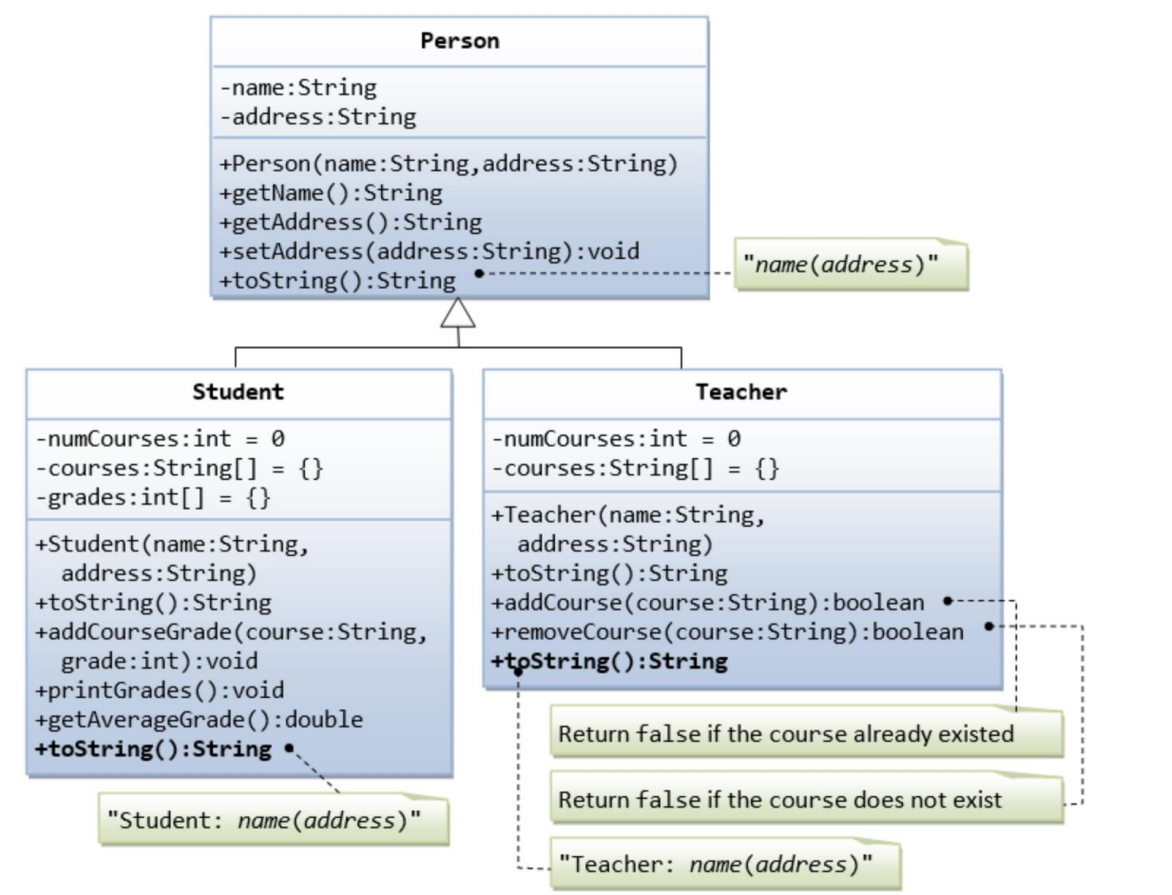
- super() statement to invoke the no-arg constructor of its immediate superclass
- Use super(*args*) to invoke a constructor of its immediate superclass
- Note:

- If no constructor is defined in a class, Java compiler automatically create a *no-argument (no-arg) constructor*.
- If the immediate superclass does not have the default constructor (it defines some constructors but does not define a no-arg constructor), you will get a compilation error in doing a super() call

Single Inheritance

- Java does not support multiple inheritance (C++ does). Multiple inheritance permits a subclass to have more than one direct superclasses.
- Drawback of multiple inheritance???

Example: Superclass Person and its Subclasses



The Superclass Person.java

```
/*
 * Superclass Person has name and address.
 */
public class Person {
   // private instance variables
   private String name, address;

   // Constructor
   public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   // Getters and Setters
   public String getName() {
      return name;
   }
   public String getAddress() {
      return address;
   }
   public void setAddress(String address) {
      this.address = address;
   }

   // Describle itself
   public String toString() {
      return name + "(" + address + ")";
   }
}
```

The Subclass Student.java

```
/*
 * The Student class, subclass of Person.
 */
public class Student extends Person {
   // private instance variables
   private int numCourses;    // number of courses taken so far
   private String[] courses; // course codes
   private int[] grades;      // grade for the corresponding course codes
   private static final int MAX_COURSES = 30; // maximum number of courses

   // Constructor
   public Student(String name, String address) {
      super(name, address);
      numCourses = 0;
      courses = new String[MAX_COURSES];
      grades = new int[MAX_COURSES];
   }

   // Describe itself
   @Override
   public String toString() {
      return "Student: " + super.toString();
   }

   // Add a course and its grade - No validation in this method
   public void addCourseGrade(String course, int grade) {
      courses[numCourses] = course;
      grades[numCourses] = grade;
      ++numCourses;
   }
```

The Subclass Student.java

```
   // Print all courses taken and their grade
   public void printGrades() {
      System.out.print(this);
      for (int i = 0; i < numCourses; ++i) {
         System.out.print(" " + courses[i] + ":" + grades[i]);
      }
      System.out.println();
   }

   // Compute the average grade
   public double getAverageGrade() {
      int sum = 0;
      for (int i = 0; i < numCourses; i++ ) {
         sum += grades[i];
      }
      return (double)sum/numCourses;
   }
}
```

The Subclass Teacher.java

```
/*
 * The Teacher class, subclass of Person.
 */
public class Teacher extends Person {
   // private instance variables
   private int numCourses;   // number of courses taught currently
   private String[] courses; // course codes
   private static final int MAX_COURSES = 5; // maximum courses

   // Constructor
   public Teacher(String name, String address) {
      super(name, address);
      numCourses = 0;
      courses = new String[MAX_COURSES];
   }

   // Describe itself
   @Override
   public String toString() {
      return "Teacher: " + super.toString();
   }

   // Return false if the course already existed
   public boolean addCourse(String course) {
      // Check if the course already in the course list
      for (int i = 0; i < numCourses; i++) {
         if (courses[i].equals(course)) return false;
      }
      courses[numCourses] = course;
      numCourses++;
      return true;
   }
```

The Subclass Teacher.java

```
   // Return false if the course cannot be found in the course list
   public boolean removeCourse(String course) {
      boolean found = false;
      // Look for the course index
      int courseIndex = -1;  // need to initialize
      for (int i = 0; i < numCourses; i++) {
         if (courses[i].equals(course)) {
            courseIndex = i;
            found = true;
            break;
         }
      }
      if (found) {
         // Remove the course and re-arrange for courses array
         for (int i = courseIndex; i < numCourses-1; i++) {
            courses[i] = courses[i+1];
         }
         numCourses--;
         return true;
      } else {
         return false;
      }
   }
}
```

A Test Driver (TestPerson.java)

```
/*
 * A test driver for Person and its subclasses.
 */
public class TestPerson {
   public static void main(String[] args) {
      /* Test Student class */
      Student s1 = new Student("Tan Ah Teck", "1 Happy Ave");
      s1.addCourseGrade("IM101", 97);
      s1.addCourseGrade("IM102", 68);
      s1.printGrades();
      System.out.println("Average is " + s1.getAverageGrade());

      /* Test Teacher class */
      Teacher t1 = new Teacher("Paul Tan", "8 sunset way");
      System.out.println(t1);
      String[] courses = {"IM101", "IM102", "IM101"};
      for (String course: courses) {
         if (t1.addCourse(course)) {
            System.out.println(course + " added.");
         } else {
            System.out.println(course + " cannot be added.");
         }
      }
      for (String course: courses) {
         if (t1.removeCourse(course)) {
            System.out.println(course + " removed.");
         } else {
            System.out.println(course + " cannot be removed.");
         }
      }
   }
}
```

Output

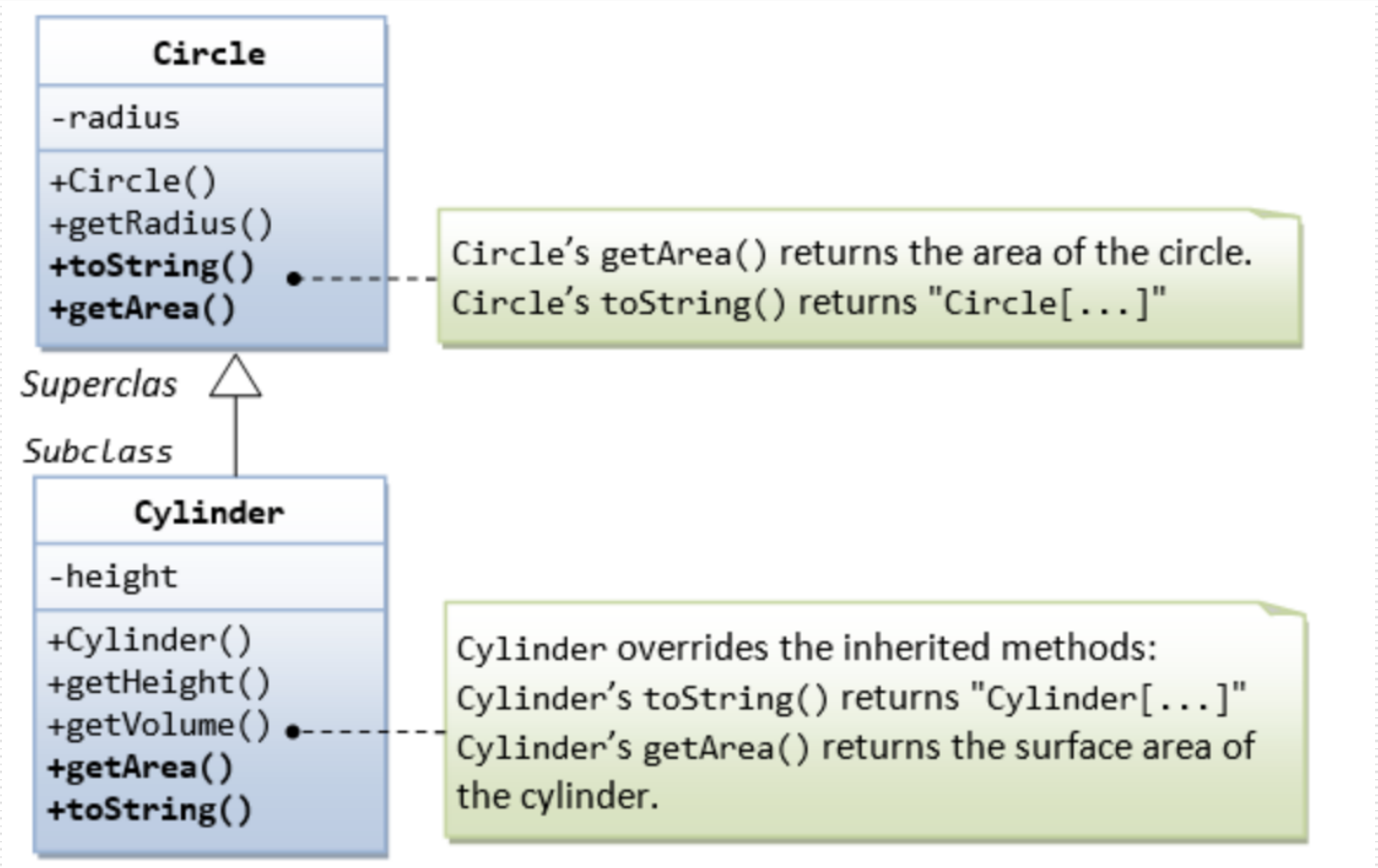
```
Tan Ah Teck(1 Happy Ave)
Tan Ah Teck
8 Sunrise Place
Student: Mohd Ali(8 Kg Java)
Mohd Ali
9 Kg Satu
Student: Mohd Ali(9 Kg Satu) IM101:97 IM102:68
Average is: 82.5
Teacher: Paul Tan(8 sunset way)
IM101 added.
IM102 added.
IM101 cannot be added.
IM101 removed.
IM102 removed.
IM101 cannot be removed.
```

Rule of Thumb

- Use composition if possible, before considering inheritance. Use inheritance only if there is a clear hierarchical relationship between classes.

Polymorphism

Substitutability



Substitutability (Cont.)

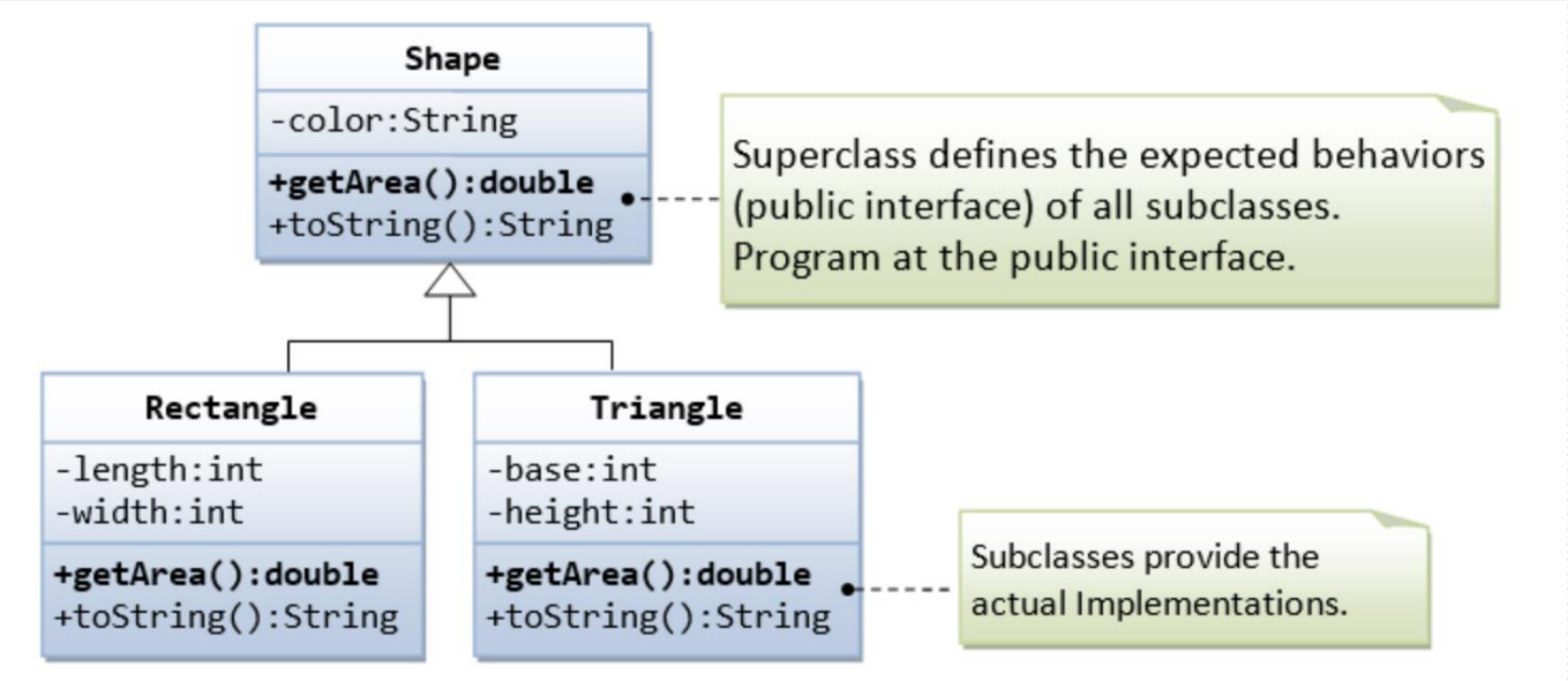
```
// Substitute a subclass instance to a superclass reference
Circle c1 = new Cylinder(1.1, 2.2);
```

```
// Invoke superclass Circle's methods
c1.getRadius();
```

```
// CANNOT invoke method in Cylinder as it is a Circle reference!
c1.getHeight();  // compilation error
c1.getVolume();  // compilation error
```

```
c1.toString();  // Run the overridden version!
c1.getArea();   // Run the overridden version!
```

Example: Shape and its Subclasses



The Superclass Shape.java

```
/*
 * Superclass Shape maintain the common properties of all shapes
 */
public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape[color=" + color + "]";
   }

   // All shapes must have a method called getArea().
   public double getArea() {
      // We have a problem here!
      // We need to return some value to compile the program.
      System.err.println("Shape unknown! Cannot compute area!");
      return 0;
   }
}
```

The Subclass Rectangle.java

```
/*
 * The Rectangle class, subclass of Shape
 */
public class Rectangle extends Shape {
   // Private member variables
   private int length;
   private int width;

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

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

   // Override the inherited getArea() to provide the proper implementation
   @Override
   public double getArea() {
      return length*width;
   }
}
```

The Subclass Triangle.java

```
/*
 * The Triangle class, subclass of Shape
 */
public class Triangle extends Shape {
   // Private member variables
   private int base;
   private int height;

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

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

   // Override the inherited getArea() to provide the proper implementation
   @Override
   public double getArea() {
      return 0.5*base*height;
   }
}
```

A Test Driver (TestShape.java)

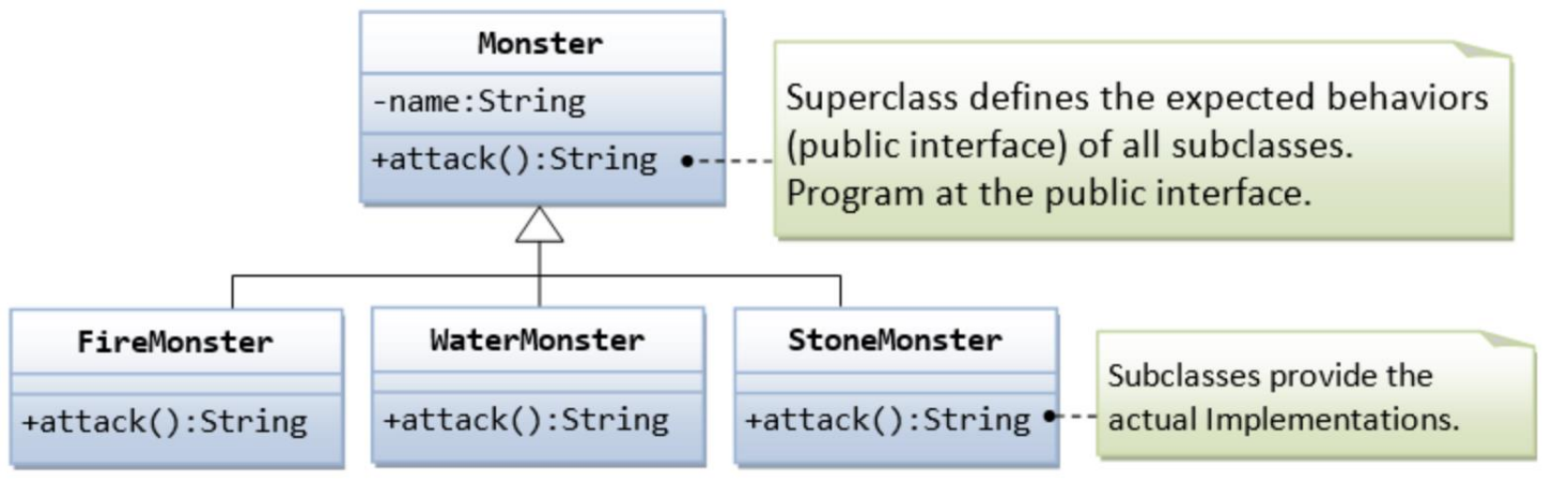
```
/*
 * A test driver for Shape and its subclasses
 */
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);  // Upcast
      System.out.println(s1);  // Run Rectangle's toString()
      System.out.println("Area is " + s1.getArea());  // Run Rectangle's getArea()

      Shape s2 = new Triangle("blue", 4, 5);  // Upcast
      System.out.println(s2);  // Run Triangle's toString()
      System.out.println("Area is " + s2.getArea());  // Run Triangle's getArea()
   }
}
```

The expected outputs are:

```
Rectangle[length=4,width=5,Shape[color=red]]
Area is 20.0
Triangle[base=4,height=5,Shape[color=blue]]
Area is 10.0
```

```
public class TestShape {
   public static void main(String[] args) {
      // Constructing a Shape instance poses problem!
      Shape s3 = new Shape("green");
      System.out.println(s3);
      System.out.println("Area is " + s3.getArea());  // Invalid output
   }
}
```

Example: Monster and its Subclasses



Superclass Monster.java

```
/*
 * The superclass Monster defines the expected common behaviors for its subclasses.
 */
public class Monster {
   // private instance variable
   private String name;

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

   // Define common behavior for all its subclasses
   public String attack() {
      return "!^_&^$@+%$* I don't know how to attack!";
      // We have a problem here!
      // We need to return a String; else, compilation error!
   }
}
```

Subclass FireMonster.java

```
public class FireMonster extends Monster {
   // Constructor
   public FireMonster(String name) {
      super(name);
   }
   // Subclass provides actual implementation
   @Override public String attack() {
      return "Attack with fire!";
   }
}
```

Subclass WaterMonster.java

```
public class WaterMonster extends Monster {
   // Constructor
   public WaterMonster(String name) {
      super(name);
   }
   // Subclass provides actual implementation
   @Override public String attack() {
      return "Attack with water!";
   }
}
```

Subclass StoneMonster.java

```
public class StoneMonster extends Monster {
   // Constructor
   public StoneMonster(String name) {
      super(name);
   }
   // Subclass provides actual implementation
   @Override public String attack() {
      return "Attack with stones!";
   }
}
```

A Test Driver TestMonster.java

```
public class TestMonster {
   public static void main(String[] args) {
      // Program at the "interface" defined in the superclass.
      // Declare instances of the superclass, substituted by subclasses.
      Monster m1 = new FireMonster("r2u2");   // upcast
      Monster m2 = new WaterMonster("u2r2");  // upcast
      Monster m3 = new StoneMonster("r2r2");  // upcast

      // Invoke the actual implementation
      System.out.println(m1.attack());  // Run FireMonster's attack()
      System.out.println(m2.attack());  // Run WaterMonster's attack()
      System.out.println(m3.attack());  // Run StoneMonster's attack()

      // m1 dies, generate a new instance and re-assign to m1.
      m1 = new StoneMonster("a2b2");  // upcast
      System.out.println(m1.attack());  // Run StoneMonster's attack()

      // We have a problem here!!!
      Monster m4 = new Monster("u2u2");
      System.out.println(m4.attack());  // garbage!!!
   }
}
```

The abstract Method and abstract class

- 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.

Example: Shape and its Subclasses

```
/*
abstract public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }

   // All Shape subclasses must implement a method called getArea()
   abstract public double getArea();
}
```

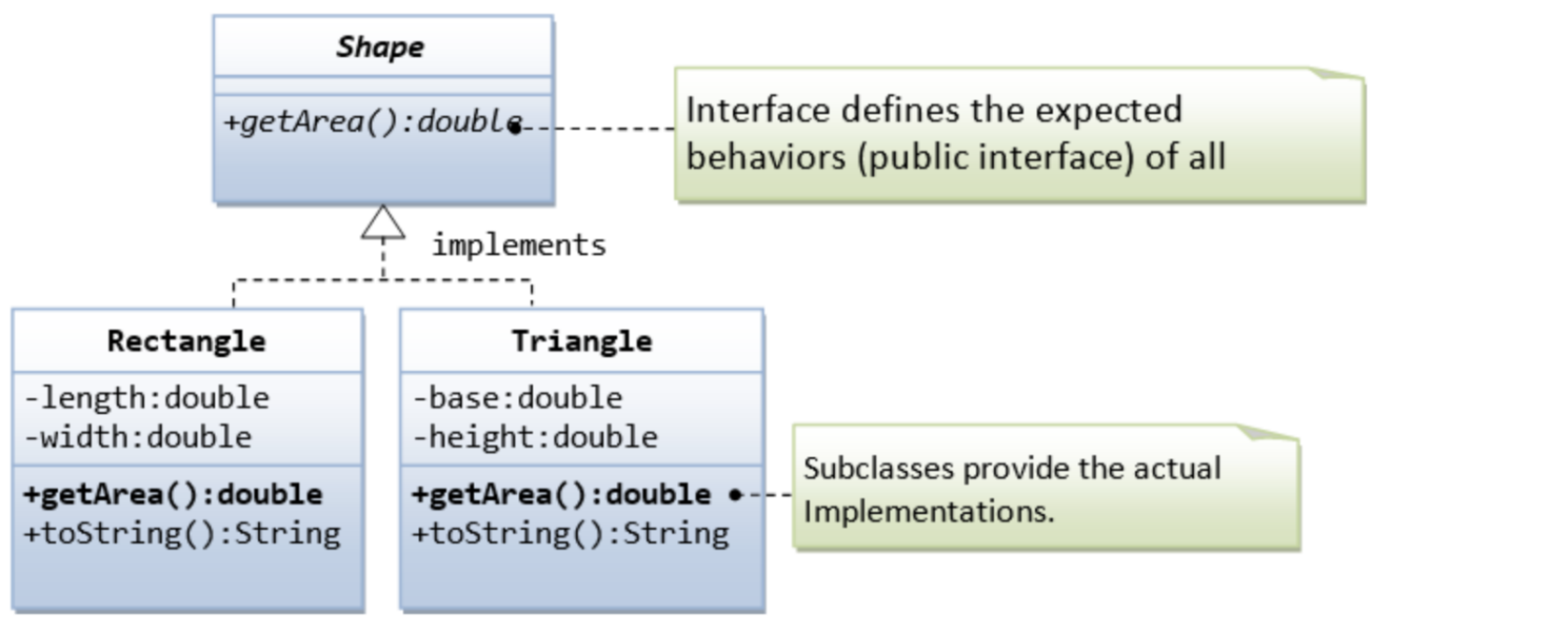
Example: Shape and its Subclasses (Cont.)

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

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

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* (methods with signature and no implementation) and possibly *constants* (public static final variables)
- Similar to an abstract superclass, an interface cannot be instantiated

Example: Shape Interface and its Implementations



Example: Shape Interface and its Implementations (Cont.)

```
/*
 * 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
   // All methods in interface are "public abstract".
   // "protected", "private" and "package" methods are NOT allowed.
   double getArea();
}
```

Example: Shape Interface and its Implementations (Cont.)

```
// 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;
   }
}
```

Example: Shape Interface and its Implementations (Cont.)

```
// 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;
   }
}
```

Example: Shape Interface and its Implementations (Cont.)

```
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!!
   }
}
```

Implementing Multiple Interfaces

```
public class Circle extends Shape implements Movable, Adjustable {
          // extends one superclass but implements multiple interfaces
   .......
}
```

More thinking …

- Why interfaces?
- Interface vs. Abstract Superclass?

Q&A

Thank you!