Java Programming Tutorial

OOP - Composition, Inheritance & Polymorphism

TABLE OF CONTENTS (HIDE)

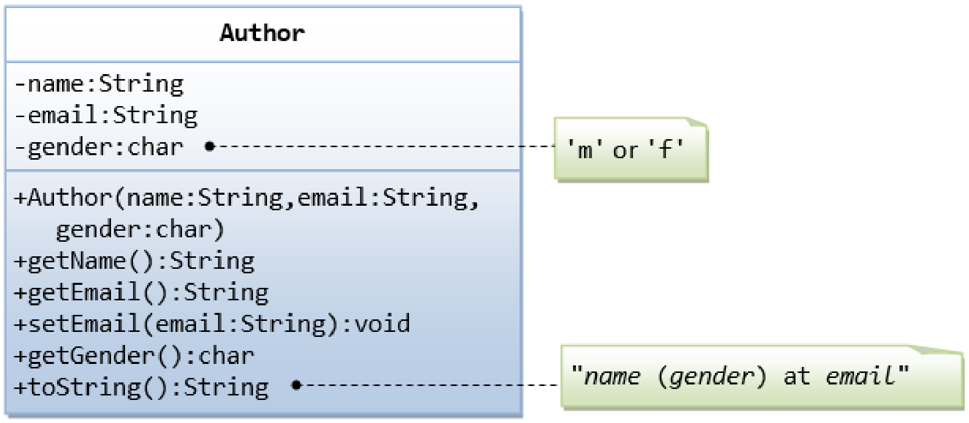
There are two ways to *reuse* existing classes, namely, *composition* and *inheritance*. With *composition* (aka *aggregation*), you define a new class, which is composed of existing classes. With *inheritance*, you derive a new class based on an existing class, with modifications or extensions.

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`".

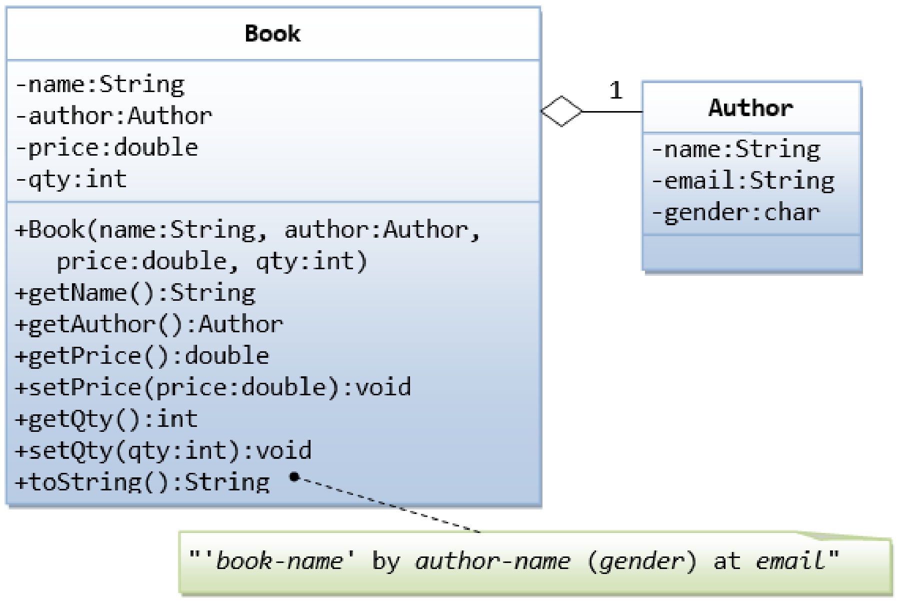
The Author Class (Author.java)

```java
/**
 * The Author class model a book's author.
 */
public class Author {
   // The private instance variables
   private String name;
   private String email;
   private char gender;   // 'm' or 'f'

   /** Constructs a Author instance with the given inputs */
   public Author(String name, String email, char gender) {
      this.name = name;
      this.email = email;
      this.gender = gender;
   }

   // The public getters and setters for the private instance variables.
   // No setter for name and gender as they are not designed to be changed.
   /** Returns the name */
   public String getName() {
      return name;
   }
   /** Returns the gender */
   public char getGender() {
      return gender;
   }
   /** Returns the email */
   public String getEmail() {
      return email;
   }
   /** Sets the email */
   public void setEmail(String email) {
      this.email = email;
   }

   /** Returns a self-descriptive String */
   public String toString() {
      return name + " (" + gender + ") at " + email;
   }
}
```

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

```java
/**
 * A test driver for the Author class.
 */
public class TestAuthor {
   public static void main(String[] args) {
      // Test constructor and toString()
      Author ahTeck = new Author("Tan Ah Teck", "teck@nowhere.com", 'm');
      System.out.println(ahTeck);  // toString()
      //Tan Ah Teck (m) at teck@nowhere.com

      // Test Setters and Getters
      ahTeck.setEmail("teck@somewhere.com");
      System.out.println(ahTeck);  // toString()
      //Tan Ah Teck (m) at teck@somewhere.com
      System.out.println("name is: " + ahTeck.getName());
      //name is: Tan Ah Teck
      System.out.println("gender is: " + ahTeck.getGender());
      //gender is: m
      System.out.println("email is: " + ahTeck.getEmail());
      //email is: teck@somewhere.com
   }
}
```

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)

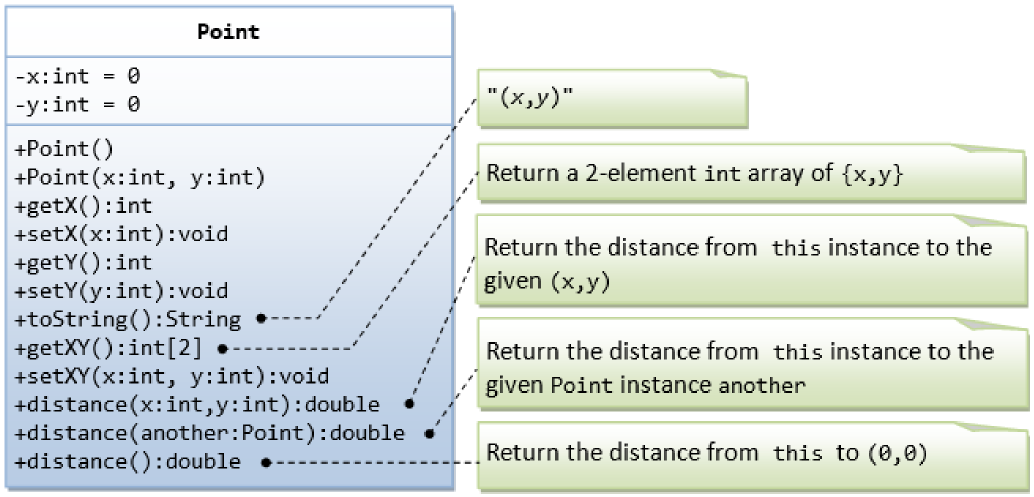
```java
/**
 * The Book class models a book with one (and only one) author.
 */
public class Book {
   // The private instance variables
   private String name;
   private Author author;
   private double price;
   private int qty;

   /** Constructs a Book instance with the given author */
   public Book(String name, Author author, double price, int qty) {
      this.name = name;
      this.author = author;
      this.price = price;
      this.qty = qty;
   }

   // Getters and Setters
   /** Returns the name of this book */
   public String getName() {
      return name;
   }
   /** Return the Author instance of this book */
   public Author getAuthor() {
      return author;  // return member author, which is an instance of the class Author
   }
   /** Returns the price */
   public double getPrice() {
      return price;
   }
   /** Sets the price */
   public void setPrice(double price) {
      this.price = price;
   }
   /** Returns the quantity */
   public int getQty() {
      return qty;
   }
   /** Sets the quantity */
   public void setQty(int qty) {
      this.qty = qty;
   }

   /** Returns a self-descriptive String */
   public String toString() {
      return "'" + name + "' by " + author;  // author.toString()
   }
}
```

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

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

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

      // Test Setters and Getters
      dummyBook.setPrice(8.88);
      dummyBook.setQty(88);
      System.out.println("name is: " + dummyBook.getName());
      //name is: Java for dummies
      System.out.println("price is: " + dummyBook.getPrice());
      //price is: 8.88
      System.out.println("qty is: " + dummyBook.getQty());
      //qty is: 88
      System.out.println("author is: " + dummyBook.getAuthor());  // invoke Author's toString()
      //author is: Tan Ah Teck (m) at ahTeck@somewhere.com
      System.out.println("author's name is: " + dummyBook.getAuthor().getName());
      //author's name is: Tan Ah Teck
      System.out.println("author's email is: " + dummyBook.getAuthor().getEmail());
      //author's email is: ahTeck@somewhere.com
      System.out.println("author's gender is: " + dummyBook.getAuthor().getGender());
      //author's gender is: m

      // Using an anonymous Author instance to create a Book instance
      Book moreDummyBook = new Book("Java for more dummies",
            new Author("Peter Lee", "peter@nowhere.com", 'm'), // an anonymous Author's instance
            19.99, 8);
      System.out.println(moreDummyBook);  // Book's toString()
      //'Java for more dummies' by Peter Lee (m) at peter@nowhere.com
   }
}
```

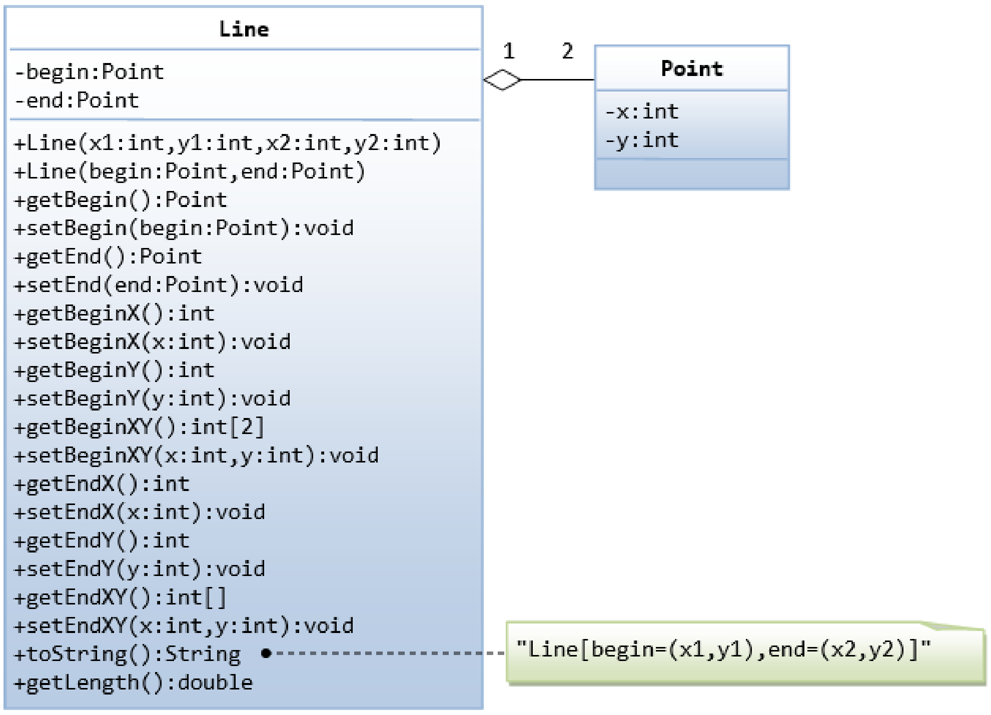
Notes: In this example, I used "name" for Book class instead of "title" to illustrate that you can have a variable name in both the Author and Book classes, but they are distinct.

1.2 Composition EG. 2: The Point and Line Classes


Point
-x:int = 0
-y:int = 0
+Point()
+Point(x:int, y:int)
+getX():int
+setX(x:int):void
+getY():int
+setY(y:int):void
+toString():String
+getXY():int[2]
+setXY(x:int, y:int):void
+distance(x:int,y:int):double
+distance(another:Point):double
+distance():double
"(x,y)"
Return a 2-element int array of {x,y}
Return the distance from this instance to the given (x,y)
Return the distance from this instance to the given Point instance another
Return the distance from this to (0,0)

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. The source code is HERE.

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)

```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, end;   // instances of the "Point" class

   /** Constructs a Line instance given 2 points at (x1, y1) and (x2, y2) */
   public Line(int x1, int y1, int x2, int y2) {
      begin = new Point(x1, y1);  // Construct the instances declared
      end   = new Point(x2, y2);
   }
   /** Construct a Line instance given 2 Point instances */
   public Line(Point begin, Point end) {
      this.begin = begin;  // The caller had constructed the instances
      this.end   = end;
   }

   // The public getter and setter for the private instance variables
   public Point getBegin() {
      return begin;
   }
   public Point getEnd() {
      return end;
   }
   public void setBegin(Point begin) {
      this.begin = begin;
   }
   public void setEnd(Point end) {
      this.end = end;
   }

   public int getBeginX() {
      return begin.getX();  // Point's getX()
   }
   public void setBeginX(int x) {
      begin.setX(x);  // Point's setX()
   }
   public int getBeginY() {
      return begin.getY();  // Point's getY()
   }
   public void setBeginY(int y) {
      begin.setY(y);  // Point's setY()
   }
   public int[] getBeginXY() {
      return begin.getXY();  // Point's getXY()
   }
   public void setBeginXY(int x, int y) {
      begin.setXY(x, y);  // Point's setXY()
   }
   public int getEndX() {
      return end.getX();  // Point's getX()
   }
   public void setEndX(int x) {
      end.setX(x);  // Point's setX()
   }
   public int getEndY() {
      return end.getY();  // Point's getY()
   }
   public void setEndY(int y) {
      end.setY(y);  // Point's setY()
   }
   public int[] getEndXY() {
      return end.getXY();  // Point's getXY()
   }
   public void setEndXY(int x, int y) {
      end.setXY(x, y);  // Point's setXY()
   }

   /** Returns a self-descriptive String */
   public String toString() {
      return "Line[begin=" + begin + ",end=" + end + "]";
            // Invoke begin.toString() and end.toString()
   }

   /** Returns the length of this line */
   public double getLength() {
      return begin.distance(end);  // use Point's distance() method
   }
}
```

A Test Driver for Line Class (`TestLine.java`)

```java
import java.util.Arrays;
/**
 * A Test Driver for the Line class.
 */
public class TestLine {
   public static void main(String[] args) {
      // Test constructor and toString()
      Line l1 = new Line(1, 2, 3, 4);
      System.out.println(l1);  // Line's toString()
      //Line[begin=(1,2),end=(3,4)]
      Line l2 = new Line(new Point(5,6), new Point(7,8));  // anonymous Point's instances
      System.out.println(l2);  // Line's toString()
      //Line[begin=(5,6),end=(7,8)]

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

      l1.setBeginX(21);
      l1.setBeginY(22);
      l1.setEndX(23);
      l1.setEndY(24);
      System.out.println(l1);  // Line's toString()
      //Line[begin=(21,22),end=(23,24)]
      System.out.println("begin's x is: " + l1.getBeginX());
      //begin's x is: 21
      System.out.println("begin's y is: " + l1.getBeginY());
      //begin's y is: 22
      System.out.println("end's x is: " + l1.getEndX());
      //end's x is: 23
      System.out.println("end's y is: " + l1.getEndY());
      //end's y is: 24

      l1.setBeginXY(31, 32);
      l1.setEndXY(33, 34);
      System.out.println(l1);  // Line's toString()
      //Line[begin=(31,32),end=(33,34)]
      System.out.println("begin's x and y are: " + Arrays.toString(l1.getBeginXY()));
      //begin's x and y are: [31, 32]
      System.out.println("end's x and y are: " + Arrays.toString(l1.getEndXY()));
      //end's x and y are: [33, 34]

      // Test getLength()
      System.out.printf("length is: %.2f%n", l1.getLength());
      //length is: 2.83
   }
}
```

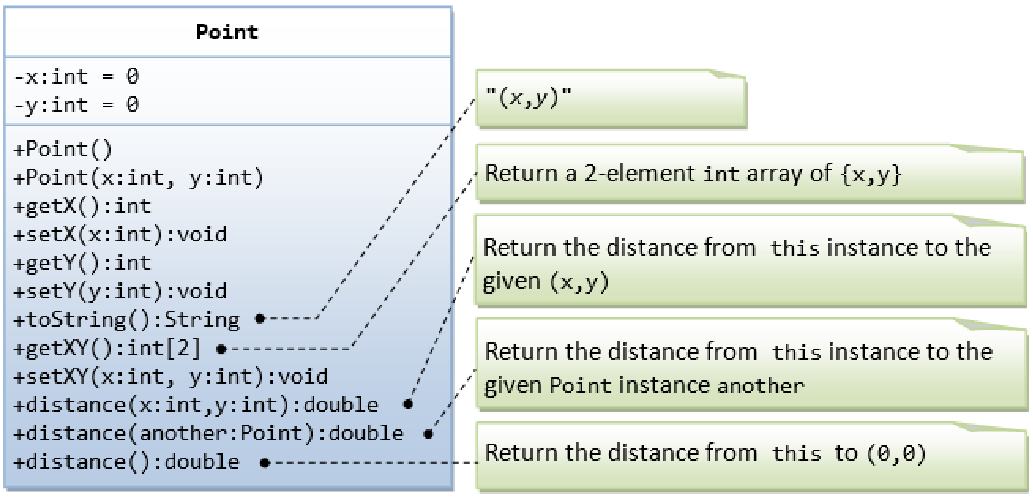
Exercise: Try writing these more complex methods for the Line class:

```java
// Return the gradient of this line in radian (use Math.atan2(y, x)).
public double getGradient()
```

```
// Return the distance from this line to the given point.
public double distance(int x, int y)
public double distance(Point p)

// Return true if this line intersects the given line.
public boolen intersects(Line another)
```

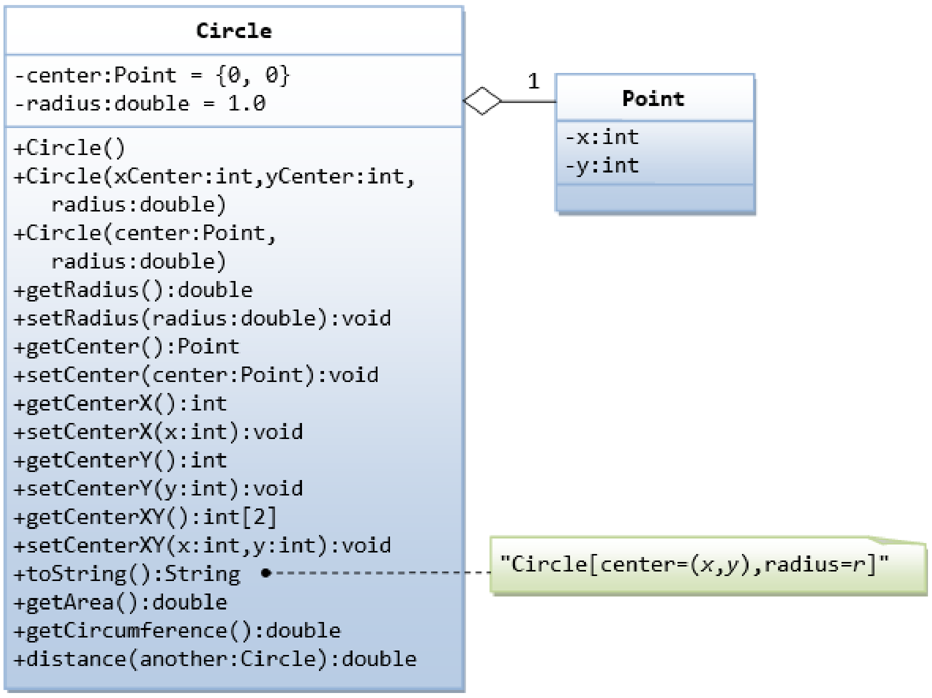
1.3 Composition EG. 3: The Point and Circle Classes

Suppose that we have an *existing* class called `Point`, defined as shown in the class diagram. The source code is HERE.



Point
-x:int = 0
-y:int = 0
+Point()
+Point(x:int, y:int)
+getX():int
+setX(x:int):void
+getY():int
+setY(y:int):void
+toString():String
+getXY():int[2]
+setXY(x:int, y:int):void
+distance(x:int,y:int):double
+distance(another:Point):double
+distance():double
"(x,y)"
Return a 2-element int array of {x,y}
Return the distance from this instance to the given (x,y)
Return the distance from this instance to the given Point instance another
Return the distance from this to (0,0)

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



Circle
-center:Point = {0, 0}
-radius:double = 1.0
+Circle()
+Circle(xCenter:int,yCenter:int, radius:double)
+Circle(center:Point, radius:double)
+getRadius():double
+setRadius(radius:double):void
+getCenter():Point
+setCenter(center:Point):void
+getCenterX():int
+setCenterX(x:int):void
+getCenterY():int
+setCenterY(y:int):void
+getCenterXY():int[2]
+setCenterXY(x:int,y:int):void
+toString():String
+getArea():double
+getCircumference():double
+distance(another:Circle):double
1
Point
-x:int
-y:int
"Circle[center=(x,y),radius=r]"

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`)

```java
/**
 * The Circle class composes a "Point" instance as its center and a radius.
 * We reuse the "Point" class via composition.
 */
public class Circle {
   // The private member variables
   private Point center;  // Declare an instance of the "Point" class
   private double radius;

   // Constructors
   /** Constructs a Circle instance with the default values */
   public Circle() {
      this.center = new Point(); // Construct a Point at (0,0)
      this.radius = 1.0;
   }
   /** Constructs a Circle instance with the center at (xCenter, yCenter) and radius */
   public Circle(int xCenter, int yCenter, double radius) {
      center = new Point(xCenter, yCenter); // Construct a Point at (xCenter,yCenter)
      this.radius = radius;
   }
   /** Constructs a Circle instance with the given Point instance as center and radius */
   public Circle(Point center, double radius) {
      this.center = center;  // The caller had constructed a Point instance
      this.radius = radius;
   }

   // Getters and Setters
   public double getRadius() {
      return this.radius;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public Point getCenter() {
      return this.center;  // return a Point instance
   }
   public void setCenter(Point center) {
      this.center = center;
   }

   public int getCenterX() {
      return center.getX();  // Point's getX()
   }
   public void setCenterX(int x) {
      center.setX(x);  // Point's setX()
   }
   public int getCenterY() {
      return center.getY();  // Point's getY()
   }
   public void setCenterY(int y) {
      center.setY(y);  // Point's setY()
   }
   public int[] getCenterXY() {
      return center.getXY();  // Point's getXY()
   }
   public void setCenterXY(int x, int y) {
      center.setXY(x, y);  // Point's setXY()
   }

   /** Returns a self-descriptive String */
   public String toString() {
      return "Circle[center=" + center + ",radius=" + radius + "]";  // invoke center.toString()
   }

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

   /** Returns the circumference of this circle */
   public double getCircumference() {
      return 2.0 * Math.PI * radius;
   }

   /** Returns the distance from the center of this circle to the center of
       the given Circle instance called another */
   public double distance(Circle another) {
      return center.distance(another.center);  // Use distance() of the Point class
   }
}
```

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

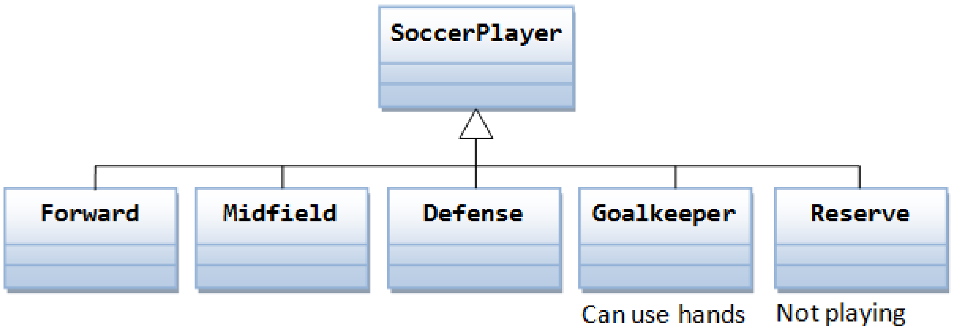
```
/**
 * A test driver for the Circle class.
 */
public class TestCircle {
   public static void main(String[] args) {
      // Test Constructors and toString()
      Circle c1 = new Circle();
      System.out.println(c1);  // Circle's toString()
      Circle c2 = new Circle(1, 2, 3.3);
      System.out.println(c2);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

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

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

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

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

1.4 Exercises

LINK TO EXERCISES

2. Inheritance

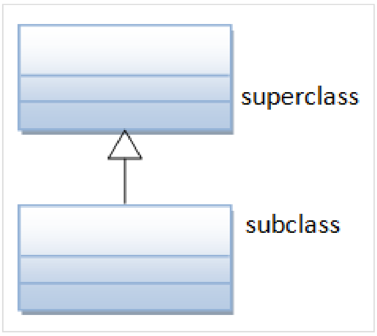
In OOP, we often organize classes in *hierarchy* to *avoid duplication and reduce redundancy*. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a *subclass* (or *derived*, *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.

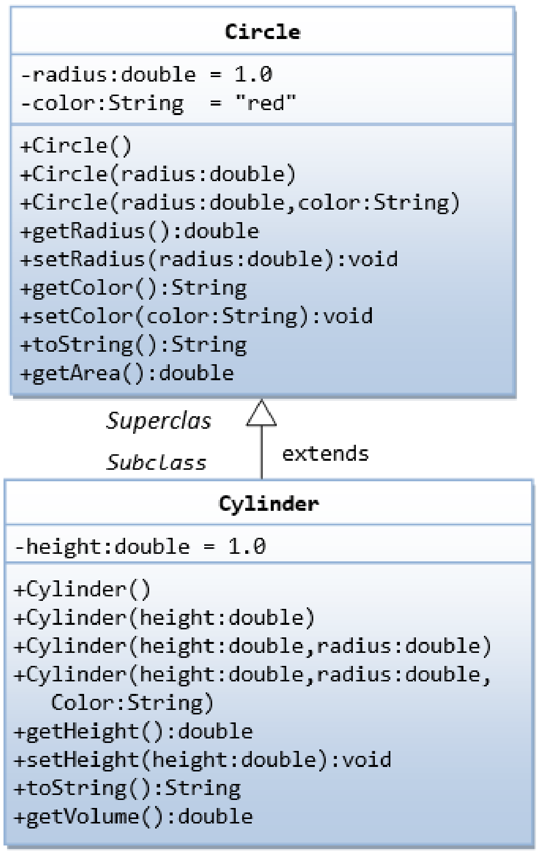
In Java, you define a subclass using the keyword "`extends`", e.g.,

```
class Goalkeeper extends SoccerPlayer {......}
class MyApplet extends java.applet.Applet {.....}
class Cylinder extends Circle {......}
```



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

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)

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

   // Constructors
   public Circle() {
      this.radius = 1.0;
      this.color = "red";
      System.out.println("Construced a Circle with Circle()");  // for debugging
   }
   public Circle(double radius) {
      this.radius = radius;
      this.color = "red";
      System.out.println("Construced a Circle with Circle(radius)");  // for debugging
   }
   public Circle(double radius, String color) {
      this.radius = radius;
      this.color = color;
      System.out.println("Construced a Circle with Circle(radius, color)");  // for debugging
   }

   // public getters and setters for the private variables
   public double getRadius() {
      return this.radius;
   }
   public String getColor() {
      return this.color;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public void setColor(String color) {
      this.color = color;
   }

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

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

Cylinder.java

```
/**
 * A Cylinder is a Circle plus a height.
 */
public class Cylinder extends Circle {
   // private instance variable
   private double height;

   // Constructors
   public Cylinder() {
      super();  // invoke superclass' constructor Circle()
      this.height = 1.0;
      System.out.println("Constructed a Cylinder with Cylinder()");  // for debugging
   }
   public Cylinder(double height) {
      super();  // invoke superclass' constructor Circle()
      this.height = height;
      System.out.println("Constructed a Cylinder with Cylinder(height)");  // for debugging
   }
   public Cylinder(double height, double radius) {
      super(radius);  // invoke superclass' constructor Circle(radius)
      this.height = height;
      System.out.println("Constructed a Cylinder with Cylinder(height, radius)");  // for debugging
   }
   public Cylinder(double height, double radius, String color) {
      super(radius, color);  // invoke superclass' constructor Circle(radius, color)
      this.height = height;
      System.out.println("Constructed a Cylinder with Cylinder(height, radius, color)");  // for debugging
   }

   // Getter and Setter
   public double getHeight() {
      return this.height;
   }
   public void setHeight(double height) {
      this.height = height;
   }

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

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

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

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

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

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:

```java
public class Cylinder extends Circle {
   ......
   // Override the getArea() method inherited from superclass Circle
   @Override
   public double getArea() {
      return 2*Math.PI*getRadius()*height + 2*super.getArea();
   }
   // Need to change the getVolume() as well
   public double getVolume() {
      return super.getArea()*height;   // use superclass' getArea()
   }
   // Override the inherited toString()
   @Override
   public String toString() {
      return "Cylinder[" + super.toString() + ",height=" + height + "]";
   }
}
```

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()` (=`getArea()*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 no 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:

```
// If no constructor is defined in a class, compiler inserts this no-arg constructor
public ClassName () {
   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)`.

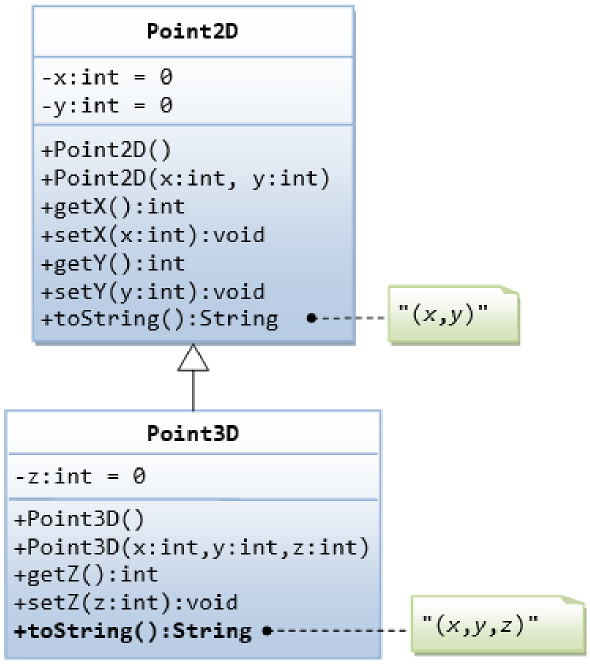
2.7 Single Inheritance

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`

```
/**
 * The Point2D class models a 2D point at (x, y).
 */
public class Point2D {
   // Private instance variables
   private int x, y;

   // Constructors
   /** Construct a Point2D instance at (0,0) */
   public Point2D() {  // default constructor
      this.x = 0;
      this.y = 0;
   }
   /** Constructs a Point2D instance at the given (x,y) */
   public Point2D(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Getters and Setters
   public int getX() {
      return this.x;
   }
   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
      return this.y;
   }
   public void setY(int y) {
      this.y = y;
   }

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

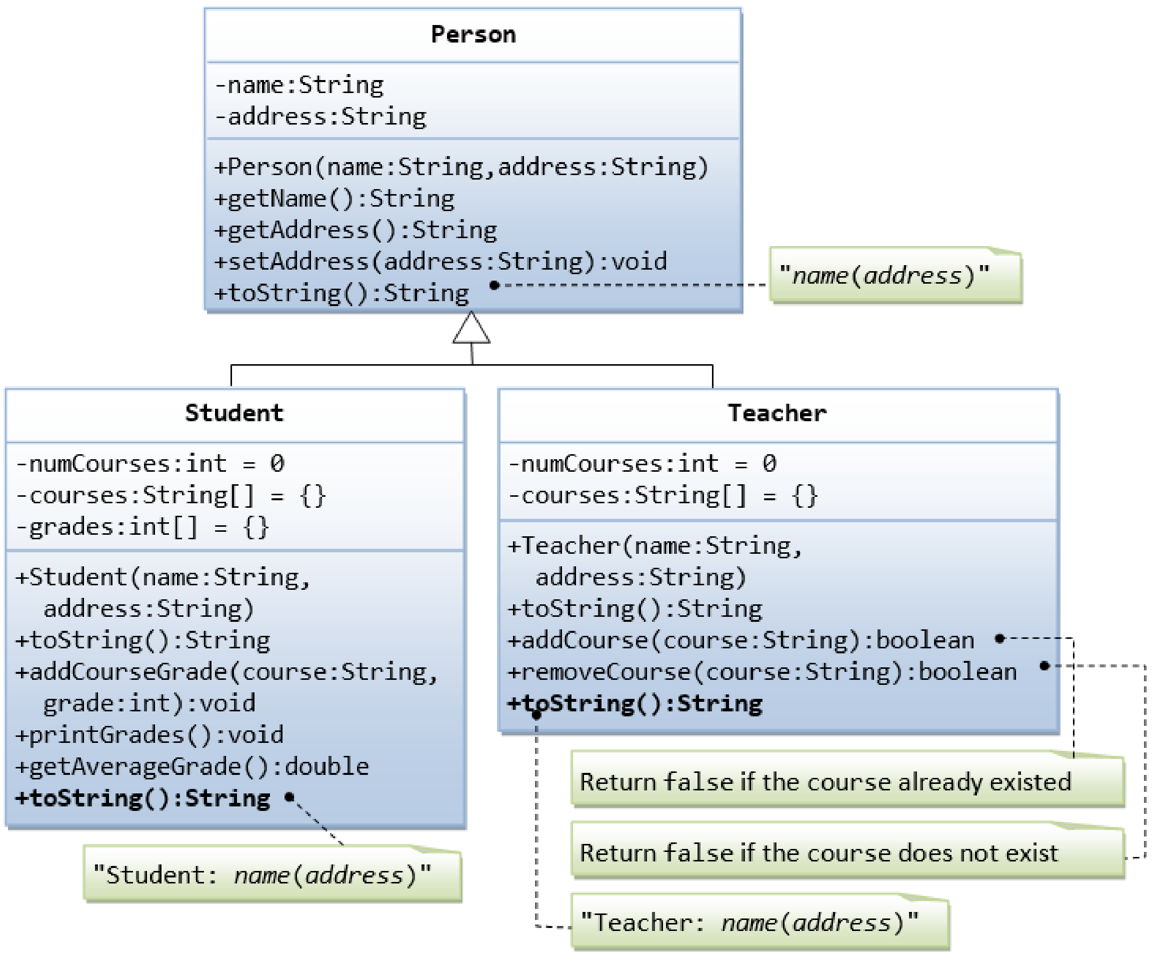
The Subclass `Point3D.java`

```
/**
 * The Point3D class models a 3D point at (x, y,z),
 * which is a subclass of Point2D.
 */
public class Point3D extends Point2D {
   // Private instance variables
   private int z;

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

   // Getters and Setters
   public int getZ() {
      return this.z;
   }
   public void setZ(int z) {
      this.z = z;
   }

   /** Returns a self-descriptive string in the form of "(x,y,z)" */
   @Override
   public String toString() {
      return "(" + super.getX() + "," + super.getY() + "," + this.z + ")";
   }
}
```

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

```
/**
 * A test driver for the Point2D and Point3D classes
 */
public class TestPoint2DPoint3D {
   public static void main(String[] args) {
        /* Test Point2D */
        // Test constructors and toString()
        Point2D p2a = new Point2D(1, 2);
        System.out.println(p2a);  // toString()
        Point2D p2b = new Point2D();  // default constructor
        System.out.println(p2b);
        // Test Setters and Getters
        p2a.setX(3);  // Test setters
        p2a.setY(4);
        System.out.println(p2a);  // toString()
        System.out.println("x is: " + p2a.getX());
        System.out.println("x is: " + p2a.getY());

        /* Test Point3D */
        // Test constructors and toString()
        Point3D p3a = new Point3D(11, 12, 13);
        System.out.println(p3a);  // toString()
        Point2D p3b = new Point3D();  // default constructor
        System.out.println(p3b);
        // Test Setters and Getters
        p3a.setX(21);  // in superclass
        p3a.setY(22);  // in superclass
        p3a.setZ(23);  // in this class
        System.out.println(p3a);  // toString()
        System.out.println("x is: " + p3a.getX());  // in superclass
        System.out.println("y is: " + p3a.getY());  // in superclass
        System.out.println("z is: " + p3a.getZ());  // in this class
    }
}
```

2.10 Inheritance EG. 3: Superclass Person and its Subclasses


Person
-name:String
-address:String
+Person(name:String,address:String)
+getName():String
+getAddress():String
+setAddress(address:String):void
+toString():String
"name(address)"
Student
-numCourses:int = 0
-courses:String[] = {}
-grades:int[] = {}
+Student(name:String, address:String)
+toString():String
+addCourseGrade(course:String, grade:int):void
+printGrades():void
+getAverageGrade():double
+toString():String
"Student: name(address)"
Teacher
-numCourses:int = 0
-courses:String[] = {}
+Teacher(name:String, address:String)
+toString():String
+addCourse(course:String):boolean
+removeCourse(course:String):boolean
+toString():String
Return false if the course already existed
Return false if the course does not exist
"Teacher: name(address)"

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`

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

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

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

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

The Subclass `Student.java`

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

   /** Constructs a Student instance with the given name and address */
   public Student(String name, String address) {
      super(name, address);
      numCourses = 0;
      courses = new String[MAX_COURSES];
      grades = new int[MAX_COURSES];
   }

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

   /** Adds a course and its grade - No input validation */
   public void addCourseGrade(String course, int grade) {
      courses[numCourses] = course;
      grades[numCourses] = grade;
      ++numCourses;
   }
   /** Prints all courses taken and their grade */
   public void printGrades() {
      System.out.print(this);
      for (int i = 0; i < numCourses; ++i) {
         System.out.print(" " + courses[i] + ":" + grades[i]);
      }
      System.out.println();
   }

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

The Subclass Teacher.java

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

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

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

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

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

A Test Driver (`TestPerson.java`)

```
/**
 * A test driver for Person and its subclasses.
 */
public class TestPerson {
   public static void main(String[] args) {
      /* Test Student class */
      Student s1 = new Student("Tan Ah Teck", "1 Happy Ave");
      s1.addCourseGrade("IM101", 97);
      s1.addCourseGrade("IM102", 68);
      s1.printGrades();
      //Student: Tan Ah Teck(1 Happy Ave) IM101:97 IM102:68
      System.out.println("Average is " + s1.getAverageGrade());
      //Average is 82.5

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

2.11 Exercises

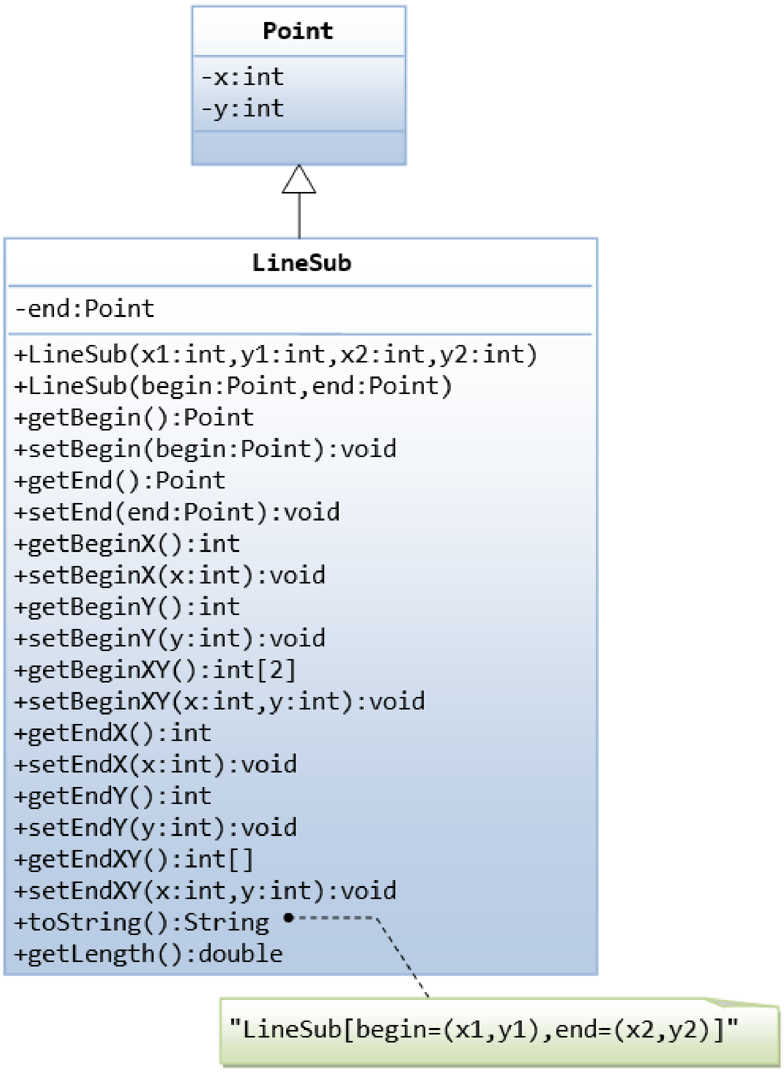
LINK TO EXERCISES

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 Superclass `Point.java`

As above.

The Subclass `LineSub.java`

```java
/**
 * The LineSub class, subclass of Point.
 * It inherits the begin point from the superclass, and adds an end point.
 */
public class LineSub extends Point {  // Inherited the begin point
   // Private instance variables
   Point end;   // Declare end as instance of Point

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

   // Getters and Setters
   public Point getBegin() {
      return this;   // upcast to Point (polymorphism - to be discussed later)
   }
   public Point getEnd() {
      return end;
   }
   public void setBegin(Point begin) {
      super.setX(begin.getX());
      super.setY(begin.getY());
   }
   public void setEnd(Point end) {
      this.end = end;
   }

   // Other Get and Set methods
   public int getBeginX() {
      return super.getX();  // inherited, super is optional
   }
   public void setBeginX(int x) {
      super.setX(x);        // inherited, super is optional
   }
   public int getBeginY() {
      return super.getY();
   }
   public void setBeginY(int y) {
      super.setY(y);
   }
   public int[] getBeginXY() {
      return super.getXY();
   }
   public void setBeginXY(int x, int y) {
      super.setXY(x, y);
   }
   public int getEndX() {
      return end.getX();
   }
   public void setEndX(int x) {
      end.setX(x);
   }
   public int getEndY() {
      return end.getY();
   }
   public void setEndY(int y) {
      end.setY(y);
   }
   public int[] getEndXY() {
      return end.getXY();
   }
   public void setEndXY(int x, int y) {
      end.setXY(x, y);
   }

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

   /** Returns the length of this line */
   public double getLength() {
      return super.distance(end);
   }
}
```

A Test Driver (`TestLineSub.java`)

```java
/**
 * Test Driver for the LineSub class
 */
public class TestLineSub {
   public static void main(String[] args) {
      // Test constructors and toString()
      LineSub l1 = new LineSub(1, 2, 3, 4);
      System.out.println(l1);  // toString()
      LineSub l2 = new LineSub(new Point(5,6), new Point(7,8));
      System.out.println(l2);

      // Test Setters and Getters
      l1.setBegin(new Point(11, 12));
      l1.setEnd(new Point(13, 14));
      System.out.println(l1);  // toString()
      System.out.println("begin is: " + l1.getBegin());
      System.out.println("end is: " + l1.getEnd());

      l1.setBeginX(21);
      l1.setBeginY(22);
      l1.setEndX(23);
      l1.setEndY(24);
      System.out.println(l1);
      System.out.println(l1);  // toString()
      System.out.println("begin's x is: " + l1.getBeginX());
      System.out.println("begin's y is: " + l1.getBeginY());
      System.out.println("end's x is: " + l1.getEndX());
      System.out.println("end's y is: " + l1.getEndY());

      l1.setBeginXY(31, 32);
      l1.setEndXY(33, 34);
      System.out.println(l1);  // toString()
      System.out.println("begin's x is: " + l1.getBeginXY()[0]);
      System.out.println("begin's y is: " + l1.getBeginXY()[1]);
      System.out.println("end's x is: " + l1.getEndXY()[0]);
      System.out.println("end's y is: " + l1.getEndXY()[1]);

      // Test getLength()
      System.out.printf("length is: %.2f%n", l1.getLength());
   }
}
```

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.

3.2 Exercises

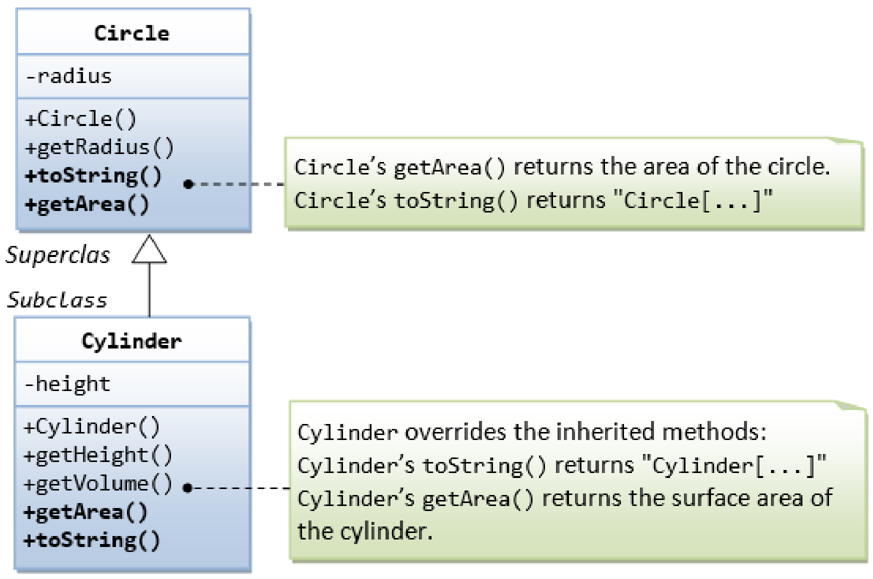
LINK TO EXERCISES ON COMPOSITION VS INHERITANCE

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 it 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*.


Circle
-radius
+Circle()
+getRadius()
+toString()
+getArea()
Circle's getArea() returns the area of the circle.
Circle's toString() returns "Circle[...]"
Superclas
Subclass
Cylinder
-height
+Cylinder()
+getHeight()
+getVolume()
+getArea()
+toString()
Cylinder overrides the inherited methods:
Cylinder's toString() returns "Cylinder[...]"
Cylinder's getArea() returns the surface area of the cylinder.

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

```
/** The superclass Circle */
public class Circle {
   // private instance variable
   private double radius;
   /** Constructs a Circle instanve with the given radius */
   public Circle(double radius) {
      this.radius = radius;
   }
   /** Returns the radius */
   public double getRadius() {
      return this.radius;
   }
   /** Returns the area of this circle */
   public double getArea() {
      return radius * radius * Math.PI;
   }
   /** Returns a self-descriptive string */
   public String toString() {
      return "Circle[radius=" + radius + "]";
   }
}
```

Cylinder.java

```
/** The subclass Cylinder */
public class Cylinder extends Circle {
   // private instance variable
   private double height;
   /** Constructs a Cylinder instance with the given height and radius */
   public Cylinder(double height, double radius) {
      super(radius);
      this.height = height;
   }
   /** Returns the height */
   public double getHeight() {
      return this.height;
   }
   /** Returns the volume of this cylinder */
   public double getVolumne() {
      return super.getArea() * height;
   }
   /** Overrides the inherited method to return the surface area */
   @Override
   public double getArea() {
      return 2.0 * Math.PI * getRadius() * height;
   }
   /** Override the inherited method to describe itself */
   @Override
   public String toString() {
      return "Cylinder[height=" + height + "," + super.toString() + "]";
   }
}
```

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
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
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.

```
// CANNOT invoke method in Cylinder as c1 is a Circle reference
c1.getHeight();
//compilation error: cannot find symbol method getHeight()
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
//Cylinder[height=1.1,Circle[radius=2.2]]
System.out.println(c1.getArea());   // Run the overridden version
//15.205308443374602
```

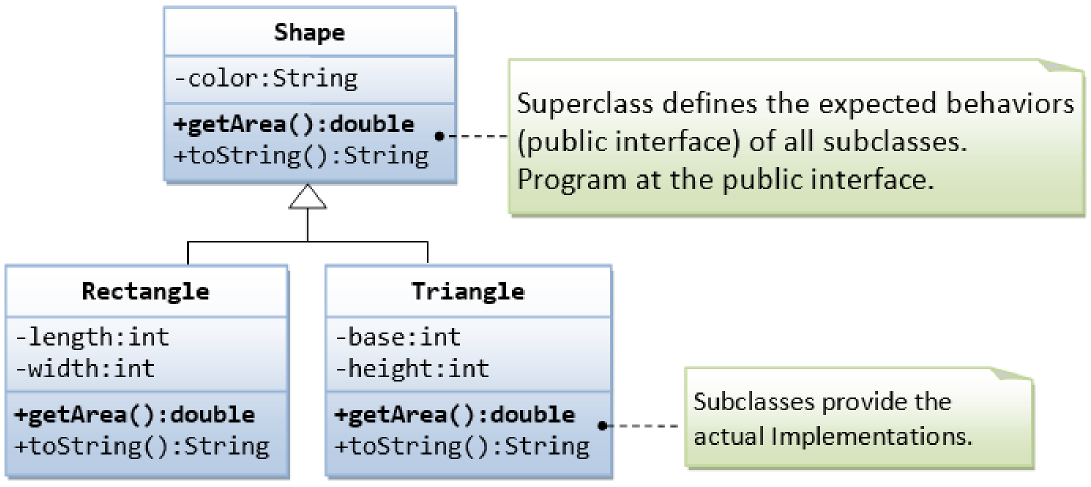
Summary

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`

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

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

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

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

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`

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

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

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

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

Subclass `Triangle.java`

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

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

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

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

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:

```
/**
 * A test driver for Shape and its subclasses
 */
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);  // Upcast
      System.out.println(s1);  // Run Rectangle's toString()
      //Rectangle[length=4,width=5,Shape[color=red]]
      System.out.println("Area is " + s1.getArea());  // Run Rectangle's getArea()
      //Area is 20.0

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

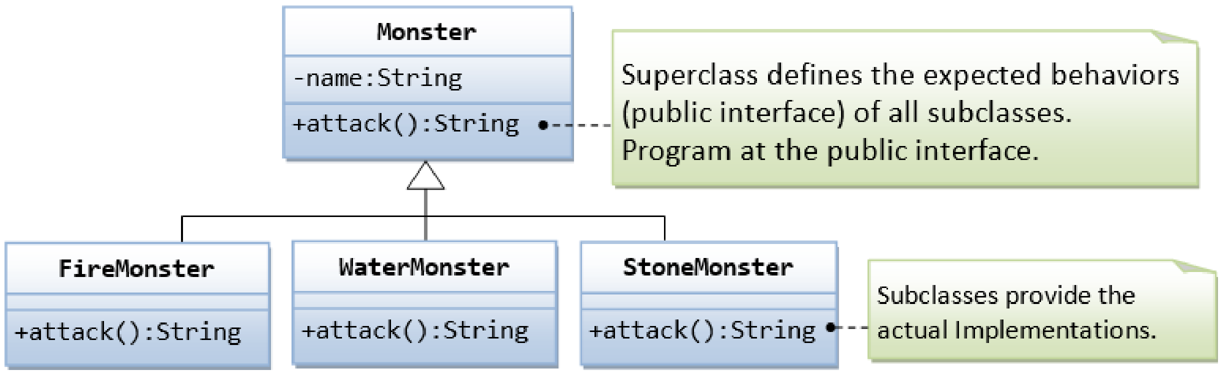
The beauty of this code is that *all the references are from the superclass* (i.e., *programming at the interface level*). You could instantiate different subclass instance, and the code still works. You could extend your program easily by adding in more subclasses, such as `Circle`, Square, etc, with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

```
public class TestShape {
   public static void main(String[] args) {
      // Constructing a Shape instance poses problem!
      Shape s3 = new Shape("green");
      System.out.println(s3);
      //Shape[color=green]
      System.out.println("Area is " + s3.getArea());  // Invalid output
      //Shape unknown! Cannot compute area!
      //Area is 0.0
   }
}
```

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


Monster
-name:String
+attack():String
Superclass defines the expected behaviors (public interface) of all subclasses. Program at the public interface.
FireMonster
+attack():String
WaterMonster
+attack():String
StoneMonster
+attack():String
Subclasses provide the actual Implementations.

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 provides 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`

```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`

```java
public class FireMonster extends Monster {
   /** Constructs a FireMonster with the given name */
   public FireMonster(String name) {
      super(name);
   }
   /** Subclass provides actual implementation for attack() */
   @Override
   public String attack() {
      return "Attack with fire!";
   }
}
```

Subclass `WaterMonster.java`

```java
public class WaterMonster extends Monster {
   /** Constructs a WaterMonster instance with the given name */
   public WaterMonster(String name) {
      super(name);
   }
   /** Subclass provides actual implementation for attack() */
   @Override
   public String attack() {
      return "Attack with water!";
   }
}
```

Subclass `StoneMonster.java`

```
public class StoneMonster extends Monster {
   /** Constructs a StoneMonster instance with the given name */
   public StoneMonster(String name) {
      super(name);
   }
   /** Subclass provides actual implementation for attack() */
   @Override
   public String attack() {
      return "Attack with stones!";
   }
}
```

A Test Driver `TestMonster.java`

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

      // Invoke the actual implementation
      System.out.println(m1.attack());  // Run FireMonster's attack()
      //Attack with fire!
      System.out.println(m2.attack());  // Run WaterMonster's attack()
      //Attack with water!
      System.out.println(m3.attack());  // Run StoneMonster's attack()
      //Attack with stones!

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

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

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,

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

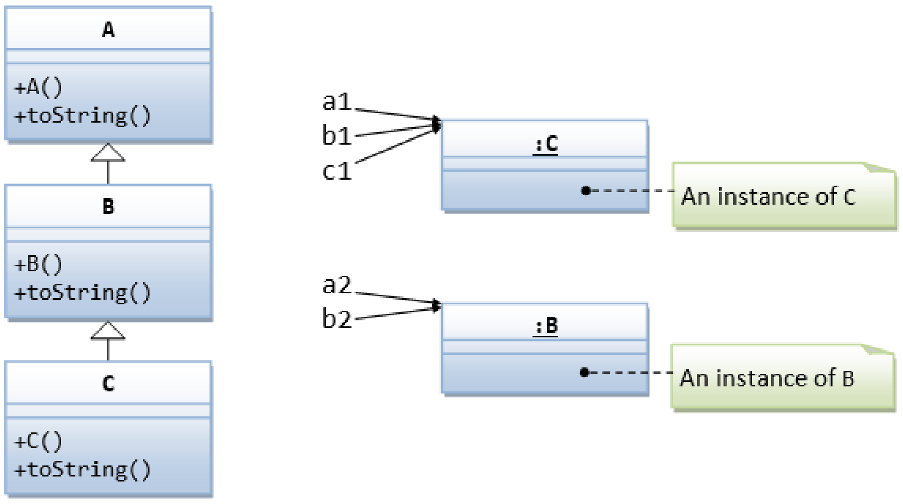
Downcasting a Substituted Reference to Its Original Class

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

```
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



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

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

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

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

```
public class TestCasting {
   public static void main(String[] args) {
      A a1 = new C();   // upcast
      //Constructed an instance of A
      //Constructed an instance of B
      //Constructed an instance of C
      System.out.println(a1);  // run C's toString()
      //This is C
      B b1 = (B)a1;     // downcast okay
      System.out.println(b1);  // run C's toString()
      //This is C
      C c1 = (C)b1;     // downcast okay
      System.out.println(c1);  // run C's toString()
      //This is C

      A a2 = new B();  // upcast
      //Constructed an instance of A
      //Constructed an instance of B
      System.out.println(a2);  // run B's toString()
      //This is B
      B b2 = (B)a2;    // downcast okay
      C c2 = (C)a2;
      //compilation okay, but runtime error:
      //java.lang.ClassCastException: class B cannot be cast to class C
   }
}
```

Casting Operator

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

```
Circle c1 = new Circle(5);
Point p1 = new Point();

c1 = p1;          //compilation error: incompatible types (Point is not a subclass of Circle)
c1 = (Circle)p1;  //runtime error: java.lang.ClassCastException: 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
```

```
Circle c1 = new Circle();
System.out.println(c1 instanceof Circle);  // true

if (c1 instanceof Circle) { ...... }
```

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

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

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

4.7 Exercises

LINK TO EXERCISES

5. Abstract Classes & Interfaces

5.1 The abstract Method and abstract class

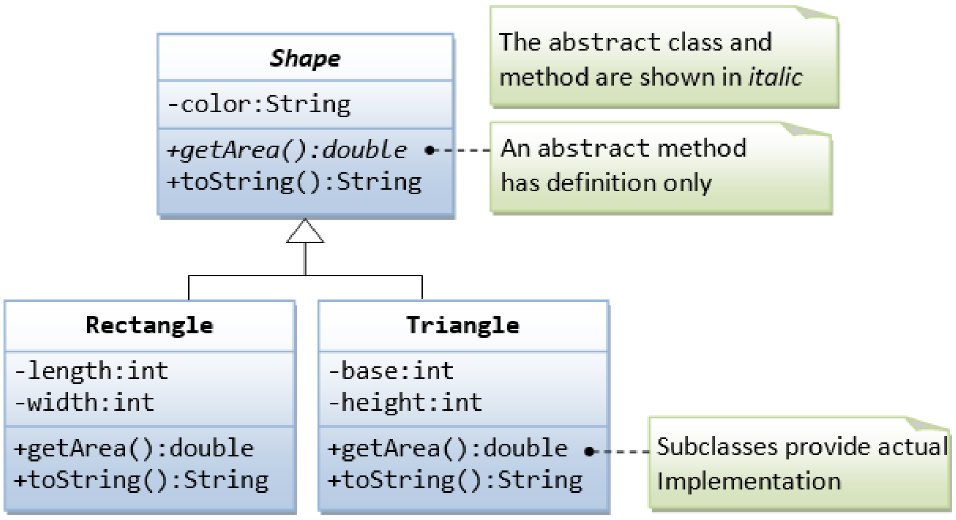
In the above examples of Shape and `Monster`, we encountered a problem when we create instances of Shape and `Monster` and run the `getArea()` or `attack()`. This can be resolved via `abstract` method and `abstract` class.

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:

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

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*.

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`

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

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

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

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

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,

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

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

      // Cannot create instance of an abstract class
      Shape s3 = new Shape("green");
      //compilation error: Shape is abstract; cannot be instantiated
   }
}
```

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, you 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 method 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).

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

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

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

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.

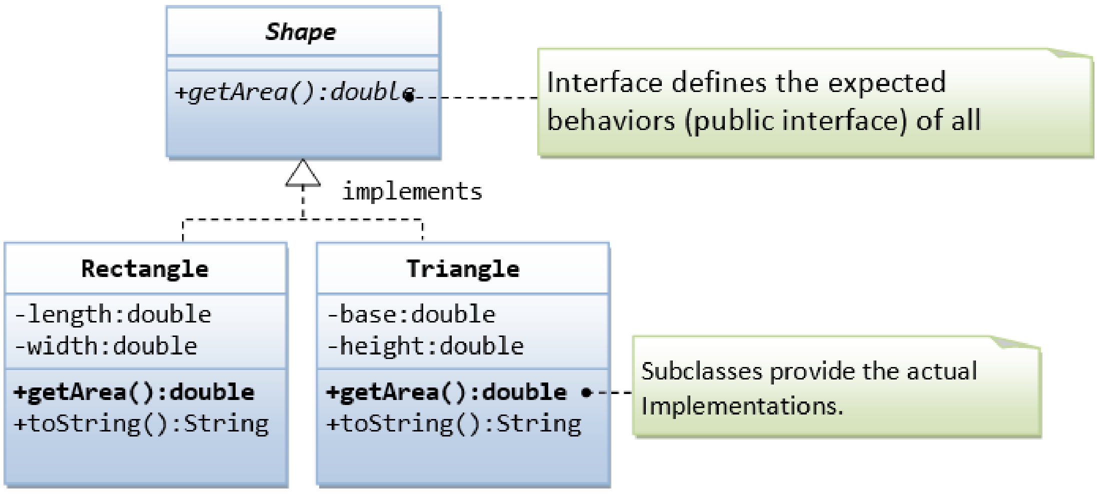
An interface provides a *form*, a *protocol*, a *standard*, a *contract*, a *specification*, a set of *rules*, an *interface*, for all objects that implement it. It is a *specification* and *rules* that any object implementing it agrees to follow.

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`, `Extenalizable`, `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.

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

```
/**
 * 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, 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;
   }
}
```

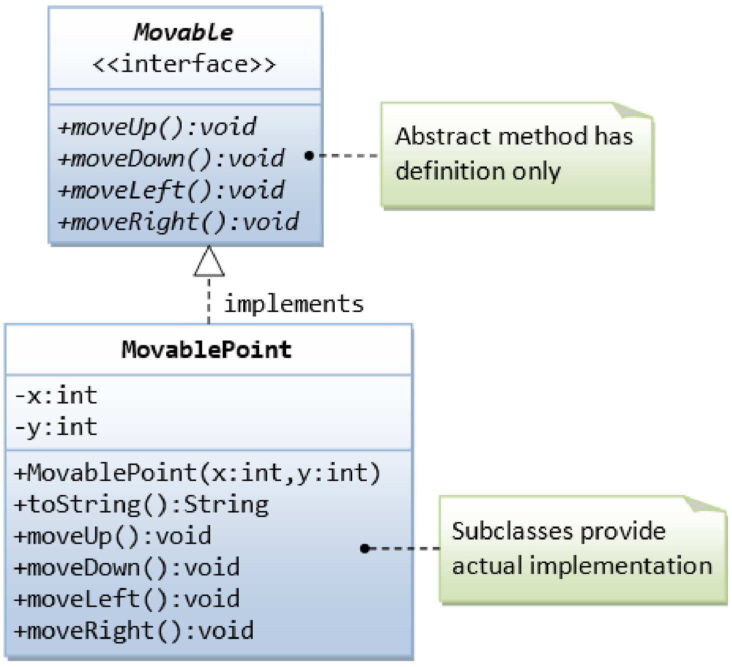
```
/**
 * The subclass Triangle need to implement all the abstract methods in Shape
 */
public class Triangle implements Shape {
   // Private member variables
   private int base, height;

   /** Constructs a Triangle instance with the given base and height */
   public Triangle(int base, int height) {
      this.base = base;
      this.height = height;
   }

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

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

A test driver is as follows:

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle(1, 2);  // upcast
      System.out.println(s1);
      //Rectangle[length=1,width=2]
      System.out.println("Area is " + s1.getArea());
      //Area is 2.0

      Shape s2 = new Triangle(3, 4);  // upcast
      System.out.println(s2);
      //Triangle[base=3,height=4]
      System.out.println("Area is " + s2.getArea());
      //Area is 6.0

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

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`

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

```
/**
 * The subclass MovablePoint needs to implement all the abstract methods
 *   defined in the interface Movable
 */
public class MovablePoint implements Movable {
   // Private member variables
   private int x, y;   // x and y coordinates of the point

   /** Constructs a MovablePoint instance at the given x and y */
   public MovablePoint(int x, int y) {
      this.x = x;
      this.y = y;
   }

   /** Returns a self-descriptive string */
   @Override
   public String toString() {
      return "(" + x + "," + y + ")";
   }

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

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.

```
public class TestMovable {
   public static void main(String[] args) {
      MovablePoint p1 = new MovablePoint(1, 2);
      System.out.println(p1);
      //(1,2)
      p1.moveDown();
      System.out.println(p1);
      //(1,3)
      p1.moveRight();
      System.out.println(p1);
      //(2,3)

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

      MovablePoint p3 = (MovablePoint)p2;   // downcast
      System.out.println(p3);
      //(3,3)
   }
}
```

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,

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

5.8 interface Formal Syntax

The formal syntax for declaring interface is:

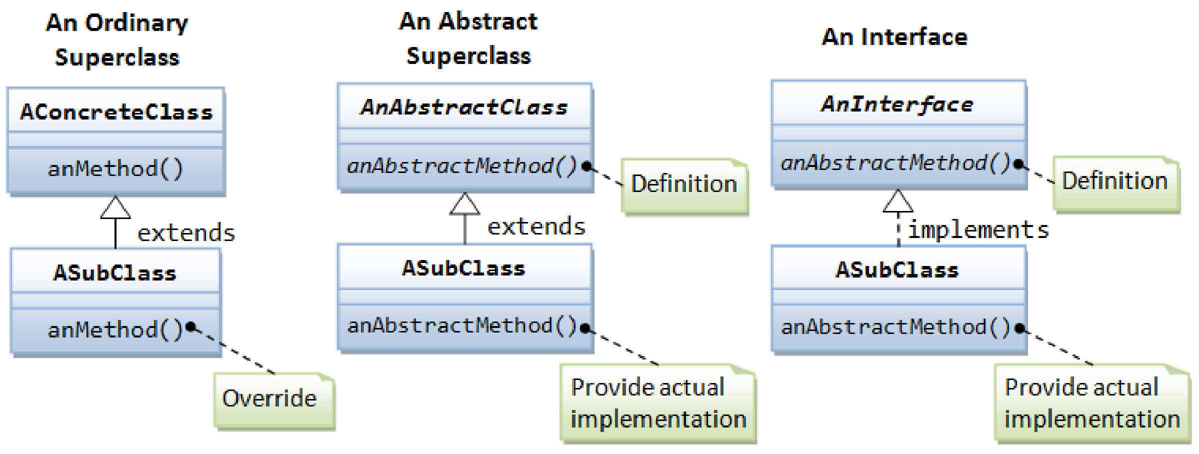
```
[public|protected|package] interface interfaceName
[extends superInterfaceName] {
   // constants
   static final ...;

   // public abstract methods' signature
   ...
}
```

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.

As an example, Java's thread can be built using interface `Runnable` or superclass `Thread`.

5.11 Exercises

LINK TO EXERCISES ON POLYMORPHISM, ABSTRACT CLASSES AND INTERFACES

5.12 (Advanced) 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 depends 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. We shall discuss this problem and generics in details in the later chapter.

5.13 Exercises

LINK TO EXERCISES

6. (Advanced) 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).

[TODO] 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.

The benefit of tight encapsulation out-weights the overhead needed in additional method calls.

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.

[TODO] 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.

[TODO] 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

LINK TO JAVA REFERENCES & RESOURCES

Latest version tested: JDK 1.13.0
Last modified: February, 2020