

Tutorial: Composition, Inheritance, Polymorphism & Abstraction

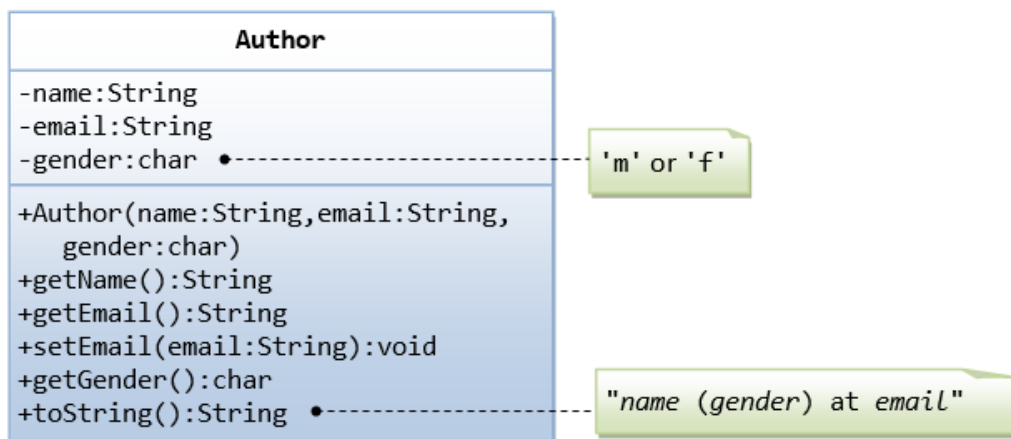
There are two ways to reuse existing classes, namely, composition and inheritance. With composition (aka aggregation), you define a new class, which is composed of existing classes. With inheritance, you derive a new class based on an existing class, with modifications or extensions.

1 Composition

We shall begin with reusing classes via composition - through examples.

1.1 Composition EG. 1: The Author and Book Classes

Let's start with the Author class



A class called Author is designed as shown in the class diagram. It contains:

- Three private member variables: name (String), email (String), and gender (char of either 'm' or 'f' - you might also use a boolean variable called isMale having value of true or false).
- A constructor to initialize the name, email and gender with the given values. (There is no default constructor, as there is no default value for name, email and gender.)
- Public getters/setters: getName(), getEmail(), setEmail(), and getGender(). (There are no setters for name and gender, as these properties are not designed to be changed.)

- A toString() method that returns "name (gender) at email", e.g., "Tan Ah Teck (m) at ahTeck@somewhere.com".

The Author Class (Author.java)



```
1  /**
   * The Author class model a book's author.
   */
3  public class Author {
4      // The private instance variables
5      private String name;
6      private String email;
7      private char gender;    // 'm' or 'f'
8
9      /** Constructs a Author instance with the given inputs */
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.
18     // No setter for name and gender as they are not designed to be changed.
19     /** Returns the name */
20     public String getName() {
21         return name;
22     }
23
24     /** Returns the gender */
25     public char getGender() {
26         return gender;
27     }
28
29     /** Returns the email */
30     public String getEmail() {
31         return email;
32     }
33
34     /** Sets the email */
35     public void setEmail(String email) {
36         this.email = email;
37     }
38
39     /** Returns a self-descriptive String */
40     public String toString() {
41         return name + " (" + gender + ") at " + email;
42     }
43 }
```

A Test Driver for the Author Class (TestAuthor.java)

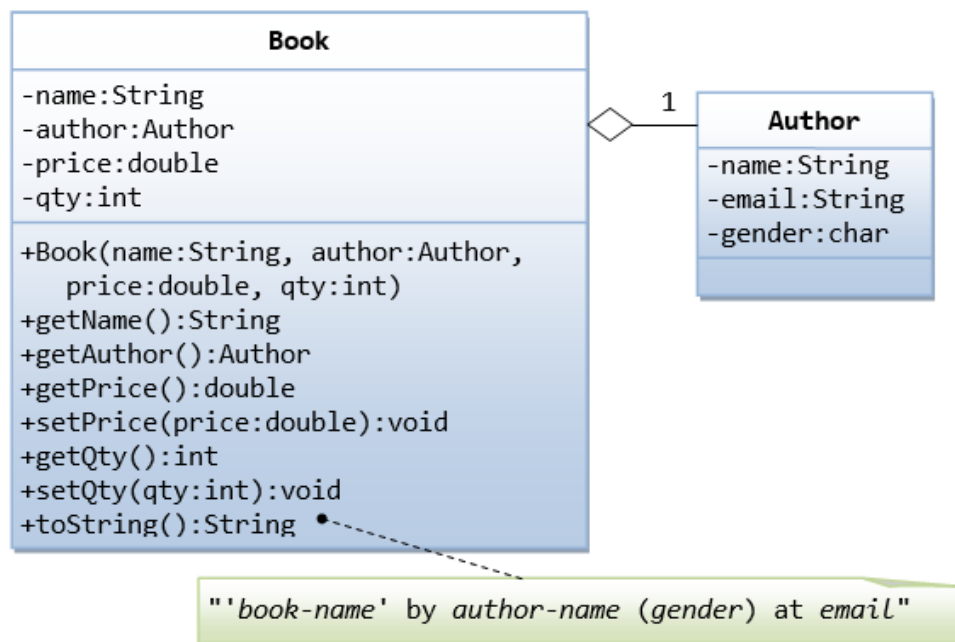


```

1  /**
   * A test driver for the Author class.
   */
3  public class TestAuthor {
4      public static void main(String[] args) {
5          // Test constructor and toString()
6          Author ahTeck = new Author("Tan Ah Teck", "teck@nowhere.com", 'm');
7          System.out.println(ahTeck); // toString()
8          // Tan Ah Teck (m) at teck@nowhere.com
9
10         // Test Setters and Getters
11         ahTeck.setEmail("teck@somewhere.com");
12         System.out.println(ahTeck); // toString()
13         // Tan Ah Teck (m) at teck@somewhere.com
14         System.out.println("name is: " + ahTeck.getName());
15         // name is: Tan Ah Teck
16         System.out.println("gender is: " + ahTeck.getGender());
17         // gender is: m
18         System.out.println("email is: " + ahTeck.getEmail());
19         // email is: teck@somewhere.com
20     }
21 }

```

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



Let's design a Book class. Assume that a book is written by one (and exactly one) author. The Book class (as shown in the class diagram) contains the following members:

- Four private member variables: name (String), author (an instance of the Author class we have just created, assuming that each book has exactly one author), price (double), and qty (int).
- The public getters and setters: getName(), getAuthor(), getPrice(), setPrice(), getQty(), setQty().
- A toString() that returns "'book-name' by author-name (gender) at email". You could reuse the Author's toString() method, which returns "author-name (gender) at email".

The Book Class (Book.java)



```
/**
2  * The Book class models a book with one (and only one) author.
   */
4  public class Book {
   // The private instance variables
6     private String name;
   private Author author;
8     private double price;
   private int qty;

10
   /** Constructs a Book instance with the given author */
12  public Book(String name, Author author, double price, int qty) {
14     this.name = name;
16     this.author = author;
18     this.price = price;
20     this.qty = qty;
22  }

24
   // Getters and Setters
   /** Returns the name of this book */
26  public String getName() {
28     return name;
30  }

   /** Return the Author instance of this book */
32  public Author getAuthor() {
34     return author; // return member author, which is an instance of the class Author
36  }

   /** Returns the price */
38  public double getPrice() {
40     return price;
42  }

   /** Sets the price */
```



```

36     public void setPrice(double price) {
37         this.price = price;
38     }

40     /** Returns the quantity */
41     public int getQty() {
42         return qty;
43     }

44     /** Sets the quantity */
45     public void setQty(int qty) {
46         this.qty = qty;
47     }

50     /** Returns a self-descriptive String */
51     public String toString() {
52         return "\"" + name + "\" by " + author; // author.toString()
53     }
54 }

```

A Test Driver Program for the Book Class (TestBook.java)



```

/**
2  * A test driver program for the Book class.
3  */
4  public class TestBook {
5      public static void main(String[] args) {
6          // We need an Author instance to create a Book instance
7          Author ahTeck = new Author("Tan Ah Teck", "ahTeck@somewhere.com", 'm');
8          System.out.println(ahTeck); // Author's toString()
9          // Tan Ah Teck (m) at ahTeck@somewhere.com

10         // Test Book's constructor and toString()
11         Book dummyBook = new Book("Java for dummies", ahTeck, 9.99, 99);
12         System.out.println(dummyBook); // Book's toString()
13         // 'Java for dummies' by Tan Ah Teck (m) at ahTeck@somewhere.com

14         // Test Setters and Getters
15         dummyBook.setPrice(8.88);
16         dummyBook.setQty(88);
17         System.out.println("name is: " + dummyBook.getName());
18         // name is: Java for dummies
19         System.out.println("price is: " + dummyBook.getPrice());
20         // price is: 8.88
21         System.out.println("qty is: " + dummyBook.getQty());
22         // qty is: 88
23         System.out.println("author is: " + dummyBook.getAuthor()); //

```



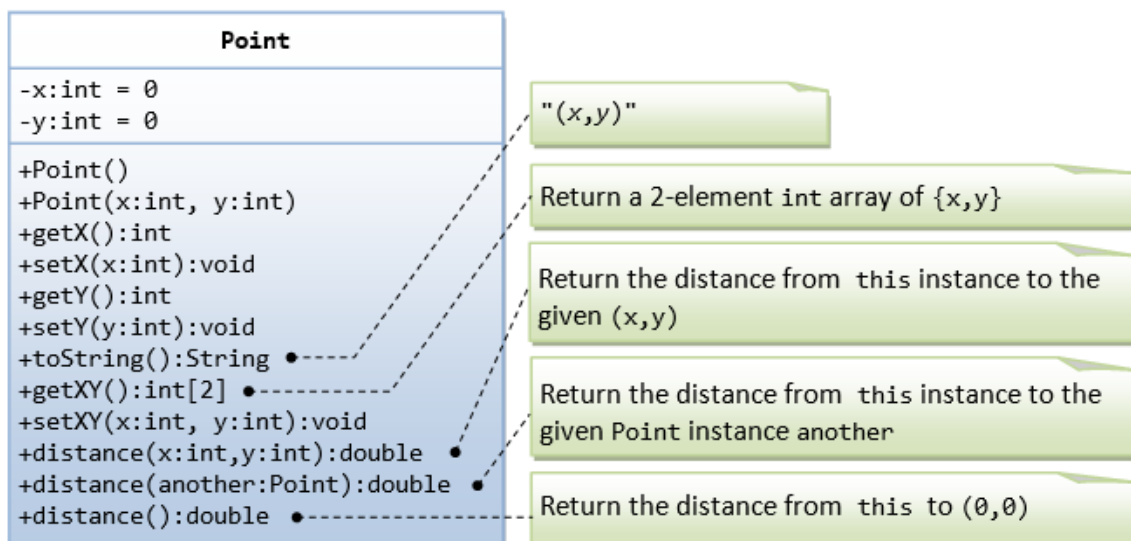
```

    ↪ invoke Author's toString()
26 // author is: Tan Ah Teck (m) at ahTeck@somewhere.com
    System.out.println("author's name is: "
28     + dummyBook.getAuthor().getName());
    // author's name is: Tan Ah Teck
30 System.out.println("author's email is: "
    + dummyBook.getAuthor().getEmail());
32 // author's email is: ahTeck@somewhere.com
    System.out.println("author's gender is: "
34     + dummyBook.getAuthor().getGender());
    // author's gender is: m
36
    // Using an anonymous Author instance to create a Book instance
38 Book moreDummyBook = new Book("Java for more dummies",
    new Author("Peter Lee", "peter@nowhere.com", 'm'), // an
        ↪ anonymous Author's instance
40     19.99, 8);
    System.out.println(moreDummyBook); // Book's toString()
42 // 'Java for more dummies' by Peter Lee (m) at peter@nowhere.com
    }
44 }

```

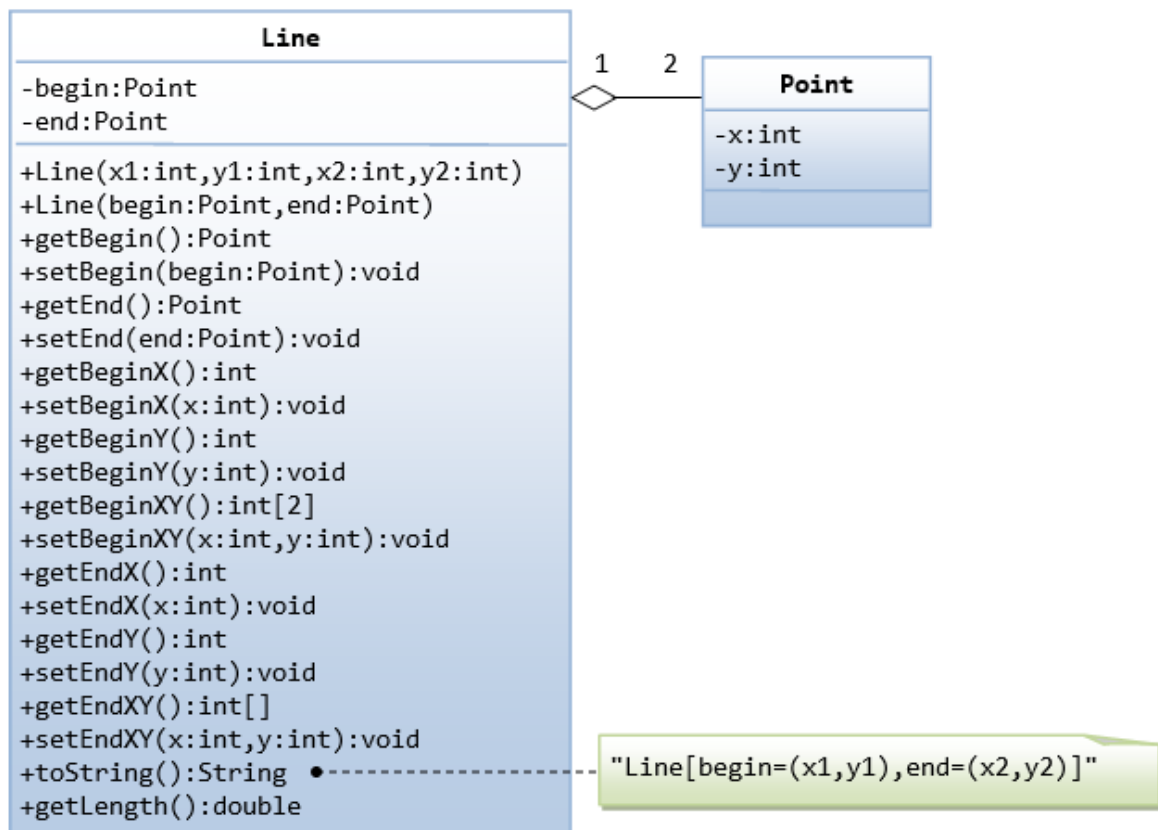
Notes: In this example, I used "name" for Book class instead of "title" to illustrate that you can have a variable name in both the Author and Book classes, but they are distinct.

1.2 Composition EG. 2: The Point and Line Classes



As an example of reusing a class via composition, suppose that we have an existing class called `Point`, defined as shown in the above class diagram.

Suppose that we need a new class called `Line`, we can design the `Line` class by re-using the `Point` class via *composition*. We say that "A line is *composed* of two points", or "A line *has* two points". Composition exhibits a "*has-a*" relationship.



UML Notation: In UML notations, composition is represented as a diamond-head line pointing to its constituents.

The Line Class via Composition (Line.java)

```

/**
 * A Line composes of two Points – a begin "Point" and an end "Point".
 * We reuse the Point class via composition.
 */
public class Line {
    // The private instance variables
    Point begin;    // instances of the "Point" class
    Point end;
}
  
```



```
10  /**
11  * Constructs a Line instance given 2 points at (x1, y1) and (x2, y2)
12  */
13  public Line(int x1, int y1, int x2, int y2) {
14      begin = new Point(x1, y1); // Construct the instances declared
15      end = new Point(x2, y2);
16  }
17
18  /** Construct a Line instance given 2 Point instances */
19  public Line(Point begin, Point end) {
20      this.begin = begin; // The caller had constructed the instances
21      this.end = end;
22  }
23
24  // The public getter and setter for the private instance variables
25  public Point getBegin() {
26      return begin;
27  }
28
29  public Point getEnd() {
30      return end;
31  }
32
33  public void setBegin(Point begin) {
34      this.begin = begin;
35  }
36
37  public void setEnd(Point end) {
38      this.end = end;
39  }
40
41  public int getBeginX() {
42      return begin.getX(); // Point's getX()
43  }
44
45  public void setBeginX(int x) {
46      begin.setX(x); // Point's setX()
47  }
48
49  public int getBeginY() {
50      return begin.getY(); // Point's getY()
51  }
52
53  public void setBeginY(int y) {
54      begin.setY(y); // Point's setY()
55  }
56
57  public int[] getBeginXY() {
58      return begin.getXY(); // Point's getXY()
59  }
60
```




```

62     public void setBeginXY(int x, int y) {
        begin.setXY(x, y); // Point's setXY()
    }
64
    public int getEndX() {
66         return end.getX(); // Point's getX()
    }
68
    public void setEndX(int x) {
70         end.setX(x); // Point's setX()
    }
72
    public int getEndY() {
74         return end.getY(); // Point's getY()
    }
76
    public void setEndY(int y) {
78         end.setY(y); // Point's setY()
    }
80
    public int[] getEndXY() {
82         return end.getXY(); // Point's getXY()
    }
84
    public void setEndXY(int x, int y) {
86         end.setXY(x, y); // Point's setXY()
    }
88
    /** Returns a self-descriptive String */
90     public String toString() {
        return "Line[begin=" + begin + ",end=" + end + "];
92         // Invoke begin.toString() and end.toString()
    }
94
    /** Returns the length of this line */
96     public double getLength() {
        return begin.distance(end); // use Point's distance() method
98     }
    }

```

A Test Driver for Line Class (TestLine.java)



```

1  import java.util.Arrays;
   /**
3   * A Test Driver for the Line class.
   */

```



```

5 public class TestLine {
    public static void main(String[] args) {
6         // Test constructor and toString()
7         Line l1 = new Line(1, 2, 3, 4);
8         System.out.println(l1); // Line's toString()
9         // Line[begin=(1,2),end=(3,4)]
11        Line l2 = new Line(new Point(5,6), new Point(7,8)); // anonymous
            ↪ Point's instances
12        System.out.println(l2); // Line's toString()
13        // Line[begin=(5,6),end=(7,8)]

14
15        // Test Setters and Getters
16        l1.setBegin(new Point(11, 12));
17        l1.setEnd(new Point(13, 14));
18        System.out.println(l1); // Line's toString()
19        // Line[begin=(11,12),end=(13,14)]
20        System.out.println("begin is: " + l1.getBegin()); // Point's toString()
21        // begin is: (11,12)
22        System.out.println("end is: " + l1.getEnd()); // Point's toString()
23        // end is: (13,14)

24
25        l1.setBeginX(21);
26        l1.setBeginY(22);
27        l1.setEndX(23);
28        l1.setEndY(24);
29        System.out.println(l1); // Line's toString()
30        // Line[begin=(21,22),end=(23,24)]
31        System.out.println("begin's x is: " + l1.getBeginX());
32        // begin's x is: 21
33        System.out.println("begin's y is: " + l1.getBeginY());
34        // begin's y is: 22
35        System.out.println("end's x is: " + l1.getEndX());
36        // end's x is: 23
37        System.out.println("end's y is: " + l1.getEndY());
38        // end's y is: 24

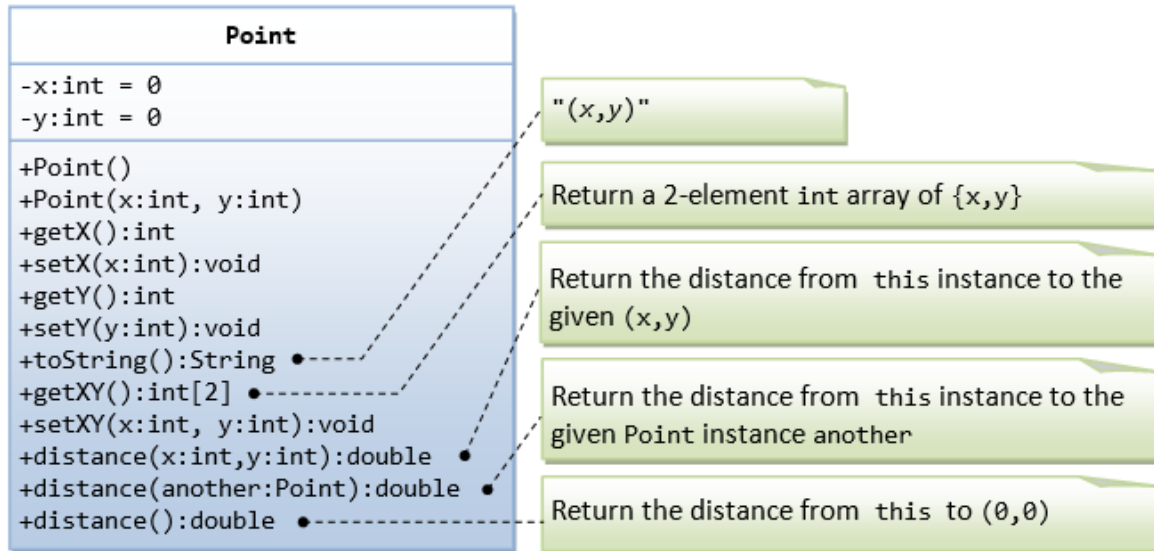
39
40        l1.setBeginXY(31, 32);
41        l1.setEndXY(33, 34);
42        System.out.println(l1); // Line's toString()
43        // Line[begin=(31,32),end=(33,34)]
44        System.out.println("begin's x and y are: "
45            + Arrays.toString(l1.getBeginXY()));
46        // begin's x and y are: [31, 32]
47        System.out.println("end's x and y are: " +
48            Arrays.toString(l1.getEndXY()));
49        // end's x and y are: [33, 34]

50
51        // Test getLength()
52        System.out.printf("length is: %.2f%n", l1.getLength());
53        // length is: 2.83
54    }
55 }

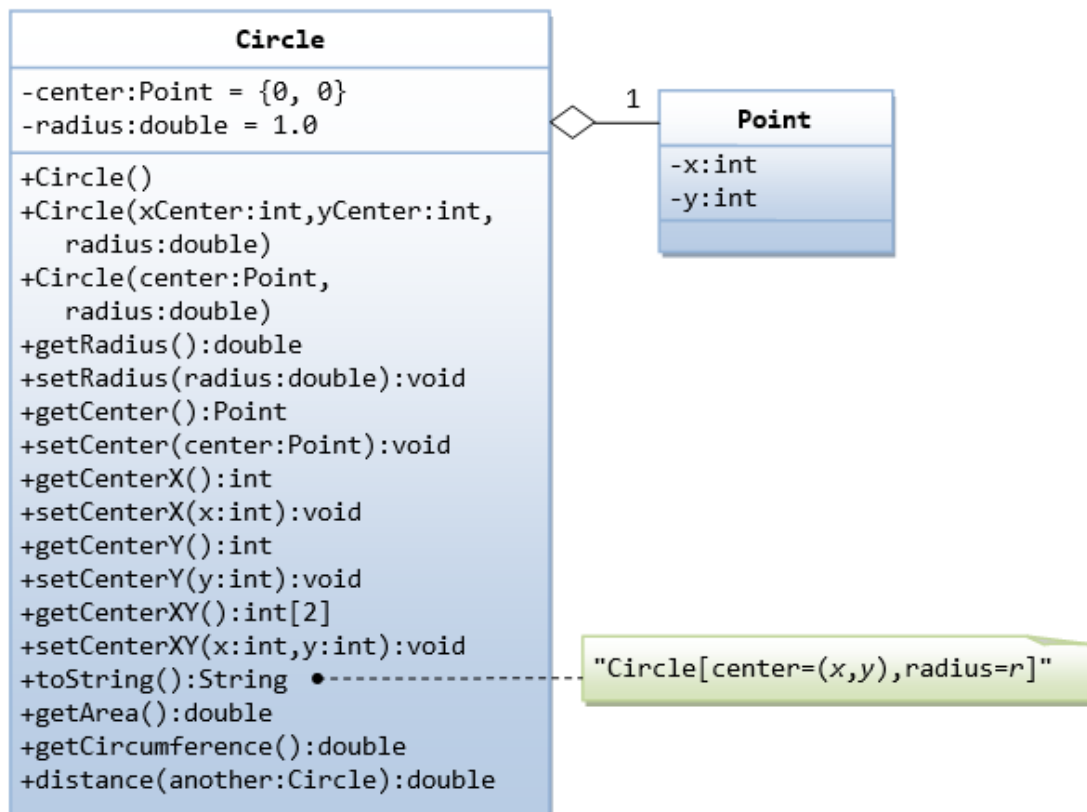
```

1.3 Composition EG. 3: The Point and Circle Classes

Suppose that we have an existing class called Point, defined as shown in the class diagram.



A class called Circle is designed as shown in the class diagram.



It contains:

- Two private member variables: a radius (double) and a center (an instance of Point class, which we created earlier). The constructors, public getters and setters. Methods getCenterX(), setCenterX(), getCenterY(), setCenterY(), getCenterXY(), setCenterXY(), etc. A toString() method that returns a string description of this instance in the format of "Circle[center = (x, y), radius = r]". You should re-use the Point's toString() to print "(x, y)". A distance(Circle another) method that returns the distance from the center of this instance to the center of the given Circle instance (called another).

The Circle class (Circle.java)



```

1  /**
   * The Circle class composes a "Point" instance as its center and a radius.
3  * We reuse the "Point" class via composition.
   */
5  public class Circle {
   // The private member variables
7  private Point center; // Declare an instance of the "Point" class
   private double radius;
9
   // Constructors
11 /** Constructs a Circle instance with the default values */
   public Circle() {
13     this.center = new Point(); // Construct a Point at (0, 0)
       this.radius = 1.0;
15 }

17 /** Constructs a Circle instance with the center at (xCenter, yCenter) and radius */
   public Circle(int xCenter, int yCenter, double radius) {
19     // Construct a Point at (xCenter, yCenter)
       center = new Point(xCenter, yCenter);
21     this.radius = radius;
   }
23
   /** Constructs a Circle instance with the given Point instance as center and radius */
25 public Circle(Point center, double radius) {
       this.center = center; // The caller had constructed a Point instance
27     this.radius = radius;
   }
29
   // Getters and Setters
31 public double getRadius() {
       return this.radius;
33 }

35 public void setRadius(double radius) {
       this.radius = radius;
37 }

```



```
39     public Point getCenter() {
40         return this.center; // return a Point instance
41     }
42
43     public void setCenter(Point center) {
44         this.center = center;
45     }
46
47     public int getCenterX() {
48         return center.getX(); // Point's getX()
49     }
50
51     public void setCenterX(int x) {
52         center.setX(x); // Point's setX()
53     }
54
55     public int getCenterY() {
56         return center.getY(); // Point's getY()
57     }
58
59     public void setCenterY(int y) {
60         center.setY(y); // Point's setY()
61     }
62
63     public int[] getCenterXY() {
64         return center.getXY(); // Point's getXY()
65     }
66
67     public void setCenterXY(int x, int y) {
68         center.setXY(x, y); // Point's setXY()
69     }
70
71     /** Returns a self-descriptive String */
72     public String toString() {
73         return "Circle[center=" + center + ",radius=" + radius + "]";
74     }
75
76     /** Returns the area of this circle */
77     public double getArea() {
78         return Math.PI * radius * radius;
79     }
80
81     /** Returns the circumference of this circle */
82     public double getCircumference() {
83         return 2.0 * Math.PI * radius;
84     }
85
86     /**
87     * Returns the distance from the center of this circle to the center
88     * of the given Circle instance called another
89     */
```



```

91     public double distance(Circle another) {
        return center.distance(another.center); // Use distance() of the Point class
93     }

```

A Test Driver for the Circle Class (TestCircle.java)



```

1  /**
   * A test driver for the Circle class.
   */
3  public class TestCircle {
5      public static void main(String[] args) {
        // Test Constructors and toString()
7         Circle c1 = new Circle();
        System.out.println(c1); // Circle's toString()
9         Circle c2 = new Circle(1, 2, 3.3);
        System.out.println(c2); // Circle's toString()
11        Circle c3 = new Circle(new Point(4, 5), 6.6); // an anonymous
                                                    // Point instance
13        System.out.println(c3); // Circle's toString()

15        // Test Setters and Getters
        c1.setCenter(new Point(11, 12));
17        c1.setRadius(13.3);
        System.out.println(c1); // Circle's toString()
19        System.out.println("center is: " + c1.getCenter()); // Point's toString()
        System.out.println("radius is: " + c1.getRadius());

21        c1.setCenterX(21);
23        c1.setCenterY(22);
        System.out.println(c1); // Circle's toString()
25        System.out.println("center's x is: " + c1.getCenterX());
        System.out.println("center's y is: " + c1.getCenterY());
27        c1.setCenterXY(31, 32);
        System.out.println(c1); // Circle's toString()
29        System.out.println("center's x is: " + c1.getCenterXY()[0]);
        System.out.println("center's y is: " + c1.getCenterXY()[1]);
31

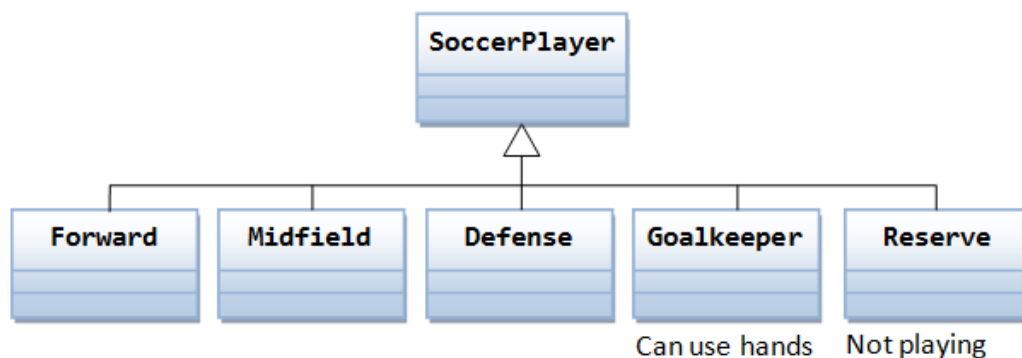
        // Test getArea() and getCircumference()
33        System.out.printf("area is: %.2f%n", c1.getArea());
        System.out.printf("circumference is: %.2f%n", c1.getCircumference());
35

        // Test distance()
37        System.out.printf("distance is: %.2f%n", c1.distance(c2));
        System.out.printf("distance is: %.2f%n", c2.distance(c1));
39    }
}

```

2 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 (static attributes) and methods (dynamic behaviors) 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*). By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, *redundancy* can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. For example,



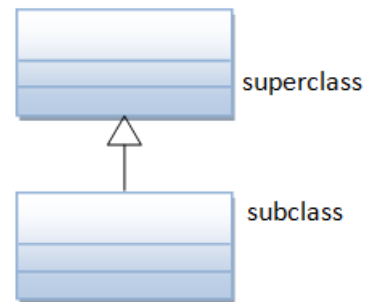
A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors. It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

In Java, you define a subclass using the keyword "extends", e.g.,

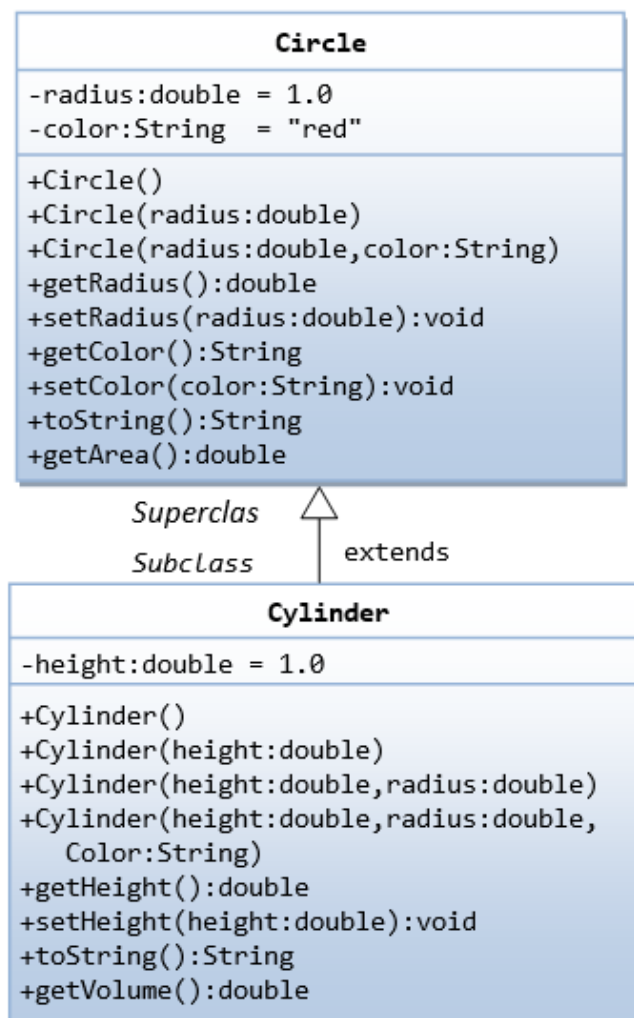


```
1  class Goalkeeper extends SoccerPlayer {
2      .....
3  }
4
5  class MyApplet extends java.applet.Applet {
6      .....
7  }
8
9  class Cylinder extends Circle {
10     .....
11 }
```

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



2.1 Inheritance EG. 1: The Circle and Cylinder Classes



In this example, we derive a subclass called Cylinder from the superclass Circle, which we have created in the previous chapter. It is important to note that we reuse the class Circle.

Reusability is one of the most important properties of OOP. (Why reinvent the wheels?) The class Cylinder inherits all the member variables (radius and color) and methods (getRadius(), getArea(), among others) from its superclass Circle. It further defines a variable called height, two public methods - getHeight() and getVolume() and its own constructors, as shown:

Circle.java (Re-produced)



```
1  public class Circle {
    // private instance variables
3  private double radius;
    private String color;
5
    // Constructors
7  public Circle() {
        this.radius = 1.0;
        this.color = "red";
        System.out.println("Construced a Circle with Circle()"); // for debugging
11 }

13 /** Constructs a Circle instance with the given radius */
    public Circle(double radius) {
15         this.radius = radius;
        this.color = "red";
        System.out.println("Construced a Circle with Circle(radius)"); // for debugging
17     }

19
21 /**
 * Constructs a Circle instance with the given radius and color
 */
23 public Circle(double radius, String color) {
        this.radius = radius;
25         this.color = color;
        System.out.println("Construced a Circle with Circle(radius, color)");
27     }

29 // public getters and setters for the private variables
    public double getRadius() {
31         return this.radius;
    }

33
    public String getColor() {
35         return this.color;
    }

37
    public void setRadius(double radius) {
39         this.radius = radius;
    }

41
    public void setColor(String color) {
43         this.color = color;
    }
}
```



```

    }
45
    /** Returns a self-descriptive String */
47    public String toString() {
        return "Circle[radius=" + radius + ",color=" + color + "]";
49    }

51    /** Returns the area of this Circle */
    public double getArea() {
53        return radius * radius * Math.PI;
    }
55 }

```

Cylinder.java



```

1  /**
   * A Cylinder is a Circle plus a height.
   */
3  public class Cylinder extends Circle {
4      // private instance variable
5      private double height;
6
7      // Constructors
8
9      public Cylinder() {
10         super(); // invoke superclass' constructor Circle()
11         this.height = 1.0;
12         System.out.println("Constructed a Cylinder with Cylinder()"); // for debugging
13     }
14
15     /**
16     * Constructs a Cylinder instance with the given height
17     */
18     public Cylinder(double height) {
19         super(); // invoke superclass' constructor Circle()
20         this.height = height;
21         System.out.println("Constructed a Cylinder with Cylinder(height)");
22     }
23
24     /**
25     * Constructs a Cylinder instance with the given height and radius
26     */
27     public Cylinder(double height, double radius) {
28         super(radius); // invoke superclass' constructor Circle(radius)
29         this.height = height;
30         System.out.println("Constructed a Cylinder with Cylinder(height, radius)");
31     }

```



```

33  /**
   * Constructs a Cylinder instance with the given height, radius and color
35  */
   public Cylinder(double height, double radius, String color) {
37      super(radius, color); // invoke superclass' constructor Circle(radius, color)
       this.height = height;
39      System.out.println("Constructed a Cylinder with Cylinder(height, radius, color)");
   }

41
   // Getter and Setter
43   public double getHeight() {
       return this.height;
45   }

47   public void setHeight(double height) {
       this.height = height;
49   }

51   /** Returns the volume of this Cylinder */
   public double getVolume() {
53       return getArea()*height; // Use Circle's getArea()
   }

55
   /** Returns a self-descriptive String */
57   public String toString() {
       return "This is a Cylinder"; // to be refined later
59   }
   }

```

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



```

/** A test driver for the Cylinder class. */
2  public class TestCylinder {
   public static void main(String[] args) {
4      Cylinder cy1 = new Cylinder();
       // Construced a Circle with Circle()
       // Constructed a Cylinder with Cylinder()
6      System.out.println("Radius is " + cy1.getRadius()
8          + ", Height is " + cy1.getHeight()
          + ", Color is " + cy1.getColor()
10         + ", Base area is " + cy1.getArea()
          + ", Volume is " + cy1.getVolume());
12     // Radius is 1.0, Height is 1.0, Color is red,
       // Base area is 3.141592653589793, Volume is 3.141592653589793

14
       Cylinder cy2 = new Cylinder(5.0, 2.0);
16     // Construced a Circle with Circle(radius)

```



```

18 // Constructed a Cylinder with Cylinder(height, radius)
    System.out.println("Radius is " + cy2.getRadius()
        + ", Height is " + cy2.getHeight()
20        + ", Color is " + cy2.getColor()
        + ", Base area is " + cy2.getArea()
22        + ", Volume is " + cy2.getVolume());
    // Radius is 2.0, Height is 5.0, Color is red,
24    // Base area is 12.566370614359172, Volume is 62.83185307179586
    }
26 }

```

Keep the "Cylinder.java" and "TestCylinder.java" in the same directory as "Circle.class" (because we are reusing the class Circle). Compile and run the program.

2.2 Method Overriding & Variable Hiding

A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). 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.

For example, the inherited method `getArea()` in a Cylinder object computes the base area of the cylinder. Suppose that we decide to override the `getArea()` to compute the surface area of the cylinder in the subclass Cylinder. Below are the changes:



```

public class Cylinder extends Circle {
2    .....
    // Override the getArea() method inherited from superclass Circle
4    @Override
    public double getArea() {
6        return 2*Math.PI*getRadius()*height + 2*super.getArea();
    }

8
    // Need to change the getVolume() as well
10    public double getVolume() {
        return super.getArea()*height;    // use superclass' getArea()
12    }

14    // Override the inherited toString()
    @Override
16    public String toString() {
        return "Cylinder[" + super.toString() + ",height=" + height + "]";
18    }
}

```

If `getArea()` is called from a `Circle` object, it computes the area of the circle. If `getArea()` is called from a `Cylinder` object, it computes the surface area of the cylinder using the *overridden implementation*. Note that you have to use public accessor method `getRadius()` to retrieve the radius of the `Circle`, because radius is declared private and therefore not accessible to other classes, including the subclass `Cylinder`.

But if you override the `getArea()` in the `Cylinder`, the `getVolume()` ($= \text{getArea()} * \text{height}$) no longer works. It is because the overridden `getArea()` will be used in `Cylinder`, which does not compute the base area. You can fix this problem by using `super.getArea()` to use the superclass' version of `getArea()`. Note that `super.getArea()` can only be issued from the subclass definition, but not from an instance created, e.g. `c1.super.getArea()`, as it break the information hiding and encapsulation principle.

2.3 Annotation `@Override` (JDK 1.5)

The "`@Override`" is known as annotation (introduced in JDK 1.5), which asks compiler to check whether there is such a method in the superclass to be overridden. This helps greatly if you *misspell* the name of the method to be overridden. For example, suppose that you wish to override method `toString()` in a subclass. If `@Override` is not used and `toString()` is misspelled as `TOSTring()`, it will be treated as a new method in the subclass, instead of overriding the superclass. If `@Override` is used, the compiler will signal an error.

`@Override` annotation is optional, but certainly nice to have.

Annotations are not programming constructs. They have no effect on the program output. It is only used by the compiler, discarded after compilation, and not used by the runtime.

2.4 Keyword "super"

Recall that inside a class definition, you can use the keyword `this` to refer to *this instance*. Similarly, the keyword `super` refers to the superclass, which could be the immediate parent or its ancestor.

The keyword `super` allows the subclass to access superclass' methods and variables within the subclass' definition. For example, `super()` and `super(argumentList)` can be used invoke the superclass' constructor. If the subclass overrides a method inherited from its superclass, says `getArea()`, you can use `super.getArea()` to invoke the superclass' version within the subclass definition. Similarly, if your subclass hides one of the superclass' variable, you can use `super.variableName` to refer to the hidden variable within the subclass definition.

2.5 More on Constructors

Recall that the subclass inherits all the variables and methods from its superclasses. Nonetheless, the subclass does not inherit the constructors of its superclasses. Each class in Java defines its own constructors.

In the body of a constructor, you can use `super(args)` to invoke a constructor of its immediate superclass. Note that `super(args)`, if it is used, must be the *first statement* in the subclass' constructor. If it is not used in the constructor, Java compiler automatically insert a `super()` statement to invoke the no-arg constructor of its immediate superclass. This follows the fact that the parent must be born before the child can be born. You need to properly construct the superclasses before you can construct the subclass.

2.6 Default no-arg Constructor

If no constructor is defined in a class, Java compiler automatically create a *no-argument (no-arg) constructor*, that simply issues a `super()` call, as follows:



```
1  // If no constructor is defined in a class, compiler inserts this no-arg constructor
   public ClassName () {
3    super();    // call the superclass ' no-arg constructor
   }
```

Take note that:

- The default no-arg constructor will not be automatically generated, if one (or more) constructor was defined. In other words, you need to define no-arg constructor explicitly if other constructors were defined.
- 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. Note that Java compiler inserts a `super()` as the first statement in a constructor if there is no `super(args)`.

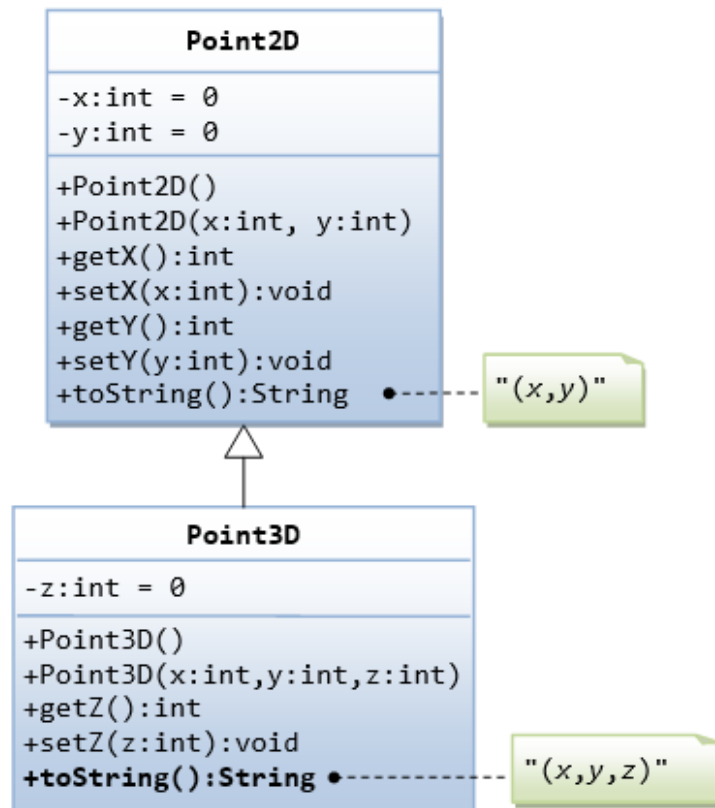
2.7 Single Inheritance

Java does not support multiple inheritance (C++ does). Multiple inheritance permits a subclass to have more than one direct superclasses. This has a serious drawback if the superclasses have conflicting implementation for the same method. In Java, each subclass can have one and only one direct superclass, i.e., single inheritance. On the other hand, a superclass can have many subclasses.

2.8 Common Root Class - `java.lang.Object`

Java adopts a so-called *common-root* approach. All Java classes are derived from a *common root class* called `java.lang.Object`. This `Object` class defines and implements the *common behaviors* that are required of all the Java objects running under the JRE. These common behaviors enable the implementation of features such as multi-threading and garbage collector.

2.9 Inheritance EG. 2: The Point2D and Point3D Classes



The Superclass Point2D.java



```

/**
2  * The Point2D class models a 2D point at (x, y).
   */
4  public class Point2D {

6      // Private instance variables
       private int x;
       private int y;

10     // Constructors
       /** Construct a Point2D instance at (0,0) */
12     public Point2D() { // default constructor
           this.x = 0;
14         this.y = 0;
       }

16     /** Constructs a Point2D instance at the given (x,y) */
18     public Point2D(int x, int y) {

```



```

20     this.x = x;
    this.y = y;
    }

22     // Getters and Setters
24     public int getX() {
        return this.x;
26     }

28     public void setX(int x) {
        this.x = x;
30     }

32     public int getY() {
        return this.y;
34     }

36     public void setY(int y) {
        this.y = y;
38     }

40     /** Returns a self-descriptive string in the form of "(x,y)" */
    @Override
42     public String toString() {
        return "(" + this.x + "," + this.y + ")";
44     }
    }

```

The Subclass Point3D.java



```

1  /**
   * The Point3D class models a 3D point at (x, y,z), which is a subclass of Point2D.
   */
3  public class Point3D extends Point2D {

5      // Private instance variables
7      private int z;

9      // Constructors
   /** Constructs a Point3D instance at (0,0,0) */
11     public Point3D() { // default constructor
        super(); // x = y = 0
13         this.z = 0;
    }

15     /** Constructs a Point3D instance at (x,y,z) */
17     public Point3D(int x, int y, int z) {

```




```

19     super(x, y);
    this.z = z;
21 }
22
23 // Getters and Setters
24 public int getZ() {
25     return this.z;
26 }
27
28 public void setZ(int z) {
29     this.z = z;
30 }
31
32 /** Returns a self-descriptive string in the form of "(x,y,z)" */
33 @Override
34 public String toString() {
35     return "(" + super.getX() + "," + super.getY() + "," + this.z + ")"
        ↪ ;
36 }
37 }

```

A Test Driver for Point2D and Point3D Classes (TestPoint2DPoint3D.java)



```

1  /**
2   * A test driver for the Point2D and Point3D classes
3   */
4  public class TestPoint2DPoint3D {
5      public static void main(String[] args) {
6          /* Test Point2D */
7          // Test constructors and toString()
8          Point2D p2a = new Point2D(1, 2);
9          System.out.println(p2a); // toString()
10         Point2D p2b = new Point2D(); // default constructor
11         System.out.println(p2b);
12         // Test Setters and Getters
13         p2a.setX(3); // Test setters
14         p2a.setY(4);
15         System.out.println(p2a); // toString()
16         System.out.println("x is: " + p2a.getX());
17         System.out.println("y is: " + p2a.getY());
18
19         /* Test Point3D */
20         // Test constructors and toString()
21         Point3D p3a = new Point3D(11, 12, 13);
22         System.out.println(p3a); // toString()
23         Point3D p3b = new Point3D(); // default constructor

```

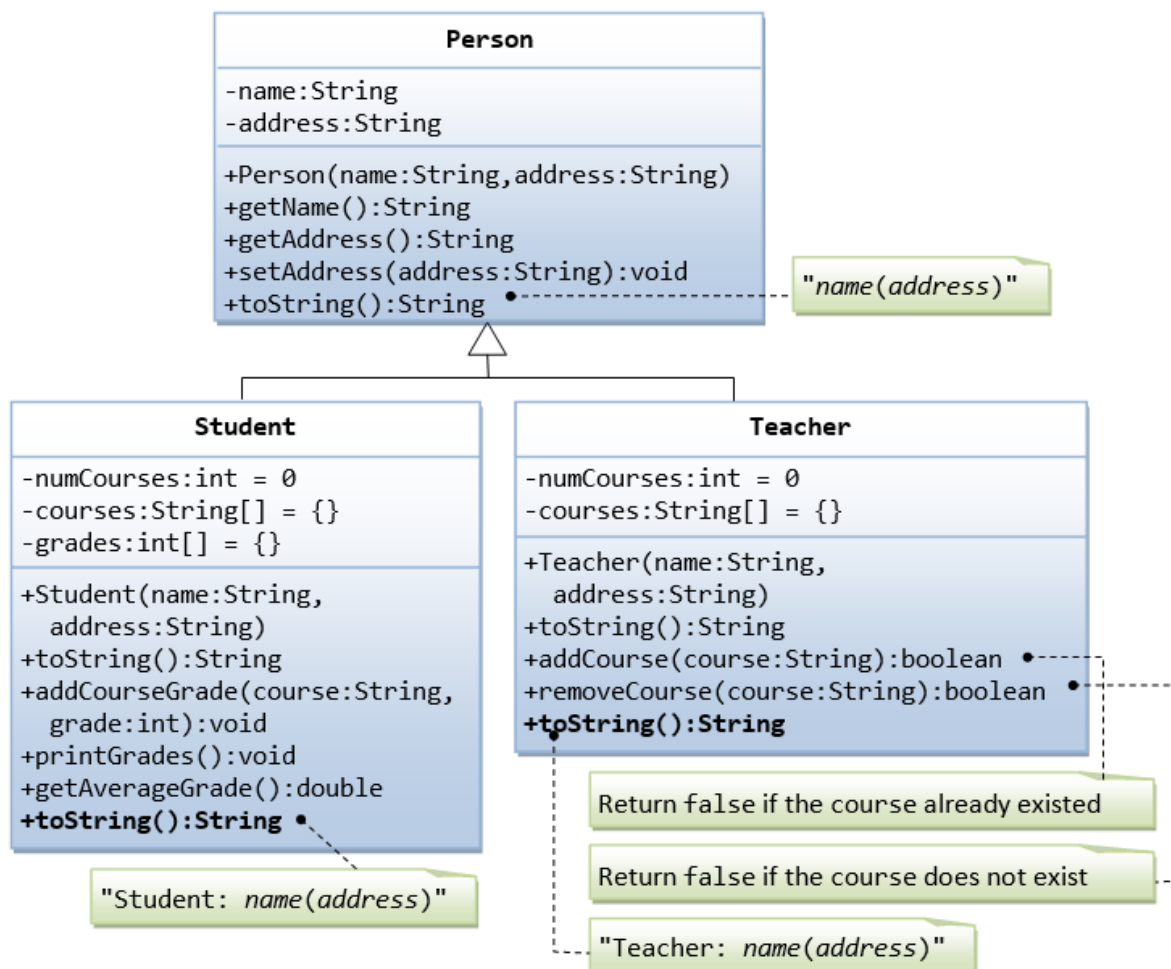


```

24      System.out.println(p3b);
        // Test Setters and Getters
26      p3a.setX(21); // in superclass
        p3a.setY(22); // in superclass
28      p3a.setZ(23); // in this class
        System.out.println(p3a); // toString()
30      System.out.println("x is: " + p3a.getX()); // in superclass
        System.out.println("y is: " + p3a.getY()); // in superclass
32      System.out.println("z is: " + p3a.getZ()); // in this class
        }
34    }

```

2.10 Inheritance EG. 3: Superclass Person and its Subclasses



Suppose that we are required to model students and teachers in our application. We can define a superclass called **Person** to store common properties such as name and address,

and subclasses Student and Teacher for their specific properties. For students, we need to maintain the courses taken and their respective grades; add a course with grade, print all courses taken and the average grade. Assume that a student takes no more than 30 courses for the entire program. For teachers, we need to maintain the courses taught currently, and able to add or remove a course taught. Assume that a teacher teaches not more than 5 courses concurrently.

We design the classes as follows.

The Superclass Person.java



```
/**
2  * The superclass Person has name and address.
   */
4  public class Person {

6      // private instance variables
       private String name;
8      private String address;

10     /**
       * Constructs a Person instance with the given name and address
12     */
       public Person(String name, String address) {
14         this.name = name;
           this.address = address;
16     }

18     // Getters and Setters
       /** Returns the name */
20     public String getName() {
           return name;
22     }

24     /** Returns the address */
       public String getAddress() {
26         return address;
           }

28     /** Sets the address */
       public void setAddress(String address) {
30         this.address = address;
32     }

34     /** Returns a self-descriptive string */
       @Override
36     public String toString() {
           return name + "(" + address + ")";
38     }
}
```

The Subclass Student.java



```

1  /** The Student class , subclass of Person. */
   public class Student extends Person {
3
   // private instance variables
5   private int numCourses; // number of courses taken so far
   private String[] courses; // course codes
7   private int[] grades; // grade for the corresponding course codes
   private static final int MAX_COURSES = 30; // maximum number of courses
9
   /** Constructs a Student instance with the given name and address */
11  public Student(String name, String address) {
       super(name, address);
13     numCourses = 0;
       courses = new String[MAX_COURSES];
15     grades = new int[MAX_COURSES];
   }
17
   /** Returns a self-descriptive string */
19  @Override
   public String toString() {
21     return "Student: " + super.toString();
   }
23
   /** Adds a course and its grade - No input validation */
25  public void addCourseGrade(String course, int grade) {
       courses[numCourses] = course;
27     grades[numCourses] = grade;
       ++numCourses;
29 }
31
   /** Prints all courses taken and their grade */
   public void printGrades() {
33     System.out.print(this);
       for (int i = 0; i < numCourses; ++i) {
35         System.out.print(" " + courses[i] + ":" + grades[i]);
       }
37     System.out.println();
   }
39
   /** Computes the average grade */
41  public double getAverageGrade() {
       int sum = 0;
43     for (int i = 0; i < numCourses; i++) {
         sum += grades[i];
45     }
       return (double)sum/numCourses;
47 }
   }

```

The Subclass Teacher.java



```

/**
2  * The Teacher class , subclass of Person .
*/
4  public class Teacher extends Person {

6      // private instance variables
    private int numCourses; // number of courses taught currently
8     private String[] courses; // course codes
    private static final int MAX_COURSES = 5; // maximum courses

10
    /**
12     * Constructs a Teacher instance with the given name and address
    */
14     public Teacher(String name, String address) {
        super(name, address);
16         numCourses = 0;
        courses = new String[MAX_COURSES];
18     }

20     /** Returns a self-descriptive string */
    @Override
22     public String toString() {
        return "Teacher: " + super.toString();
24     }

26     /**
    * Adds a course. Returns false if the course has already existed
28     */
    public boolean addCourse(String course) {
30        // Check if the course already in the course list
        for (int i = 0; i < numCourses; i++) {
32            if (courses[i].equals(course)) {
                return false;
34            }
        }

36        courses[numCourses] = course;
38        numCourses++;
        return true;
40    }

42     /**
    * Removes a course. Returns false if the course cannot be found in the course list
44     */
    public boolean removeCourse(String course) {
46        boolean found = false;
        // Look for the course index
48        int courseIndex = -1; // need to initialize
        for (int i = 0; i < numCourses; i++) {

```



```

50         if (courses[i].equals(course)) {
51             courseIndex = i;
52             found = true;
53             break;
54         }
55     }
56
57     if (found) {
58         // Remove the course and re-arrange for courses array
59         for (int i = courseIndex; i < numCourses-1; i++) {
60             courses[i] = courses[i+1];
61         }
62         numCourses--;
63         return true;
64     }
65
66     return false;
67 }
68 }

```

A Test Driver (TestPerson.java)



```

/**
2  * A test driver for Person and its subclasses.
3  */
4  public class TestPerson {
5      public static void main(String[] args) {
6          /* Test Student class */
7          Student s1 = new Student("Tan Ah Teck", "1 Happy Ave");
8          s1.addCourseGrade("IM101", 97);
9          s1.addCourseGrade("IM102", 68);
10         s1.printGrades();
11         // Student: Tan Ah Teck(1 Happy Ave) IM101:97 IM102:68
12         System.out.println("Average is " + s1.getAverageGrade());
13         // Average is 82.5
14
15         /* Test Teacher class */
16         Teacher t1 = new Teacher("Paul Tan", "8 sunset way");
17         System.out.println(t1);
18         // Teacher: Paul Tan(8 sunset way)
19
20         String[] courses = {"IM101", "IM102", "IM101"};
21         for (String course: courses) {
22             if (t1.addCourse(course)) {
23                 System.out.println(course + " added");
24             } else {

```



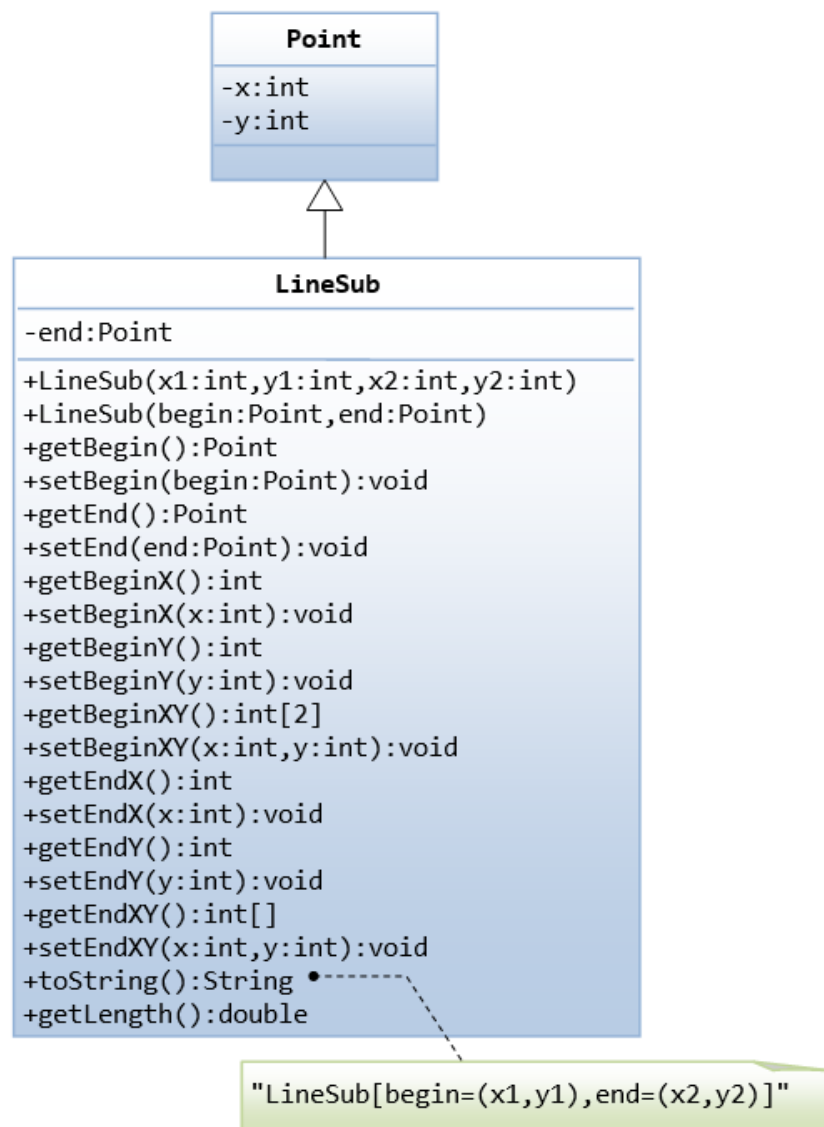
```
26         System.out.println(course + " cannot be added");
27     }
28 }
29 // IM101 added
30 // IM102 added
31 // IM101 cannot be added
32
33 for (String course: courses) {
34     if (t1.removeCourse(course)) {
35         System.out.println(course + " removed");
36     } else {
37         System.out.println(course + " cannot be removed");
38     }
39 }
40 // IM101 removed
41 // IM102 removed
42 // IM101 cannot be removed
43 }
44 }
```

3 Composition vs. Inheritance

3.1 "A line is composed of 2 points" vs. "A line is a point extended by another point"

Recall that there are two ways of reusing existing classes: *composition* and *inheritance*. We have seen that a Line class can be implemented using composition of Point class - "A line is composed of two points", in the previous section.

A Line can also be implemented, using inheritance from the Point class - "A line is a point extended by another point". Let's call this subclass LineSub (to differentiate from the Line class using composition).



The Subclass LineSub.java



```

1  /**
   * The LineSub class, subclass of Point. It inherits the begin point
3  * from the superclass, and adds an end point.
   */
5  public class LineSub extends Point { // Inherited the begin point

7      // Private instance variables
      Point end; // Declare end as instance of Point

9
11     /**
    * Constructs a LineSub instance with 2 points at (x1, y1) and (x2,y2)
    */
13     public LineSub(int x1, int y1, int x2, int y2) {
        super(x1, y1);
15         this.end = new Point(x2, y2); // Construct Point instances
    }

17
19     /** Constructs a LineSub instance with the 2 given Point instances */
    public LineSub(Point begin, Point end) {
        super(begin.getX(), begin.getY()); // Need to construct super
21         this.end = end;
    }

23
25     // Getters and Setters
    public Point getBegin() {
        return this; // upcast to Point (polymorphism - to be discussed later)
27     }

29     public Point getEnd() {
        return end;
31     }

33     public void setBegin(Point begin) {
        super.setX(begin.getX());
35         super.setY(begin.getY());
    }

37
39     public void setEnd(Point end) {
        this.end = end;
    }

41
43     // Other Get and Set methods
    public int getBeginX() {
        return super.getX(); // inherited, super is optional
45     }

47     public void setBeginX(int x) {
        super.setX(x); // inherited, super is optional
49     }

```



```

51     public int getBeginY() {
52         return super.getY();
53     }

55     public void setBeginY(int y) {
56         super.setY(y);
57     }

59     public int[] getBeginXY() {
60         return super.getXY();
61     }

63     public void setBeginXY(int x, int y) {
64         super.setXY(x, y);
65     }

67     public int getEndX() {
68         return end.getX();
69     }

71     public void setEndX(int x) {
72         end.setX(x);
73     }

75     public int getEndY() {
76         return end.getY();
77     }

79     public void setEndY(int y) {
80         end.setY(y);
81     }

83     public int[] getEndXY() {
84         return end.getXY();
85     }

87     public void setEndXY(int x, int y) {
88         end.setXY(x, y);
89     }

91     /** Returns a self-descriptive string */
92     public String toString() {
93         return "LineSub[begin=" + super.toString() + ",end=" + end + " ]";
94     }

95     /** Returns the length of this line */
96     public double getLength() {
97         return super.distance(end);
98     }
99 }

```

A Test Driver (TestLineSub.java)



```

1  /** Test Driver for the LineSub class */
2  public class TestLineSub {
3      public static void main(String[] args) {
4          // Test constructors and toString()
5          LineSub l1 = new LineSub(1, 2, 3, 4);
6          System.out.println(l1); // toString()
7          LineSub l2 = new LineSub(new Point(5,6), new Point(7,8));
8          System.out.println(l2);
9
10         // Test Setters and Getters
11         l1.setBegin(new Point(11, 12));
12         l1.setEnd(new Point(13, 14));
13         System.out.println(l1); // toString()
14         System.out.println("begin is: " + l1.getBegin());
15         System.out.println("end is: " + l1.getEnd());
16
17         l1.setBeginX(21);
18         l1.setBeginY(22);
19         l1.setEndX(23);
20         l1.setEndY(24);
21         System.out.println(l1);
22         System.out.println(l1); // toString()
23         System.out.println("begin's x is: " + l1.getBeginX());
24         System.out.println("begin's y is: " + l1.getBeginY());
25         System.out.println("end's x is: " + l1.getEndX());
26         System.out.println("end's y is: " + l1.getEndY());
27
28         l1.setBeginXY(31, 32);
29         l1.setEndXY(33, 34);
30         System.out.println(l1); // toString()
31         System.out.println("begin's x is: " + l1.getBeginXY()[0]);
32         System.out.println("begin's y is: " + l1.getBeginXY()[1]);
33         System.out.println("end's x is: " + l1.getEndXY()[0]);
34         System.out.println("end's y is: " + l1.getEndXY()[1]);
35
36         // Test getLength()
37         System.out.printf("length is: %.2f\n", l1.getLength());
38     }
39 }

```

Notes: This is the same test driver used in the earlier example on composition, except change in classname. Study both versions of the Line class (Line and LineSub). I suppose that it is easier to say that "A line is composed of two points" than that "A line is a point extended by another point".

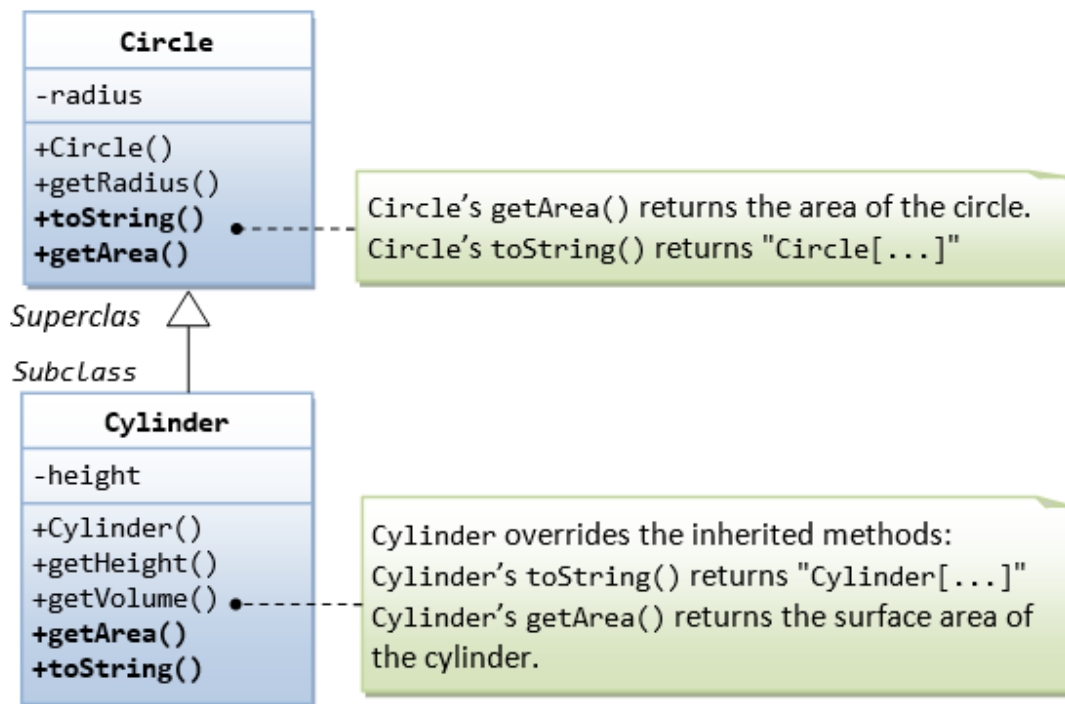
Rule of Thumb: Use composition if possible, before considering inheritance. Use inheritance only if there is a clear hierarchical relationship between classes.

4 Polymorphism

The word "*polymorphism*" means "*many forms*". It comes from Greek word "*poly*" (means *many*) and "*morphos*" (means *form*). For examples, in chemistry, carbon exhibits polymorphism because it can be found in more than one form: graphite and diamond. But, *each of the form has its own distinct properties* (and price).

4.1 Substitutability

A subclass possesses all the attributes and operations of its superclass (because a subclass inherited all attributes and operations from its superclass). This means that a subclass object can do whatever its superclass can do. As a result, we can *substitute* a subclass instance when a superclass instance is expected, and everything shall work fine. This is called *substitutability*.



In our earlier example of Circle and Cylinder: Cylinder is a subclass of Circle. We can say that Cylinder "**is-a**" Circle (actually, it "*is-more-than-a*" Circle). Subclass-superclass exhibits a so called "**is-a**" relationship.

Circle.java



```

1  /**
   * The superclass Circle
  
```



```
3  */
   public class Circle {
5
   // private instance variable
7   private double radius;

9   /** Constructs a Circle instance with the given radius */
   public Circle(double radius) {
11      this.radius = radius;
   }

13
   /** Returns the radius */
15   public double getRadius() {
       return this.radius;
17   }

19   /** Returns the area of this circle */
   public double getArea() {
21      return radius * radius * Math.PI;
   }

23
   /** Returns a self-descriptive string */
25   public String toString() {
       return "Circle[radius=" + radius + "]";
27   }
   }
```

Cylinder.java



```
   /**
2   * The subclass Cylinder
   */
4   public class Cylinder extends Circle {

6   // private instance variable
   private double height;

8
   /**
10  * Constructs a Cylinder instance with the given height and radius
   */
12  public Cylinder(double height, double radius) {
       super(radius);
14      this.height = height;
   }

16
   /** Returns the height */
18  public double getHeight() {
```



```

    return this.height;
20 }

22 /** Returns the volume of this cylinder */
    public double getVolumne() {
24     return super.getArea() * height;
    }

26
28 /**
 * Overrides the inherited method to return the surface area
 */
30 @Override
    public double getArea() {
32     return 2.0 * Math.PI * getRadius() * height;
    }

34
36 /**
 * Override the inherited method to describe itself
 */
38 @Override
    public String toString() {
40     return "Cylinder[height=" + height + ", " + super.toString() + "];"
    }
42 }

```

Via *substitutability*, we can create an instance of Cylinder, and assign it to a Circle (its superclass) reference, as follows:



```

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

```

You can invoke all the methods defined in the Circle class for the reference c1, (which is actually holding a Cylinder object), e.g.



```

// Invoke superclass Circle's methods
2 System.out.println(c1.getRadius());
// 2.2

```

This is because a subclass instance possesses all the properties of its superclass. However, you CANNOT invoke methods defined in the Cylinder class for the reference c1, e.g.



```
1 // CANNOT invoke method in Cylinder as c1 is a Circle reference
  c1.getHeight();
3 // compilation error: cannot find symbol method getHeight()

5 c1.getVolume();
  // compilation error: cannot find symbol method getVolume()
```

This is because `c1` is a reference to the `Circle` class, which does not know about methods defined in the subclass `Cylinder`.

`c1` is a reference to the `Circle` class, but holds an object of its subclass `Cylinder`. The reference `c1`, however, *retains its internal identity*. In our example, the subclass `Cylinder` overrides methods `getArea()` and `toString()`. `c1.getArea()` or `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.



```
    System.out.println(c1.toString()); // Run the overridden version
2 // Cylinder [height=1.1,Circle [radius=2.2]]

4 System.out.println(c1.getArea());    // Run the overridden version
  // 15.205308443374602
```

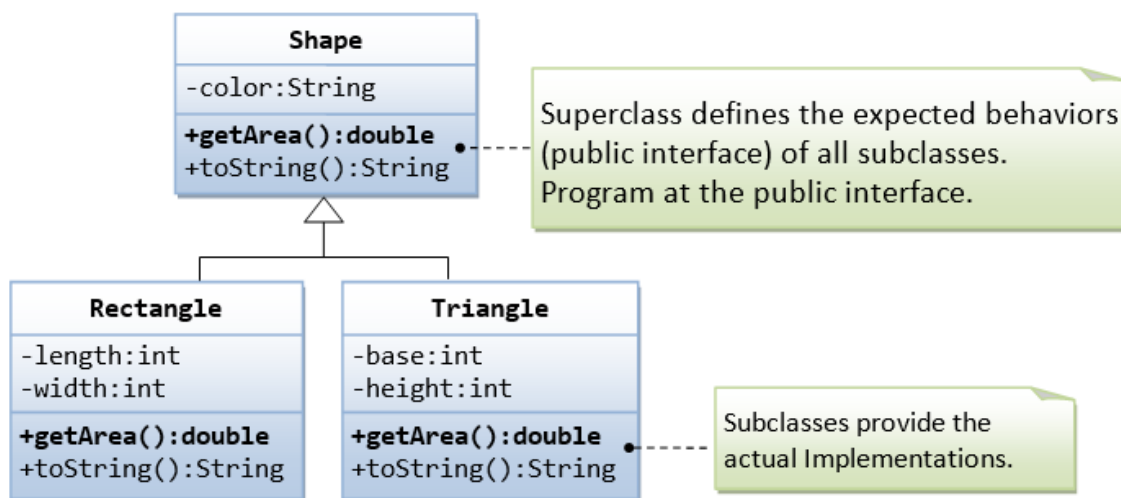
Summary

1. A subclass instance can be assigned (substituted) to a superclass' reference.
2. Once substituted, we can invoke methods defined in the superclass; we cannot invoke methods defined in the subclass.
3. However, if the subclass overrides inherited methods from the superclass, the subclass (overridden) versions will be invoked.

4.2 Polymorphism EG. 1: Shape and its Subclasses

Polymorphism is very powerful in OOP to *separate the interface and implementation* so as to allow the programmer to program at the interface in the design of a complex system.

Consider the following 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. For example, we would like all the shapes to have a method called `getArea()`, which returns the area of that particular shape. The Shape class can be written as follow.



Superclass Shape.java



```

1  /**
   * Superclass Shape maintain the common properties of all shapes
   */
3  public class Shape {
5
   // Private member variable
7  private String color;
9
   /** Constructs a Shape instance with the given color */
   public Shape (String color) {
11     this.color = color;
   }
13
   /** Returns a self-descriptive string */
   @Override
15   public String toString() {
17     return "Shape[ color=" + color + " ]";
   }
  
```




```

19      /** All shapes must provide a method called getArea() */
21      public double getArea() {
22          // We have a problem here!
23          // We need to return some value to compile the program.
24          System.err.println("Shape unknown! Cannot compute area!");
25          return 0;
26      }
27  }

```

Take note that we have a problem writing the `getArea()` method in the `Shape` class, because the area cannot be computed unless the actual shape is known. We shall print an error message for the time being. In the later section, I shall show you how to resolve this problem.

We can then derive subclasses, such as `Triangle` and `Rectangle`, from the superclass `Shape`.

Subclass `Rectangle.java`



```

1  /**
2   * The Rectangle class , subclass of Shape
3   */
4  public class Rectangle extends Shape {
5
6      // Private member variables
7      private int length;
8      private int width;
9
10     /** Constructs a Rectangle instance with the given color, length and width */
11     public Rectangle(String color , int length , int width) {
12         super(color);
13         this.length = length;
14         this.width = width;
15     }
16
17     /** Returns a self-descriptive string */
18     @Override
19     public String toString() {
20         return "Rectangle[length=" + length + ",width=" + width + ","
21             + super.toString() + " ]";
22     }
23
24     /** Override the inherited getArea() to provide the proper implementation for rectangle */
25     @Override
26     public double getArea() {
27         return length*width;
28     }
29 }

```

Subclass Triangle.java



```

1  /**
   * The Triangle class , subclass of Shape
   */
3  public class Triangle extends Shape {
5      // Private member variables
       private int base;
       private int height;

9      /**
       * Constructs a Triangle instance with the given color, base and height
       */
11     public Triangle(String color , int base , int height) {
13         super(color);
           this.base = base;
15         this.height = height;
       }

17     /**
19     * Returns a self-descriptive string
       */
21     @Override
       public String toString() {
23         return "Triangle [base=" + base + ",height=" + height + ","
           + super.toString() + "]";
25     }

27     /**
       * Override the inherited getArea() to provide the proper implementation for triangle
       */
29     @Override
31     public double getArea() {
           return 0.5*base*height;
33     }
}

```

The subclasses override the `getArea()` method inherited from the superclass, and provide the proper implementations for `getArea()`.

A Test Driver (TestShape.java)

In our application, we could create references of Shape, and assigned them instances of subclasses, as follows:



```

1  /**
2  * A test driver for Shape and its subclasses

```



```

*/
4 public class TestShape {
    public static void main(String[] args) {
6         Shape s1 = new Rectangle("red", 4, 5); // Upcast
        System.out.println(s1); // Run Rectangle's toString()
8         // Rectangle[length=4,width=5,Shape[ color=red ]]
        System.out.println("Area is " + s1.getArea()); // Run Rectangle's getArea()
10        // Area is 20.0

12        Shape s2 = new Triangle("blue", 4, 5); // Upcast
        System.out.println(s2); // Run Triangle's toString()
14        // Triangle[base=4,height=5,Shape[ color=blue ]]
        System.out.println("Area is " + s2.getArea()); // Run Triangle's getArea()
16        // Area is 10.0
    }
18 }

```

The beauty of this code is that all the references are from the superclass (i.e., programming at the interface level). You could instantiate different subclass instance, and the code still works. You could extend your program easily by adding in more subclasses, such as Circle, Square, etc, with ease.

Nonetheless, the above definition of Shape class poses a problem, if someone instantiate a Shape object and invoke the `getArea()` from the Shape object, the program breaks.



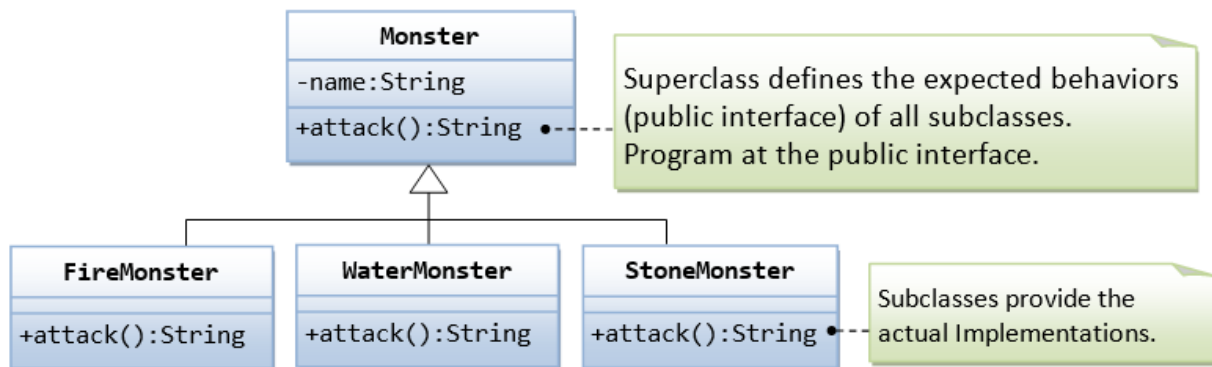
```

/**
2  * A test driver for Shape and its subclasses
*/
4 public class TestShape {
    public static void main(String[] args) {
6         // Constructing a Shape instance poses problem!
        Shape s3 = new Shape("green");
8         System.out.println(s3);
        // Shape[ color=green ]
10        System.out.println("Area is " + s3.getArea()); // Invalid output
        // Shape unknown! Cannot compute area!
12        // Area is 0.0
    }
14 }

```

This is because the Shape class is meant to provide a common interface to all its subclasses, which are supposed to provide the actual implementation. We do not want anyone to instantiate a Shape instance. This problem can be resolved by using the so-called abstract class.

4.3 Polymorphism EG. 2: Monster and its Subclasses



Polymorphism is a powerful mechanism in OOP to *separate the interface and implementation* so as to allow the programmer to program at the interface in the design of a complex system. For example, in our game app, we have many types of monsters that can attack. We shall design a superclass called `Monster` and define the method `attack()` in the superclass. The subclasses shall then provide their actual implementation. In the main program, we declare instances of superclass, substituted with actual subclass; and invoke method defined in the superclass.

Superclass `Monster.java`



```

/**
 * The superclass Monster defines the expected common behaviors for its subclasses.
 */
public class Monster {

    // private instance variable
    private String name;

    /**
     * Constructs a Monster instance with the given name
     */
    public Monster(String name) {
        this.name = name;
    }

    /** Defines a common behavior called attack() 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



```
/**
 * The class FireMonster
 */
2 public class FireMonster extends Monster {
    /**
     * Constructs a FireMonster with the given name
     */
    4 public FireMonster(String name) {
        super(name);
    6 }

    /**
     * Subclass provides actual implementation for attack()
     */
    8 @Override
    public String attack() {
        return "Attack with fire!";
    10 }
}
12
14
16
18
```

Subclass WaterMonster.java



```
1 /**
 * The class WaterMonster
 */
3 public class WaterMonster extends Monster {
    /**
     * Constructs a WaterMonster instance with the given name
     */
    5 public WaterMonster(String name) {
        super(name);
    7 }

    /**
     * Subclass provides actual implementation for attack()
     */
    9 @Override
    public String attack() {
        return "Attack with water!";
    11 }
}
13
15
17
19
```

Subclass StoneMonster.java



```

1  public class StoneMonster extends Monster {
    /**
3   * Constructs a StoneMonster instance with the given name
    */
5   public StoneMonster(String name) {
        super(name);
7   }

9   /** Subclass provides actual implementation for attack() */
    @Override
11  public String attack() {
        return "Attack with stones!";
13  }
    }

```

A Test Driver TestMonster.java



```

    public class TestMonster {
2      public static void main(String[] args) {
        // Program at the specification defined in the superclass/interface.
4        // Declare instances of the superclass, substituted by subclasses.
        Monster m1 = new FireMonster("r2u2"); // upcast
6        Monster m2 = new WaterMonster("u2r2"); // upcast
        Monster m3 = new StoneMonster("r2r2"); // upcast
8
        // Invoke the actual implementation
10       System.out.println(m1.attack()); // Run FireMonster's attack()
        // Attack with fire!
12       System.out.println(m2.attack()); // Run WaterMonster's attack()
        // Attack with water!
14       System.out.println(m3.attack()); // Run StoneMonster's attack()
        // Attack with stones!
16
        // m1 dies, generate a new instance and re-assign to m1.
18       m1 = new StoneMonster("a2b2"); // upcast
        System.out.println(m1.attack()); // Run StoneMonster's attack()
20       // Attack with stones!

22       // We have a problem here!!!
        Monster m4 = new Monster("u2u2");
24       System.out.println(m4.attack()); // garbage!!!
        // !^_&^$@+%$* I don't know how to attack!
26     }
    }

```

4.4 Upcasting & Downcasting

Upcasting a Subclass Instance to a Superclass Reference

Substituting a subclass instance for its superclass is called "*upcasting*". This is because, in a UML class diagram, subclass is often drawn below its superclass. Upcasting is *always safe* because a subclass instance possesses all the properties of its superclass and can do whatever its superclass can do. The compiler checks for valid upcasting and issues error "incompatible types" otherwise. For example,



```
1 Circle c1 = new Cylinder(1.1, 2.2); // Compiler checks to ensure that
                                     // R-value is a subclass of L-value.
3 Circle c2 = new String();           // Compilation error: incompatible types
```

Downcasting a Substituted Reference to Its Original Class

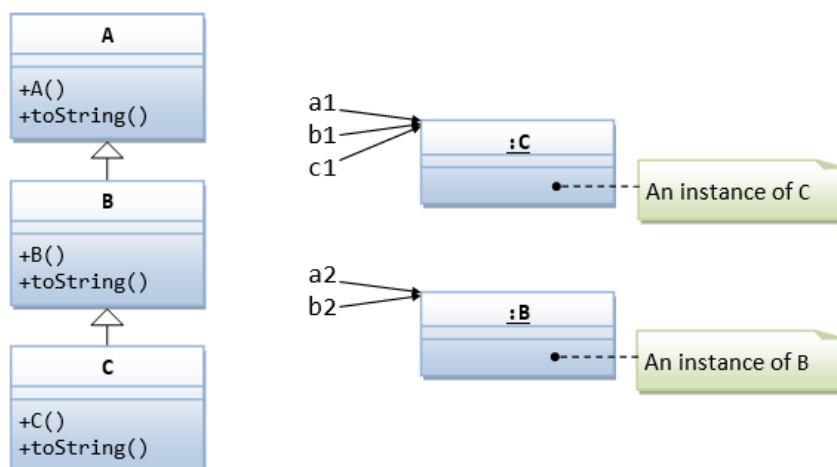
You can revert a substituted instance back to a subclass reference. This is called "*downcasting*". For example,



```
1 Circle c1 = new Cylinder(1.1, 2.2); // upcast is safe
  Cylinder cy1 = (Cylinder) c1;       // downcast needs the casting operator
```

Downcasting requires *explicit type casting operator* in the form of prefix operator (*new-type*). Downcasting is not always safe, and throws a runtime `ClassCastException` if the instance to be downcasted does not belong to the correct subclass. A subclass object can be substituted for its superclass, but the reverse is not true.

Another Example on Upcasting and Downcasting



A.java



```
public class A {  
2   public A() { // Constructor  
      System.out.println("Constructed an instance of A");  
4   }  
  
6   @Override  
   public String toString() {  
8       return "This is A";  
   }  
10 }
```

B.java



```
public class B extends A {  
2   public B() { // Constructor  
      super();  
4   System.out.println("Constructed an instance of B");  
   }  
  
6   @Override  
8   public String toString() {  
      return "This is B";  
10  }  
   }
```

C.java



```
1   public class C extends B {  
   public C() { // Constructor  
3       super();  
      System.out.println("Constructed an instance of C");  
5   }  
  
7   @Override  
   public String toString() {  
9       return "This is C";  
   }  
11 }
```


The following program tests the upcasting and downcasting (refer to the above instance diagram):

TestCasting.java



```

1  public class TestCasting {
    public static void main(String[] args) {
2      A a1 = new C();    // upcast
        // Constructed an instance of A
3      // Constructed an instance of B
        // Constructed an instance of C
4      System.out.println(a1); // run C's toString()
        // This is C
5      B b1 = (B)a1;      // downcast okay
        System.out.println(b1); // run C's toString()
6      // This is C
        C c1 = (C)b1;      // downcast okay
7      System.out.println(c1); // run C's toString()
        // This is C
8
9      A a2 = new B();    // upcast
        // Constructed an instance of A
10     // Constructed an instance of B
11     System.out.println(a2); // run B's toString()
        // This is B
12     B b2 = (B)a2;      // downcast okay
13     C c2 = (C)a2;
        // compilation okay, but runtime error:
14     // java.lang.ClassCastException: class B cannot be cast to class C
15
16     }
17 }

```

Casting Operator

Compiler may not be able to detect error in explicit cast, which will be detected only at runtime. For example,



```

1  Circle c1 = new Circle(5);
2  Point p1 = new Point();

3
4  c1 = p1;           // compilation error: incompatible types (Point is not a subclass of Circle)
5  c1 = (Circle)p1;   // runtime error: java.lang.ClassCastException:
6                      // Point cannot be casted to Circle

```

4.5 The "instanceof" Operator

Java provides a binary operator called instanceof which returns true if an object is an instance of a particular class. The syntax is as follows:



```
anObject instanceof aClass
```



```
1  Circle c1 = new Circle();
   System.out.println(c1 instanceof Circle); // true
3
   if (c1 instanceof Circle) {
5       .....
   }
```

An instance of subclass is also an instance of its superclass. For example,



```
   Circle c1 = new Circle(1.1);
2  Cylinder cy1 = new Cylinder(2.2, 3.3);
   System.out.println(c1 instanceof Circle); //true
4   System.out.println(c1 instanceof Cylinder); //false
   System.out.println(cy1 instanceof Cylinder); //true
6   System.out.println(cy1 instanceof Circle); //true

8   Circle c2 = new Cylinder(4.4, 5.5);
   System.out.println(c2 instanceof Circle); //true
10  System.out.println(c2 instanceof Cylinder); //true
```

4.6 Summary of Polymorphism

1. A subclass instance processes all the attributes operations of its superclass. When a superclass instance is expected, it can be substituted by a subclass instance. In other words, a reference to a class may hold an instance of that class or an instance of one of its subclasses - it is called substitutability.
2. If a subclass instance is assign to a superclass reference, you can invoke the methods defined in the superclass only. You cannot invoke methods defined in the subclass.
3. However, the substituted instance retains its own identity in terms of overridden methods and hiding variables. If the subclass overrides methods in the superclass, the subclass's version will be executed, instead of the superclass's version.

5 Abstract Classes & Interfaces

5.1 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.

For example, in the `Shape` class, we can declare abstract methods `getArea()`, `draw()`, etc, as follows:

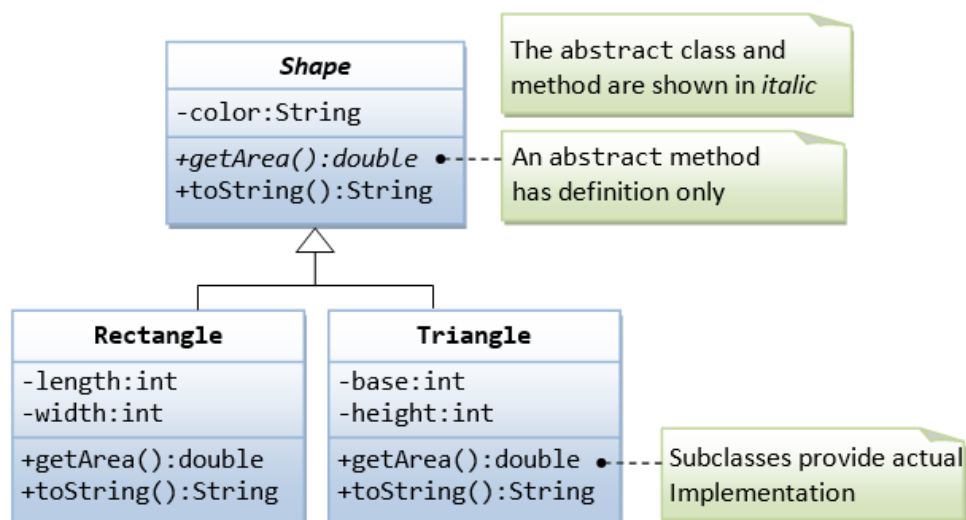


```

1 public abstract class Shape {
2     .....
3     public abstract double getArea();
4     public abstract double getPerimeter();
5     public abstract void draw();
6 }

```

Implementation of these methods is NOT possible in the `Shape` class, as the actual shape is not yet known. (How to compute the area if the shape is not known?) Implementation of these abstract methods will be provided later once the actual shape is known. These abstract methods cannot be invoked because they have no implementation.



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.

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

Note: In the above examples of Shape and Monster, we encountered a problem when we create instances of Shape and Monster and run the `getArea()` or `attack()`. This can be resolved via abstract method and abstract class.

5.2 Abstract Class EG. 1: Shape and its Subclasses

Let us rewrite our Shape class as an abstract class, containing an abstract method `getArea()` as follows:

The abstract Superclass (Shape.java)



```
/**
2  * This abstract superclass Shape contains an abstract method getArea(),
  * to be implemented by its subclasses.
4  */
public abstract class Shape {
6    // Private member variable
    private String color;

8

10   /**
   * Constructs a Shape instance with the given color
   */
12   public Shape (String color) {
        this.color = color;
14   }

16   /** Returns a self-descriptive string */
   @Override
18   public String toString() {
        return "Shape[ color=" + color + " ]";
20   }

22   /** All Shape's concrete subclasses must implement a method called getArea() */
   public abstract double getArea();
24 }
```

An abstract class is *incomplete* in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class *cannot be instantiated*. In other words, you cannot create instances from an abstract class (otherwise, you will have an incomplete instance with missing method's body).

To use an abstract class, you have to derive a subclass from the abstract class. In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods. The subclass derived is now complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.)

This property of the abstract class solves our earlier problem. In other words, you can create instances of the subclasses such as `Triangle` and `Rectangle`, and upcast them to `Shape` (so as to program and operate at the interface level), but you cannot create instance of `Shape`, which avoid the pitfall that we have faced. For example,

TestShape.java



```
1  /**
2   * The class TestShape
3   */
4  public class TestShape {
5      public static void main(String[] args) {
6          Shape s1 = new Rectangle("red", 4, 5);
7          System.out.println(s1);
8          System.out.println("Area is " + s1.getArea());
9
10         Shape s2 = new Triangle("blue", 4, 5);
11         System.out.println(s2);
12         System.out.println("Area is " + s2.getArea());
13
14         // Cannot create instance of an abstract class
15         Shape s3 = new Shape("green");
16         // compilation error: Shape is abstract; cannot be instantiated
17     }
18 }
```

In summary, an *abstract class provides a template for further development*. The purpose of an abstract class is to provide a common interface (or protocol, or contract, or understanding, or naming convention) to all its subclasses. For example, in the abstract class `Shape`, you can define abstract methods such as `getArea()` and `draw()`. No implementation is possible because the actual shape is not known. However, by specifying the signature of the abstract methods, *all the subclasses are forced to use these methods' signature*. The subclasses could provide the proper implementations.

Coupled with polymorphism, you can upcast subclass instances to `Shape`, and program at the `Shape` level, i.e., program at the interface. The separation of interface and implementation enables better software design, and ease in expansion. For example, `Shape` defines a method called `getArea()`, which all the subclasses must provide the correct implementation. You can ask for a `getArea()` from any subclasses of `Shape`, the correct area will be computed. Furthermore, your application can be extended easily to accommodate new shapes (such as `Circle` or `Square`) by deriving more subclasses.

Rule of Thumb: Program at the interface, not at the implementation. (That is, make references at the superclass; substitute with subclass instances; and invoke methods defined in the superclass only.)

Notes:

- An abstract method cannot be declared final, as final method cannot be overridden. An abstract method, on the other hand, must be overridden in a descendant before it can be used.
- An abstract method cannot be private (which generates a compilation error). This is because private methods are not visible to the subclass and thus cannot be overridden.

5.3 Abstract Class EG. 2: Monster

We shall define the superclass Monster as an abstract class, containing an abstract method attack(). The abstract class cannot be instantiated (i.e., creating instances).

Monster.java

```
1  /**
2   * The abstract superclass Monster defines the expected common behaviors,
3   * via abstract methods.
4   */
5   public abstract class Monster {
6       // private instance variable
7       private String name;
8
9
10      /**
11       * Constructs a Monster instance of the given name
12       */
13      public Monster(String name) {
14          this.name = name;
15      }
16
17      /**
18       * Define common behavior for all its subclasses
19       */
20      public abstract String attack();
21  }
```

5.4 The Java's interface

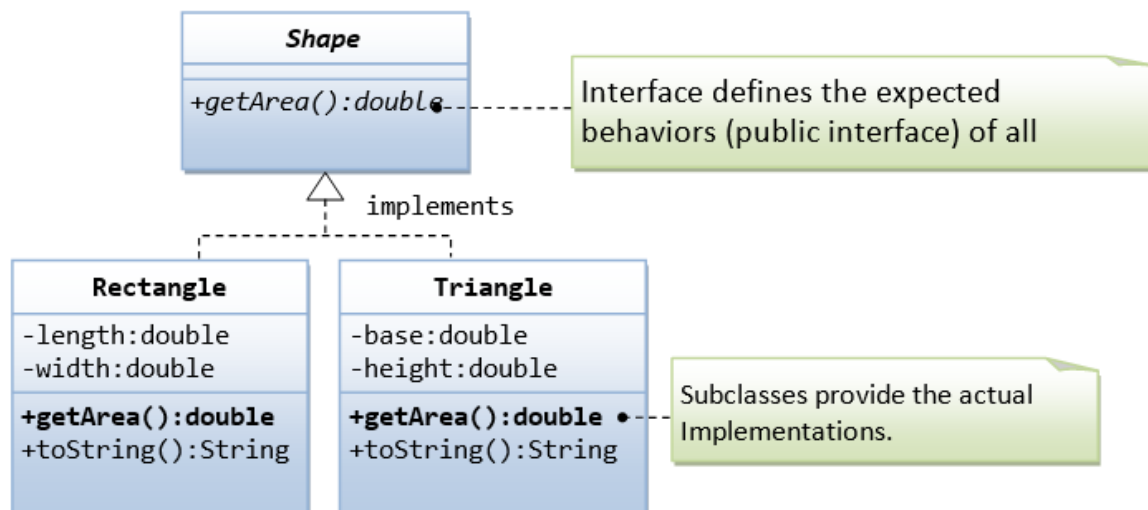
A Java interface is a 100% *abstract superclass* which define a set of methods its subclasses must support. An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables). You have to use the keyword "interface" 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. (JDK 8 introduces default and static methods in the interface. JDK 9 introduces private methods in the interface. These will not be covered in this article.)

- Similar to an abstract superclass, an interface cannot be instantiated. You have to create a "subclass" that implements an interface, and provide the actual implementation of all the abstract methods.
- Unlike a normal class, where you use the keyword "extends" to derive a subclass. 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.
- An interface provides a *form*, a *protocol*, a *standard*, a *contract*, a *specification*, a set of *rules*, an *interface*, for all objects that implement it. It is a *specification* and *rules* that any object implementing it agrees to follow.
- In Java, abstract class and interface are used to *separate the public interface of a class from its implementation* so as to allow the programmer to *program at the interface instead of the various implementation*.

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, Extensalizable, Movable, Clonable, Runnable, etc.

5.5 Interface EG. 1: Shape Interface and its Implementations

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



UML Notations: Abstract classes, Interfaces and abstract methods are shown in italics. Implementation of interface is marked by a dash-arrow leading from the subclasses to the interface.

Shape.java



```
/**
 * 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();
}
```

Rectangle.java



```
/**
 * 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;

    /**
     * Constructs a Rectangle instance with the given length and width
     */
    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    /** Returns a self-descriptive string */
    @Override
    public String toString() {
        return "Rectangle[length=" + length + ",width=" + width + "];"
    }

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


Triangle.java



```
1  /**
   * The subclass Triangle need to implement all the abstract methods in Shape
   */
3  */
   public class Triangle implements Shape {
5
       // Private member variables
7       private int base;
       private int height;
9
       /** Constructs a Triangle instance with the given base and height */
11      public Triangle(int base, int height) {
           this.base = base;
13          this.height = height;
       }
15
       /** Returns a self-descriptive string */
17      @Override
       public String toString() {
19          return "Triangle [base=" + base + ", height=" + height + "]";
       }
21
       // Need to implement all the abstract methods defined in the interface
23      /** Returns the area of this triangle */
       @Override
25      public double getArea() {
           return 0.5 * base * height;
27      }
   }
```

A test driver is as follows:

TestShape.java



```
   public class TestShape {
2       public static void main(String[] args) {
           Shape s1 = new Rectangle(1, 2); // upcast
4           System.out.println(s1);
           // Rectangle [length=1,width=2]
6           System.out.println("Area is " + s1.getArea());
           // Area is 2.0
8
           Shape s2 = new Triangle(3, 4); // upcast
10          System.out.println(s2);
           // Triangle [base=3,height=4]
12          System.out.println("Area is " + s2.getArea());
       }
```



```

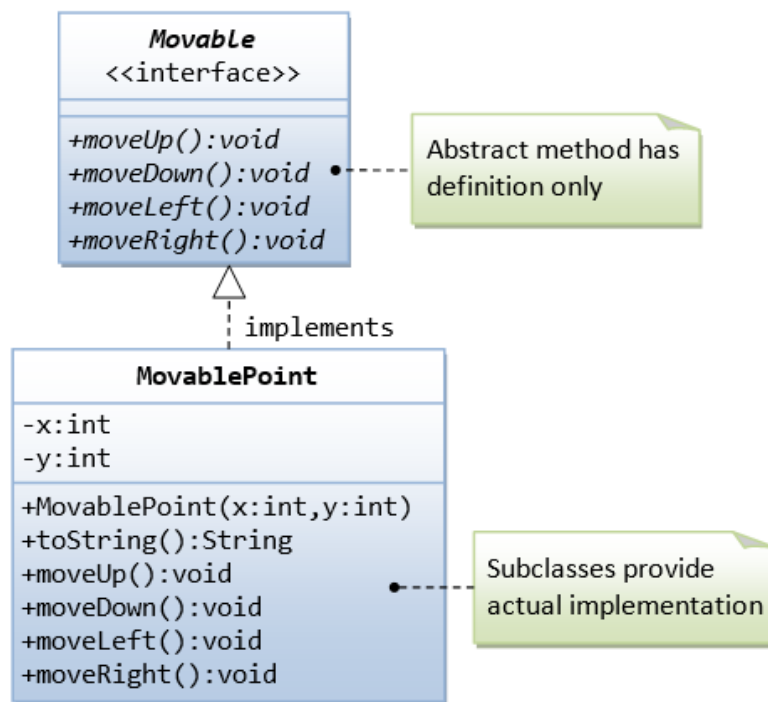
14      // Area is 6.0

16      // Cannot create instance of an interface
17      Shape s3 = new Shape("green");
18      // compilation error: Shape is abstract; cannot be instantiated
19  }
20  }

```

5.6 Interface EG. 2: Movable Interface and its Implementations

Suppose that our application involves many objects that can move. We could define an interface called movable, containing the signatures of the various movement methods.



Interface Moveable.java



```

1  /**
2   * The Movable interface defines a list of public abstract methods
3   * to be implemented by its subclasses
4   */
5  public interface Movable { // use keyword "interface" (instead of
6                             // "class") to define an interface
7      // An interface defines a list of public abstract methods to be implemented by the subclasses

```



```

9      public void moveUp();        // "public" and "abstract" optional
      public void moveDown();
      public void moveLeft();
11     public void moveRight();
    }

```

Similar to an abstract class, an interface cannot be instantiated; because it is incomplete (the abstract methods' body is missing). To use an interface, again, you must derive subclasses and provide implementation to all the abstract methods declared in the interface. The subclasses are now complete and can be instantiated.

MovablePoint.java

To derive subclasses from an interface, a new keyword "implements" is to be used instead of "extends" for deriving subclasses from an ordinary class or an abstract class. It is important to note that the subclass implementing an interface need to override ALL the abstract methods defined in the interface; otherwise, the subclass cannot be compiled. For example,



```

2      /**
   * The subclass MovablePoint needs to implement all the abstract methods
   * defined in the interface Movable
4      */
   public class MovablePoint implements Movable {
6
       // Private member variables
8       private int x;    // x coordinate of the point
       private int y;    // y coordinate of the point
10
       /** Constructs a MovablePoint instance at the given x and y */
12      public MovablePoint(int x, int y) {
           this.x = x;
14         this.y = y;
       }
16
       /** Returns a self-descriptive string */
18      @Override
       public String toString() {
20         return "(" + x + "," + y + ")";
       }
22
       // Need to implement all the abstract methods defined in the interface Movable
24      @Override
       public void moveUp() {
26         y--;
       }
28
       @Override

```



```

30     public void moveDown() {
31         y++;
32     }

34     @Override
35     public void moveLeft() {
36         x--;
37     }

38     @Override
39     public void moveRight() {
40         x++;
41     }
42 }

```

Other classes in the application can similarly implement the Movable interface and provide their own implementation to the abstract methods defined in the interface Movable.

TestMovable.java

We can also upcast subclass instances to the Movable interface, via polymorphism, similar to an abstract class.



```

1  public class TestMovable {
2      public static void main(String[] args) {
3          MovablePoint p1 = new MovablePoint(1, 2);
4          System.out.println(p1);
5          // (1,2)
6          p1.moveDown();
7          System.out.println(p1);
8          // (1,3)
9          p1.moveRight();
10         System.out.println(p1);
11         // (2,3)

12         // Test Polymorphism
13         Movable p2 = new MovablePoint(3, 4); // upcast
14         p2.moveUp();
15         System.out.println(p2);
16         // (3,3)

17         MovablePoint p3 = (MovablePoint)p2; // downcast
18         System.out.println(p3);
19         // (3,3)
20     }
21 }
22
23 }

```

5.7 Implementing Multiple Interfaces

As mentioned, Java supports only *single inheritance*. That is, a subclass can be derived from one and only one superclass. *Java does not support multiple inheritance* to avoid inheriting conflicting properties from multiple superclasses. Multiple inheritance, however, does have its place in programming.

A subclass, however, can implement more than one interfaces. This is permitted in Java as an interface merely defines the abstract methods without the actual implementations and less likely leads to inheriting conflicting properties from multiple interfaces. In other words, Java indirectly supports multiple inheritances via implementing multiple interfaces. For example,



```
1 public class Circle extends Shape implements Movable, Adjustable {  
    // extends one superclass but implements multiple interfaces  
3     .....  
}
```

5.8 interface Formal Syntax

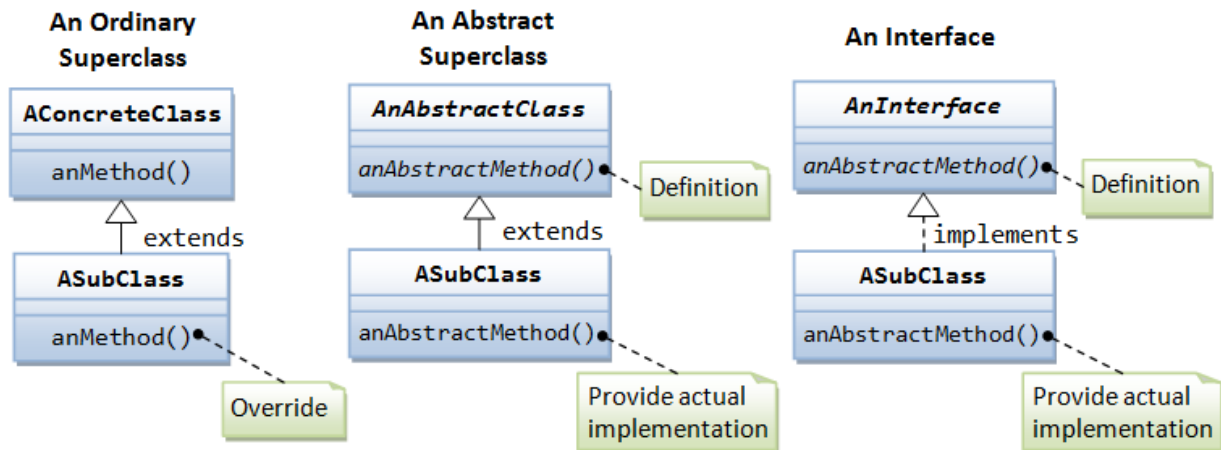
The formal syntax for declaring interface is:



```
    [public|protected|package] interface interfaceName  
2    [extends superInterfaceName] {  
    // constants  
4    static final ...;  
  
6    // public abstract methods' signature  
    ...  
8    }
```

- 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).
- An interface may "extends" from a super-interface.

UML Notation: The UML notation uses a solid-line arrow linking the subclass to a concrete or abstract superclass, and dashed-line arrow to an interface as illustrated. Abstract class and abstract method are shown in italics.



5.9 Why interfaces?

An interface is a *contract* (or a protocol, or a common understanding) of what the classes can do. When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface. Interface defines a set of common behaviors. The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors. This allows you to program at the interface, instead of the actual implementation. One of the main usage of interface is provide a *communication contract* between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.

Secondly, Java does not support multiple inheritance (whereas C++ does). Multiple inheritance permits you to derive a subclass from more than one direct superclass. This poses a problem if two direct superclasses have conflicting implementations. (Which one to follow in the subclass?). However, multiple inheritance does have its place. Java does this by permitting you to "implements" more than one interfaces (but you can only "extends" from a single superclass). Since interfaces contain only abstract methods without actual implementation, no conflict can arise among the multiple interfaces. (Interface can hold constants but is not recommended. If a subclass implements two interfaces with conflicting constants, the compiler will flag out a compilation error.)

5.10 Interface vs. Abstract Superclass

Which is a better design: interface or abstract superclass? There is no clear answer.

Use abstract superclass if there is a clear class hierarchy. Abstract class can contain partial implementation (such as instance variables and methods). Interface cannot contain any implementation, but merely defines the behaviors.

5.11 Dynamic Binding or Late Binding

We often treat an object not as its own type, but as its base type (superclass or interface). This allows you to write codes that do not depend on a specific implementation type. In the Shape example, we can always use `getArea()` and do not have to worry whether they are triangles or circles.

This, however, poses a new problem. The compiler cannot know at compile time precisely which piece of codes is going to be executed at run-time (e.g., `getArea()` has different implementation for `Rectangle` and `Triangle`).

In the procedural language like C, the compiler generates a call to a specific function name, and the linkage editor resolves this call to the absolute address of the code to be executed at run-time. This mechanism is called *static binding* (or *early binding*).

To support polymorphism, object-oriented language uses a different mechanism called *dynamic binding* (or *late-binding* or *run-time binding*). When a method is invoked, the code to be executed is only determined at run-time. During the compilation, the compiler checks whether the method exists and performs type check on the arguments and return type, but does not know which piece of codes to execute at run-time. When a message is sent to an object to invoke a method, the object figures out which piece of codes to execute at run-time.

Although dynamic binding resolves the problem in supporting polymorphism, it poses another new problem. The compiler is unable to check whether the type casting operator is safe. It can only be checked during runtime (which throws a `ClassCastException` if the type check fails).

JDK 1.5 introduces a new feature called *generics* to tackle this issue.

6 Object-Oriented Design Issues

6.1 Encapsulation, Coupling & Cohesion

In OO Design, it is desirable to design classes that are tightly encapsulated, loosely coupled and highly cohesive, so that the classes are easy to maintain and suitable for re-use.

Encapsulation refers to keeping the data and method inside a class such users do not access the data directly but via the public methods. *Tight encapsulation* is desired, which can be achieved by declaring all the variable private, and providing public getter and setter to the variables. The benefit is you have complete control on how the data is to be read (e.g., in how format) and how to the data is to be changed (e.g., validation).

Example: Time class with private variables hour (0-23), minute (0-59) and second (0-59); getters and setters (throws `IllegalArgumentException`). The internal time could also be stored as the number of seconds since midnight for ease of operation (information hiding).

Information Hiding: Another key benefit of tight encapsulation is information hiding, which

means that the users are not aware (and do not need to be aware) of how the data is stored internally.

Coupling refers to the degree to which one class relies on knowledge of the *internals* of another class. Tight coupling is undesirable because if one class changes its internal representations, all the other tightly-coupled classes need to be rewritten.

Example: A class uses Time and relies on the variables hour, minute and second.

Clearly, Loose Coupling is often associated with tight encapsulation. For example, well-defined public method for accessing the data, instead of directly access the data.

Cohesion refers to the degree to which a class or method resists being broken down into smaller pieces. High degree of cohesion is desirable. Each class shall be designed to model a single entity with its focused set of responsibilities and perform a collection of closely related tasks; and each method shall accomplish a single task. Low cohesion classes are hard to maintain and re-use.

Example of low cohesion: Book and Author in one class, or Car and Driver in one class.

Again, high cohesion is associated with loose coupling. This is because a highly cohesive class has fewer (or minimal) interactions with other classes.

6.2 "is-a" vs. "has-a" relationships

"*is-a*" relationship: A subclass object processes all the data and methods from its superclass (and it could have more). We can say that a subclass object is-a superclass object (is more than a superclass object). Refer to "polymorphism".

"*has-a*" relationship: In composition, a class contains references to other classes, which is known as "has-a" relationship.

You can use "is-a" and 'has-a" to test whether to design the classes using inheritance or composition.

6.3 Program at the interface specification, not the implementation

Refer to polymorphism