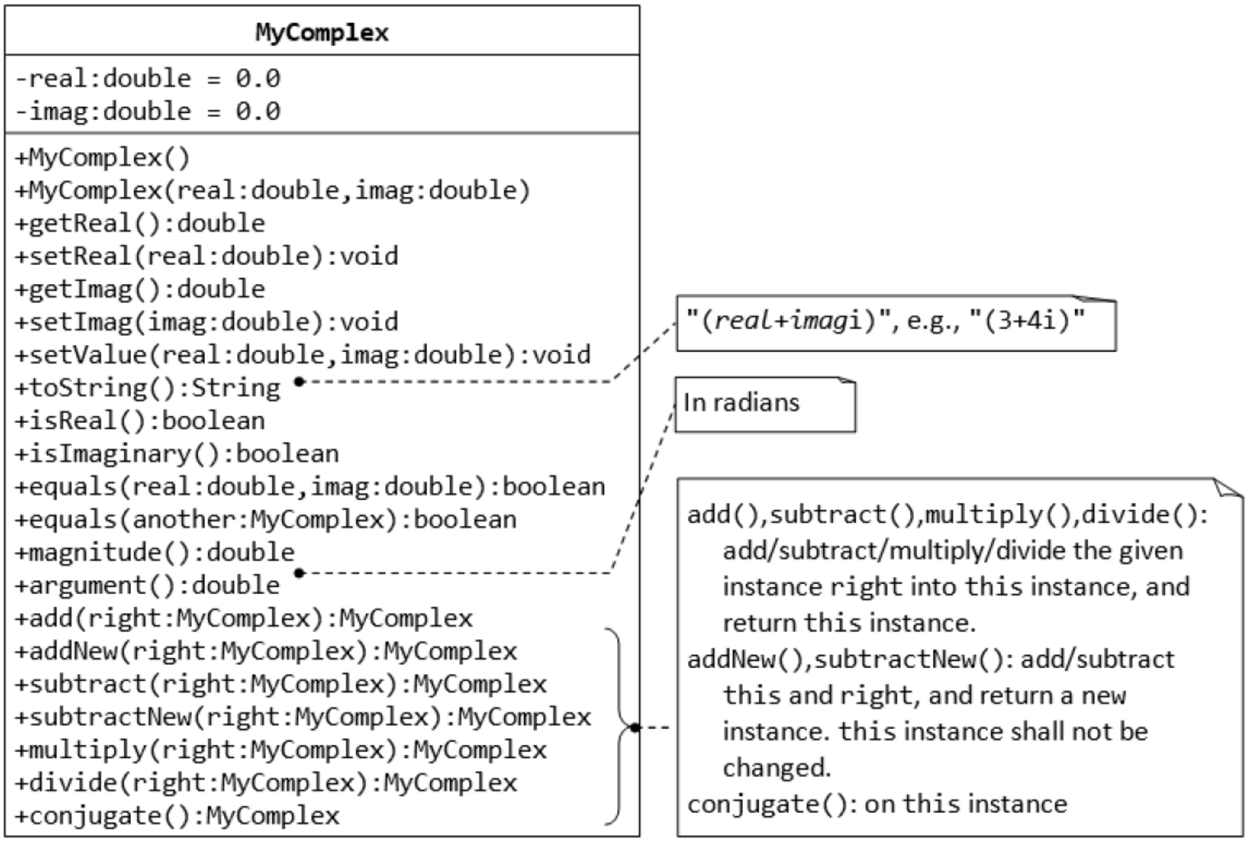
CSC 302: OBJECT ORIENTED PROGRAMMING

LAB EXERCISE 2

Muhammad Salisu Ali
ms.ali@fud.edu.ng

3. More Exercises on Classes

3.1 The MyComplex class



A class called `MyComplex`, which models complex numbers `x+yi`, is designed as shown in the above class diagram. It contains:

- Two instance variable named `real(double)` and `imag(double)` which stores the real and imaginary parts of the complex number, respectively.
- A constructor that creates a `MyComplex` instance with the given real and imaginary values.
- A default constructor that create a `MyComplex` at `0.0 + 0.0i`.
- Getters and setters for instance variables `real` and `imag`.
- A method `setValue()` to set the value of the complex number.
- A `toString()` that returns "`(x + yi)`" where `x` and `y` are the real and imaginary parts, respectively.
- Methods `isReal()` and `isImaginary()` that returns true if this complex number is `real` or `imaginary`, respectively.

Hints:

```
return (imag == 0);
```

- A method `equals(double real, double imag)` that returns `true` if this complex number is equal to the given complex number (real, imag).

Hints:

```
return (this.real == real && this.imag == imag);
```

- An `overloaded equals(MyComplex another)` that returns `true` if this complex number is equal to the given `MyComplex` instance `another`.

Hints:

```
return (this.real == another.real && this.imag == another.imag);
```

- A method `magnitude()` that returns the magnitude of this complex number.

```
magnitude(x+yi) = Math.sqrt(x*x + y*y)
```

- Methods `argument()` that returns the argument of this complex number in `radians` `(double)`.

```
arg(x+yi) = Math.atan2(y, x) (in radians)
```

Note: The Math library has two arc-tangent methods, `Math.atan(double)` and `Math.atan2(double, double)`.

- We commonly use the `Math.atan2(y, x)` instead of `Math.atan(y/x)` to avoid division by zero. Read the documentation of `Math` class in package `java.lang`.
- Methods `add(MyComplex right)` and `subtract(MyComplex right)` that adds and subtract the given `MyComplex` instance (called `right`), into/from `this` instance and returns `this` instance.

```
(a + bi) + (c + di) = (a+c) + (b+d)i
(a + bi) - (c + di) = (a-c) + (b-d)i
```

Hints:

```
return this;  // return "this" instance
```

- Methods `addNew(MyComplex right)` and `subtractNew(MyComplex right)` that adds and subtract `this` instance with the given `MyComplex` instance called `right`, and returns a new `MyComplex` instance containing the result.

Hint:

```
// construct a new instance and return the constructed instance
return new MyComplex(..., ...);
```

- Methods `multiply(MyComplex right)` and `divide(MyComplex right)` that multiplies and divides this instance with the given `MyComplex` instance `right`, and keeps the result in `this` instance, and returns this instance.

```
(a + bi) * (c + di) = (ac - bd) + (ad + bc)i
(a + bi) / (c + di) = [(a + bi) * (c - di)] / (c*c + d*d)
```

- A method `conjugate()` that operates on `this` instance and returns `this` instance containing the `complex conjugate`.

```
conjugate(x+yi) = x - yi
```

You are required to:

1. Write the `MyComplex` class.
2. Write a test driver to test all the public methods defined in the class.
3. Write an application called `MyComplexApp` that uses the `MyComplex` class. The application shall prompt the user for two complex numbers, print their values, check for real, imaginary and equality, and carry out all the arithmetic operations.

```
Enter complex number 1 (real and imaginary part): 1.1 2.2
Enter complex number 2 (real and imaginary part): 3.3 4.4

Number 1 is: (1.1 + 2.2i)
(1.1 + 2.2i) is NOT a pure real number
(1.1 + 2.2i) is NOT a pure imaginary number

Number 2 is: (3.3 + 4.4i)
```

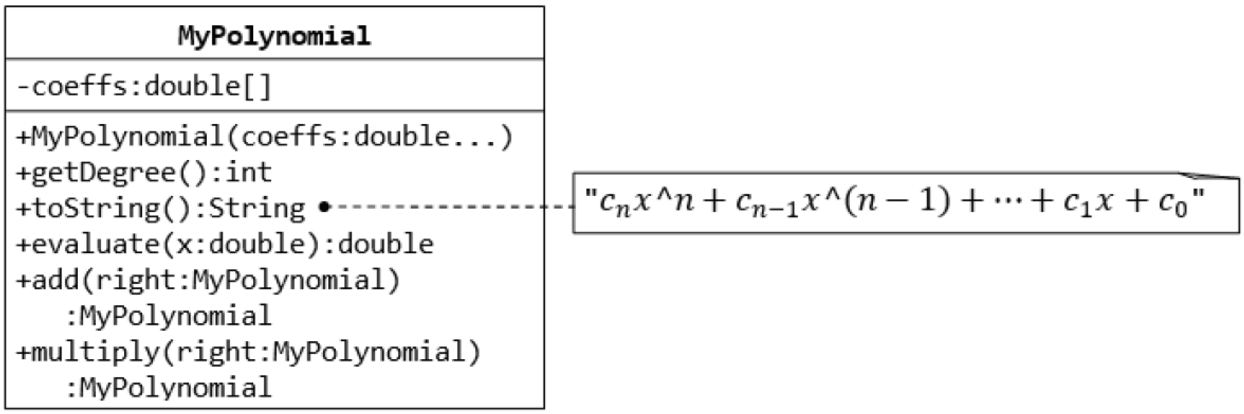
```
(3.3 + 4.4i) is NOT a pure real number
(3.3 + 4.4i) is NOT a pure imaginary number

(1.1 + 2.2i) is NOT equal to (3.3 + 4.4i)
(1.1 + 2.2i) + (3.3 + 4.4i) = (4.4 + 6.6000000000000005i)
(1.1 + 2.2i) - (3.3 + 4.4i) = (-2.1999999999999997 + -2.2i)
```

Take note that there are a few flaws in the design of this class, which was introduced solely for teaching purpose:

- Comparing `doubles` in `equal()` using "==" may produce unexpected outcome. For example, `(2.2+4.4) == 6.6` returns `false`. It is common to define a small threshold called `EPSILON` (set to about `10^-8`) for comparing floating point numbers.
- The method `addNew()`, `subtractNew()` produce new instances, whereas `add()`, `subtract()`, `multiply()`, `divide()` and `conjugate()` modify this instance. There is inconsistency in the design (introduced for teaching purpose).

Also take note that methods such as `add()` returns an instance of `MyComplex`. Hence, you can place the result inside a `System.out.println()` (which implicitly invoke the `toString()`). You can also chain the operations, e.g., `c1.add(c2).add(c3)` (same `a(c1.add(c2)).add(c3))`, or `c1.add(c2).subtract(c3)`.

3.2 The MyPolynomial Class



A class called `MyPolynomial`, which models polynomials of degree-n (see equation), is designed as shown in the class diagram.

$$c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$$

It contains:

- An instance variable named `coeffs`, which stores the coefficients of the n-degree polynomial in a double array of size `n+1`, where `c0` is kept at index `0`.
- A constructor `MyPolynomial(coeffs:double...)` that takes a variable number of doubles to initialize the `coeffs` array, where the first argument corresponds to `c0`.
- The three dots is known as `varargs` (variable number of arguments), which is a new feature introduced in JDK 1.5. It accepts an array or a sequence of comma-separated arguments. The compiler automatically packs the comma separated arguments in an array. The three dots can only be used for the last argument of the method.

Hints:

```
public class MyPolynomial {
      private double[] coeffs;
      public MyPolynomial(double... coeffs) { // varargs
            this.coeffs = coeffs; // varargs is treated as array
      }
      ......
```

```
}
// Test program
// Can invoke with a variable number of arguments
MyPolynomial p1 = new MyPolynomial(1.1, 2.2, 3.3);
MyPolynomial p1 = new MyPolynomial(1.1, 2.2, 3.3, 4.4, 5.5);
// Can also invoke with an array
Double coeffs = {1.2, 3.4, 5.6, 7.8}
MyPolynomial p2 = new MyPolynomial(coeffs);
```

- A method `getDegree()` that returns the degree of this polynomial.
- A method `toString()` that returns "cnx^n+cn-1x^(n-1)+...+c1x+c0".
- A method `evaluate(double x)` that evaluate the polynomial for the given `x`, by substituting the given `x` into the polynomial expression.
- Methods `add()` and `multiply()` that adds and multiplies this polynomial with the given `MyPolynomial` instance another, and returns `this` instance that contains the result.

Write the `MyPolynomial` class. Also write a test driver (called `TestMyPolynomial`) to test all the *public* methods defined in the class.

Question: Do you need to keep the degree of the polynomial as an instance variable in the `MyPolynomial` class in Java?

How about C/C++? Why?

3.3 Using JDK's BigInteger Class

Recall that primitive integer type `byte, short, int` and `long` represent 8-, 16-, 32-, and 64-bit signed integers, respectively. You cannot use them for integers bigger than 64 bits. Java API provides a class called `BigInteger` in a package called `java.math`. Study the API of the `BigInteger` class (Java API ⇒ From "Packages", choose "`java.math`" " From "classes", choose "`BigInteger`" " Study the constructors (choose "CONSTR") on how to construct a `BigInteger` instance, and the public methods available (choose "METHOD"). Look for methods for adding and multiplying two `BigIntegers`.

Write a program called `TestBigInteger` that:

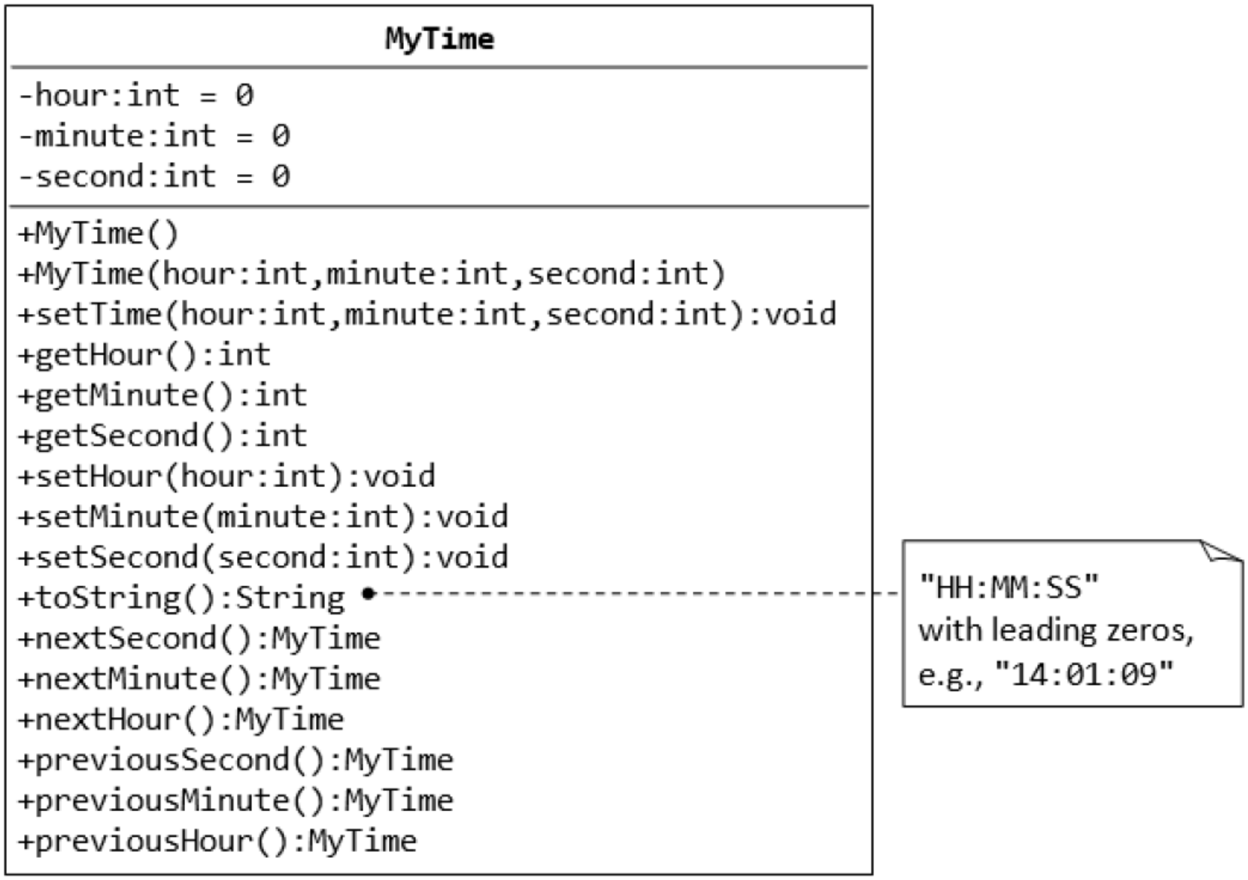
1. adds "11111111111111111111111111111111111111111111111111111111111111" to "22222222222222222222222222222222222222222222222222" and prints the result.

2. Multiplies the above two number and prints the result.

Hints:

```
import java.math.BigInteger
public class TestBigInteger {
   public static void main(String[] args) {
      BigInteger i1 = new BigInteger(...);
      BigInteger i2 = new BigInteger(...);
      System.out.println(i1.add(i2));
      .......
   }
}
```

3.4 The MyTime Class



A class called `MyTime`, which models a time instance, is designed as shown in the class diagram. It contains the following `private` instance variables:

- `hour`: between 0 to 23.
- `minute`: between 0 to 59.
- `Second`: between 0 to 59.

You are required to perform input validation.

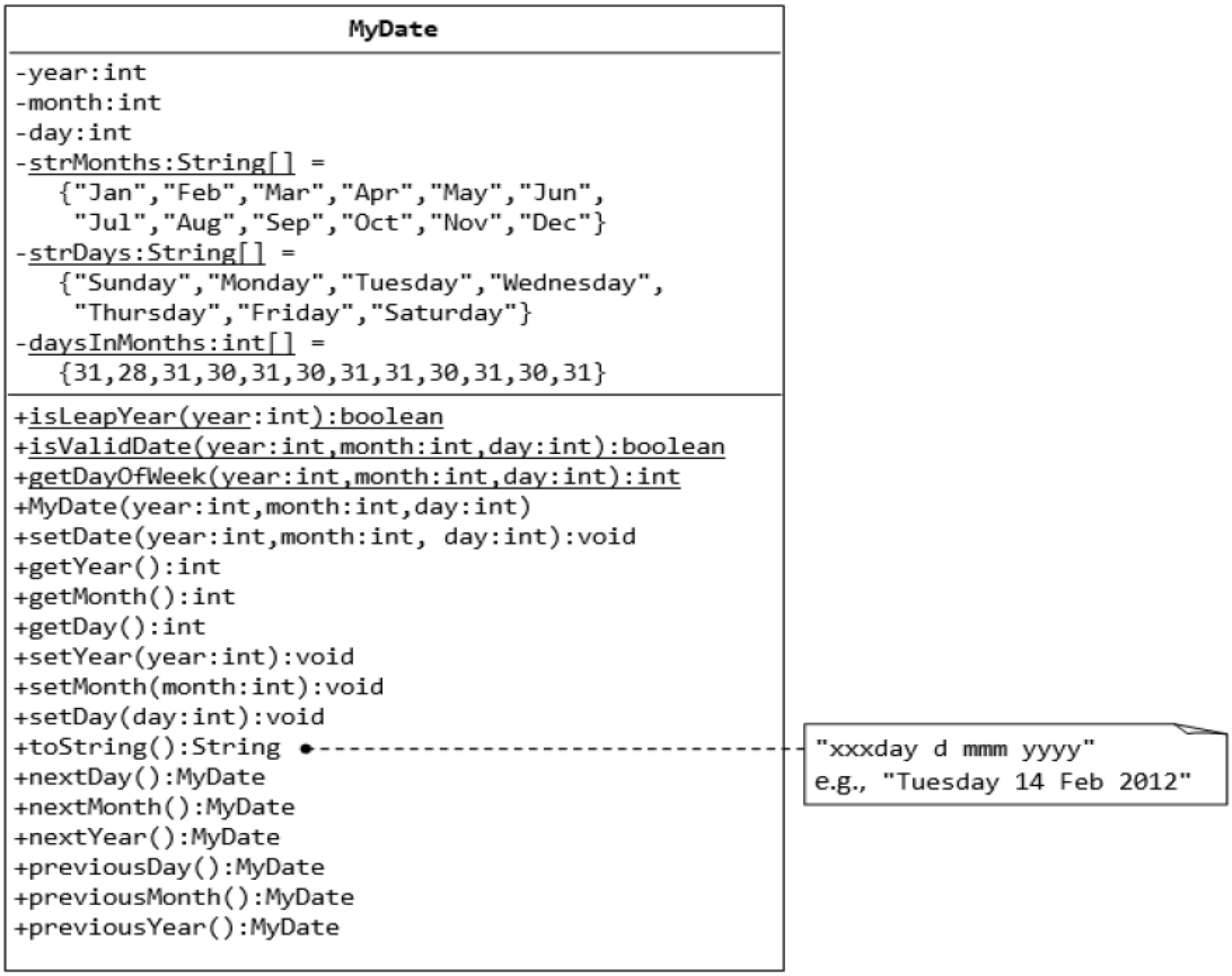
It contains the following `public` methods:

- `setTime(int hour, int minute, int second)`: It shall check if the given `hour, minute` and `second` are valid before setting the instance variables.
- (Advanced: Otherwise, it shall throw an `IllegalArgumentException` with the message "Invalid hour, minute, or second!".)
- Setters `setHour(int hour), setMinute(int minute), setSecond(int second)`: It shall check if the parameters are valid, similar to the above.

- Getters `getHour()`, `getMinute()`, `getSecond()`.
- `toString()`: returns "`HH:MM:SS`".
- `nextSecond()`: Update this instance to the next second and return this instance. Take note that the `nextSecond()` of `23:59:59` is `00:00:00`.
- `nextMinute()`, `nextHour()`, `previousSecond()`, `previousMinute()`, `previousHour()`: similar to the above.

Write the code for the `MyTime` class. Also write a test driver (called `TestMyTime`) to test all the public methods defined in the `MyTime` class.

3.5 The MyDate Class



A class called `MyDate`, which models a date instance, is defined as shown in the class diagram.

The MyDate class contains the following private instance variables:

- `year (int): Between 1 to 9999.`
- `month (int): Between 1 (Jan) to 12 (Dec).`
- `day (int): Between 1 to 28|29|30|31, where the last day depends on the month and whether it is a leap year for Feb (28|29).`

It also contains the following `private static` variables (drawn with underlined in the class diagram):

- `strMonths` (`String[]`), `strDays` (`String[]`), and `dayInMonths` (`int[]`): `static` variables, initialized as shown, which are used in the methods.

The `MyDate` class has the following public static methods (drawn with underlined in the class diagram):

- `isLeapYear(int year)`: returns `true` if the given year is a leap year. A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.
- `isValidDate(int year, int month, int day)`: returns `true` if the given `year`, `month`, and `day` constitute a valid `date`. Assume that `year` is between `1` and `9999`, `month` is between `1(Jan)` to `12(Dec)` and `day` shall be between `1` and `28|29|30|31` depending on the `month` and whether it is a leap year on Feb.
- `getDayOfWeek(int year, int month, int day)`: returns the day of the week, where `0` for `Sun`, `1` for `Mon`, `...,6` for Sat, for the given date. Assume that the date is valid. Read the earlier exercise on how to determine the day of the week.

The `MyDate` class has one constructor, which takes 3 parameters: `year`, `month` and `day`. It shall invoke `setDate()` method (to be described later) to set the instance variables.

The MyDate class has the following public methods:

- `setDate(int year, int month, int day)`: It shall invoke the `static` method `isValidDate()` to verify that the given `year`, `month` and `day` constitute a valid `date`. (Advanced: Otherwise, it shall throw an `IllegalArgumentException` with the message "`Invalid year, month, or day!`".)
- `setYear(int year)`: It shall verify that the given `year` is between `1` and `9999`. (Advanced: Otherwise, it shall throw an `IllegalArgumentException` with the message "`Invalid year!`".)
- `setMonth(int month)`: It shall verify that the given `month` is between `1` and `12`. (Advanced: Otherwise, it shall throw an `IllegalArgumentException` with the message "`Invalid month!`".)
- `setDay(int day)`: It shall verify that the given `day` is between `1` and `dayMax`, where `dayMax` depends on the `month` and whether it is a `leap year` for `Feb`. (Advanced: Otherwise, it shall throw an `IllegalArgumentException` with the message "`Invalid month!`".)

- `getYear()`, `getMonth()`, `getDay()`: return the value for the `year`, `month` and `day`, respectively.
- `toString()`: returns a date string in the format "`xxxday d mmm yyyy`", e.g., "Saturday 08 Feb 2020".
- `nextDay()`: update `this` instance to the next day and return `this` instance. Take note that `nextDay()` for `31 Dec 2000` shall be `1 Jan 2001`.
- `nextMonth()`: update `this` instance to the next month and return `this` instance. Take note that `nextMonth()` for `31 Oct 2012` shall be `30 Nov 2012`.
- `nextYear()`: update `this` instance to the next year and return `this` instance. Take note that `nextYear()` for `29 Feb 2012` shall be `28 Feb 2013`.

  (Advanced: throw an `IllegalStateException` with the message "`Year out of range!`" if `year > 9999`.)
- `previousDay()`, `previousMonth()`, `previousYear()`: similar to the above.

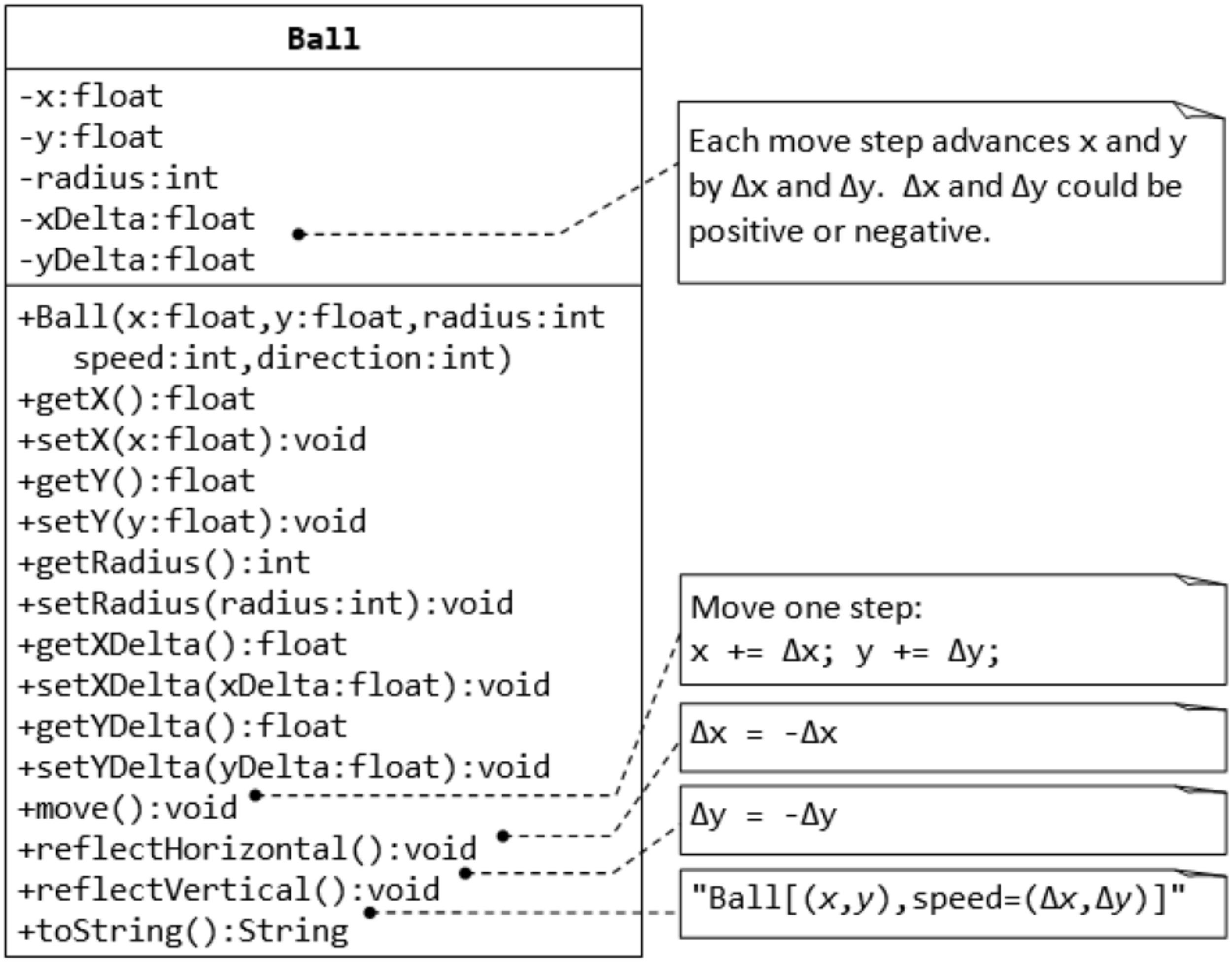
Write the code for the `MyDate` class.

Use the following test statements to test the `MyDate` class:

```
MyDate d1 = new MyDate(2012, 2, 28);
System.out.println(d1);              // Tuesday 28 Feb 2012
System.out.println(d1.nextDay());    // Wednesday 29 Feb 2012
System.out.println(d1.nextDay());    // Thursday 1 Mar 2012
System.out.println(d1.nextMonth()); // Sunday 1 Apr 2012
System.out.println(d1.nextYear());  // Monday 1 Apr 2013

MyDate d2 = new MyDate(2012, 1, 2);
System.out.println(d2);                  // Monday 2 Jan 2012
System.out.println(d2.previousDay());    // Sunday 1 Jan 2012
System.out.println(d2.previousDay());    // Saturday 31 Dec 2011
System.out.println(d2.previousMonth()); // Wednesday 30 Nov 2011
System.out.println(d2.previousYear());  // Tuesday 30 Nov 2010

MyDate d3 = new MyDate(2012, 2, 29);
System.out.println(d3.previousYear());  // Monday 28 Feb 2011

// MyDate d4 = new MyDate(2099, 11, 31); // Invalid year, month, or day!
// MyDate d5 = new MyDate(2011, 2, 29);  // Invalid year, month, or day!
```

Write a test program that tests the `nextDay()` in a loop, by printing the dates from `28 Dec 2011` to `2 Mar 2012`.

3.6 Bouncing Balls - Ball and Container Classes



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

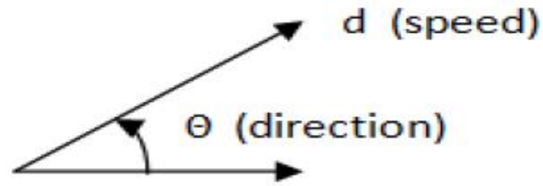
The `Ball` class contains the following `private` instance variables:

- `x`, `y` and `radius`, which represent the ball's `center (x, y)` co-ordinates and the `radius`, respectively.
- `xDelta(Δx)` and `yDelta(Δy)`, which represent the displacement (movement) per step, in the `x` and `y` direction respectively.

The `Ball` class contains the following `public` methods:

- A constructor which accepts `x`, `y`, `radius`, `speed`, and `direction` as arguments. For user friendliness, user specifies `speed` (in pixels per step) and `direction` (in degrees in the range of `(-180°, 180°])`. For the internal operations, the `speed` and `direction` are to be converted to `(Δx, Δy)` in the internal representation. Note that the y-axis of the Java graphics coordinate system is inverted, i.e., the origin `(0, 0)` is located at the top-left corner.

Java Graphics
Co-ordinates system



User's Polar
Co-ordinates

```
Δx = d × cos(θ)
Δy = -d × sin(θ)
```

- Getter and setter for all the instance variables.
- A method `move()` which move the ball by one step.

```
x += Δx
y += Δy
```

- `reflectHorizontal()` which reflects the ball horizontally (i.e., hitting a vertical wall)

```
Δx = -Δx
Δy no changes
```

- `reflectVertical()` (the ball hits a horizontal wall).

```
Δx no changes
Δy = -Δy
```

- `toString()` which prints the message "`Ball at (x, y) of velocity (Δx, Δy)`".

Write the `Ball` class. Also write a test program to test all the methods defined in the class.

Container
-x1:int -y1:int -x2:int -y2:int
+Container(x:int,y:int, width:int,height:int) +getX():int +getY():int +getWidth():int +getHeight():int +collides(ball:Ball):boolean +toString():String

"Container[(x1,y1),(x2,y2)]"

A class called `Container`, which represents the enclosing box for the ball, is designed as shown in the class diagram. It contains:

Instance variables (`x1, y1`) and (`x2, y2`) which denote the top-left and bottom-right corners of the rectangular box.

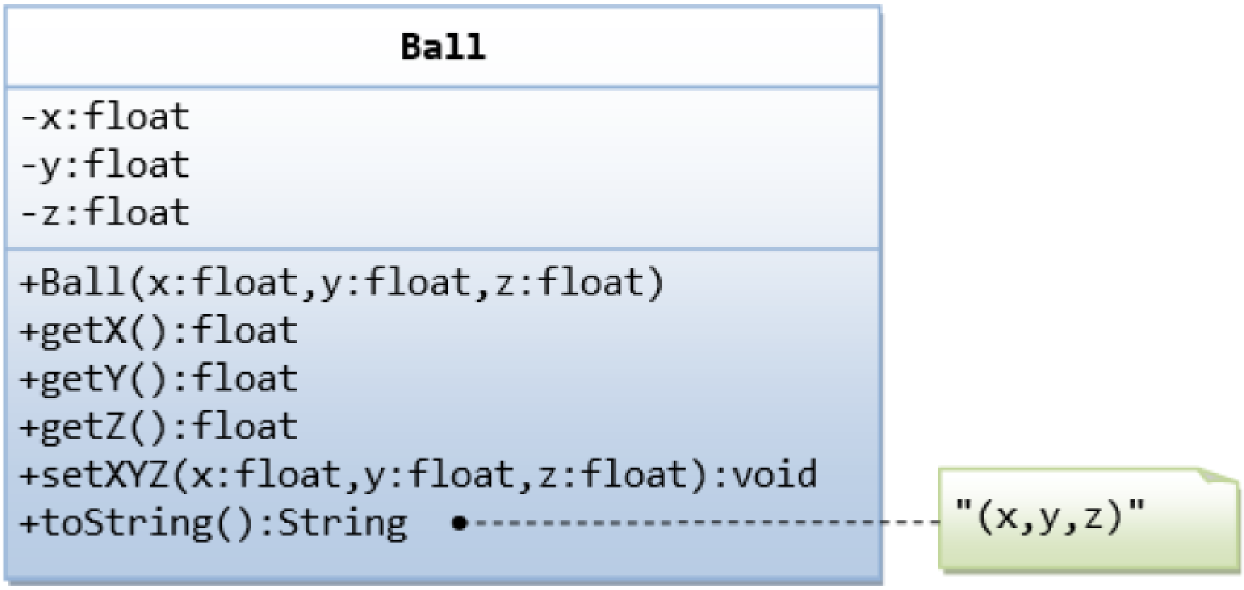
- A constructor which accepts (`x, y`) of the top-left corner, `width` and `height` as argument, and converts them into the internal representation (i.e., `x2=x1+width-1`). `Width` and `height` is used in the argument for safer operation (there is no need to check the validity of `x2>x1` etc.).
- A `toString()` method that returns "Container at (`x1,y1`) to (`x2, y2`)".
- A `boolean` method called `collidesWith(Ball)`, which check if the given `Ball` is outside the bounds of the container box. If so, it invokes the Ball's `reflectHorizontal()` and/or `reflectVertical()` to change the movement direction of the ball, and returns `true`.

```
public boolean collidesWith(Ball ball) {
     if (ball.getX() - ball.getRadius() <= this.x1 ||
     ball.getX() - ball.getRadius() >= this.x2) {
          ball.reflectHorizontal();
          return true;
     }
     ......
}
```

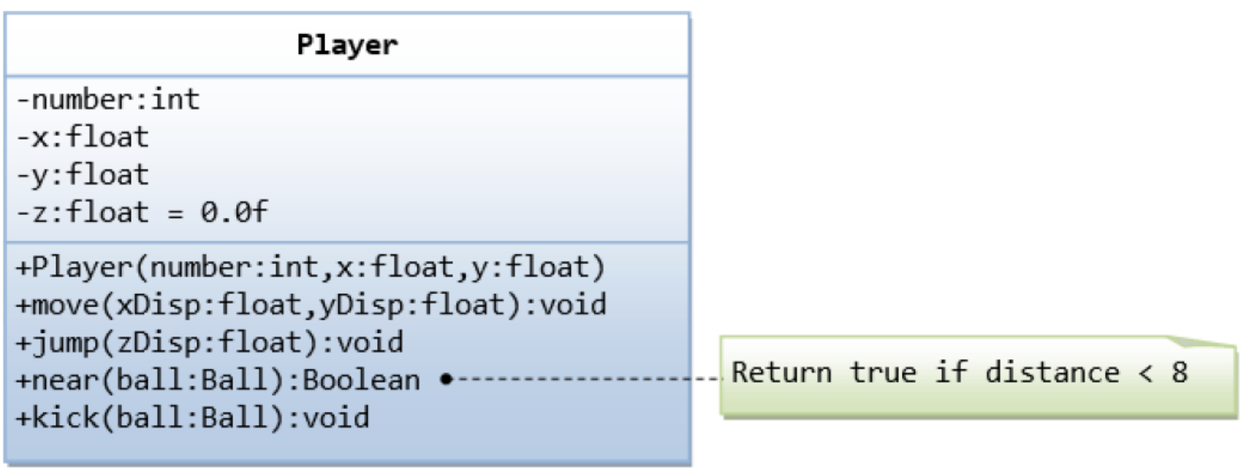
Use the following statements to test your program:

```
Ball ball = new Ball(50, 50, 5, 10, 30);
Container box = new Container(0, 0, 100, 100);
for (int step = 0; step < 100; ++step) {
      ball.move();
      box.collidesWith(ball);
      System.out.println(ball); // manual check the position of the ball
}
```

3.7 The Ball and Player Classes



The `Ball` class, which models the ball in a soccer game, is designed as shown in the class diagram. Write the codes for the `Ball` class and a test driver to test all the `public` methods.

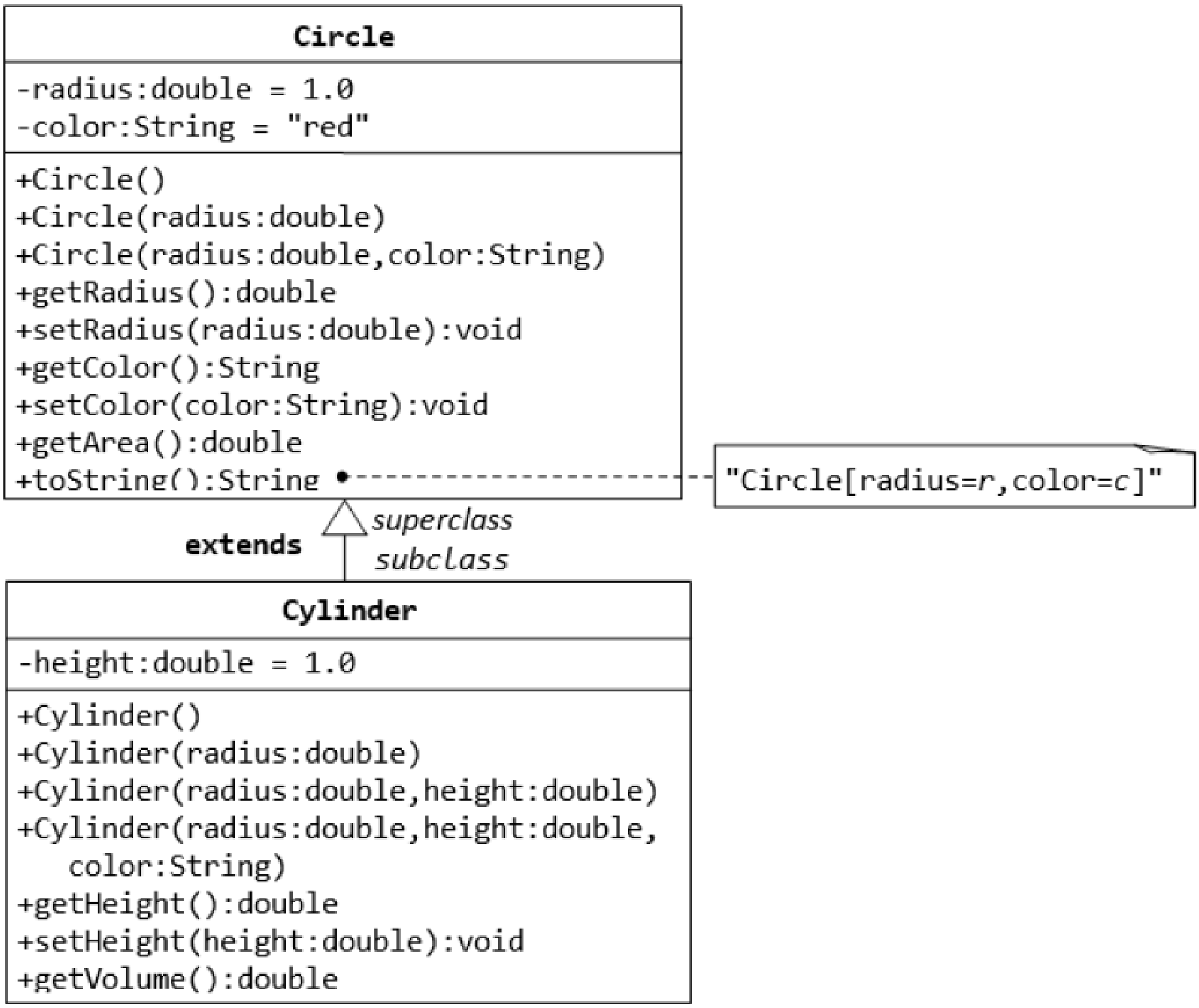


The `Player` class, which models the players in a soccer game, is designed as shown in the class diagram. The Player interacts with the Ball (written earlier). Write the codes for the `Player` class and a test driver to test all the `public` methods. Make your assumption for the `kick()`. Can you write a very simple soccer game with 2 teams of players and a ball, inside a soccer field?

4. Exercises on Inheritance

4.1 The Circle and Cylinder Classes

This exercise shall guide you through the important concepts in inheritance.



In this exercise, a subclass called `Cylinder` is derived from the superclass `Circle` as shown in the class diagram (where an arrow pointing up from the `subclass` to its `superclass`). Study how the subclass `Cylinder` invokes the superclass' constructors (via `super()` and `super(radius)`) and inherits the `variables` and `methods` from the superclass `Circle`. You can reuse the `Circle` class that you have created in the previous exercise. Make sure that you keep "`Circle.class`" in the same directory.

```java
public class Cylinder extends Circle { // Save as "Cylinder.java"
	private double height; // private variable
	// Constructor with default color, radius and height
	public Cylinder() {
		super(); // call superclass no-arg constructor Circle()
		height = 1.0;
	}
	// Constructor with default radius, color but given height
	public Cylinder(double height) {
		super(); // call superclass no-arg constructor Circle()
		this.height = height;
	}
	// Constructor with default color, but given radius, height
	public Cylinder(double radius, double height) {
		super(radius); // call superclass constructor Circle(r)
		this.height = height;
	}
	// A public method for retrieving the height
	public double getHeight() {
		return height;
	}
	// A public method for computing the volume of cylinder
	// use superclass method getArea() to get the base area
	public double getVolume() {
		return getArea()*height;
	}
}
```

Write a test program (says `TestCylinder`) to test the `Cylinder` class created, as follow:

```java
public class TestCylinder { // save as "TestCylinder.java"
	public static void main (String[] args) {
		// Declare and allocate a new instance of cylinder
		// with default color, radius, and height
		Cylinder c1 = new Cylinder();
		System.out.println("Cylinder:"
			+ " radius=" + c1.getRadius()
```

```
                  + " height=" + c1.getHeight()
                  + " base area=" + c1.getArea()
                  + " volume=" + c1.getVolume());
            // Declare and allocate a new instance of cylinder
            // specifying height, with default color and radius
            Cylinder c2 = new Cylinder(10.0);
            System.out.println("Cylinder:"
                  + " radius=" + c2.getRadius()
                  + " height=" + c2.getHeight()
                  + " base area=" + c2.getArea()
                  + " volume=" + c2.getVolume());
            // Declare and allocate a new instance of cylinder
            // specifying radius and height, with default color
            Cylinder c3 = new Cylinder(2.0, 10.0);
            System.out.println("Cylinder:"
                  + " radius=" + c3.getRadius()
                  + " height=" + c3.getHeight()
                  + " base area=" + c3.getArea()
                  + " volume=" + c3.getVolume());
      }
}
```

Method Overriding and "`Super`": The subclass `Cylinder` inherits `getArea()` method from its superclass `Circle`. Try overriding the `getArea()` method in the subclass `Cylinder` to compute the surface area (`=2π×radius×height + 2×base-area`) of the `cylinder` instead of `base area`. That is, if `getArea()` is called by a `Circle` instance, it returns the area.

If `getArea()` is called by a `Cylinder` instance, it returns the `surface area` of the cylinder. If you override the `getArea()` in the subclass `Cylinder`, the `getVolume`() no longer works. This is because the `getVolume()` uses the overridden `getArea()` method found in the same class. (Java runtime will search the superclass only if it cannot locate the method in this class). Fix the `getVolume()`.

Hints: After overridding the `getArea()` in subclass `Cylinder`, you can choose to invoke the `getArea()` of the superclass Circle by calling `super.getArea()`.

TRY:

Provide a `toString()` method to the `Cylinder` class, which overrides the `toString()` inherited from the superclass `Circle`, e.g.,

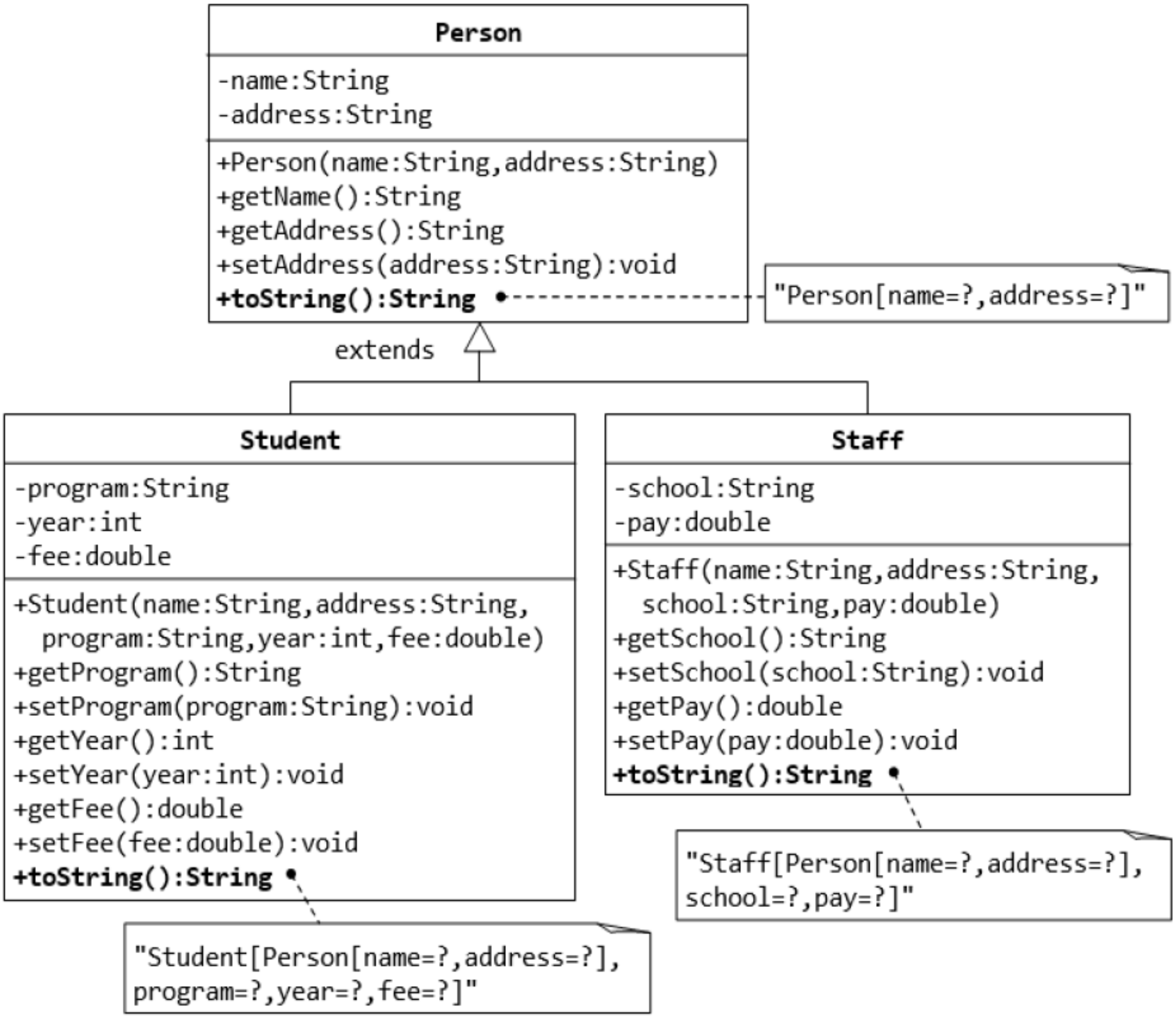
```
@Override
public String toString() { // in Cylinder class
      return "Cylinder: subclass of " + super.toString() // use Circle's
      toString()
      + " height=" + height;
}
```

Try out the `toString()` method in `TestCylinder`.

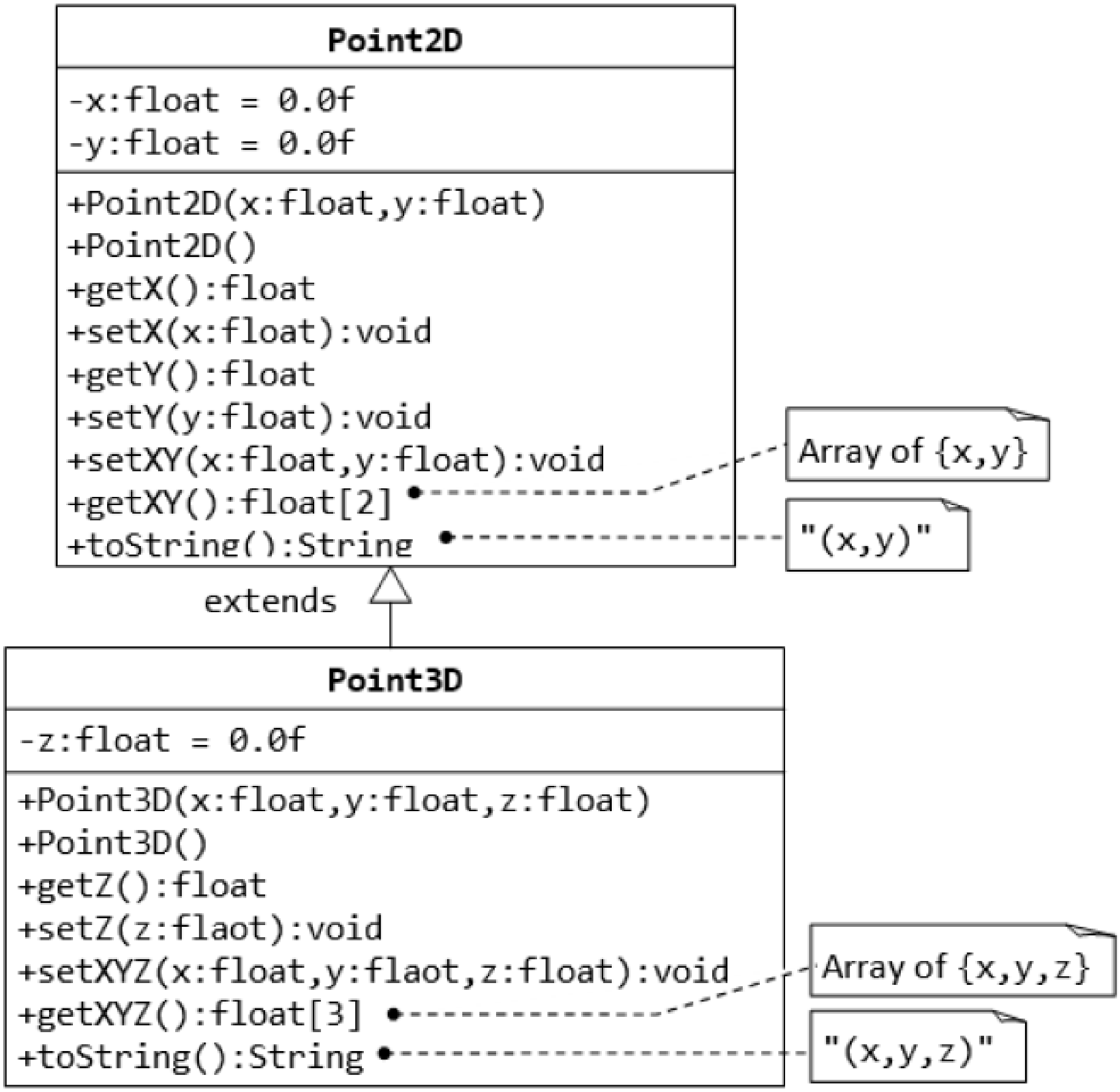
Note: `@Override` is known as annotation (introduced in `JDK 1.5`), which asks compiler to check whether there is such a method in the superclass to be overridden. This helps greatly if you misspell the name of the `toString()`. If `@Override` is not used and `toString()` is misspelled as `ToString()`, 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.

4.2 Superclass Person and its subclasses

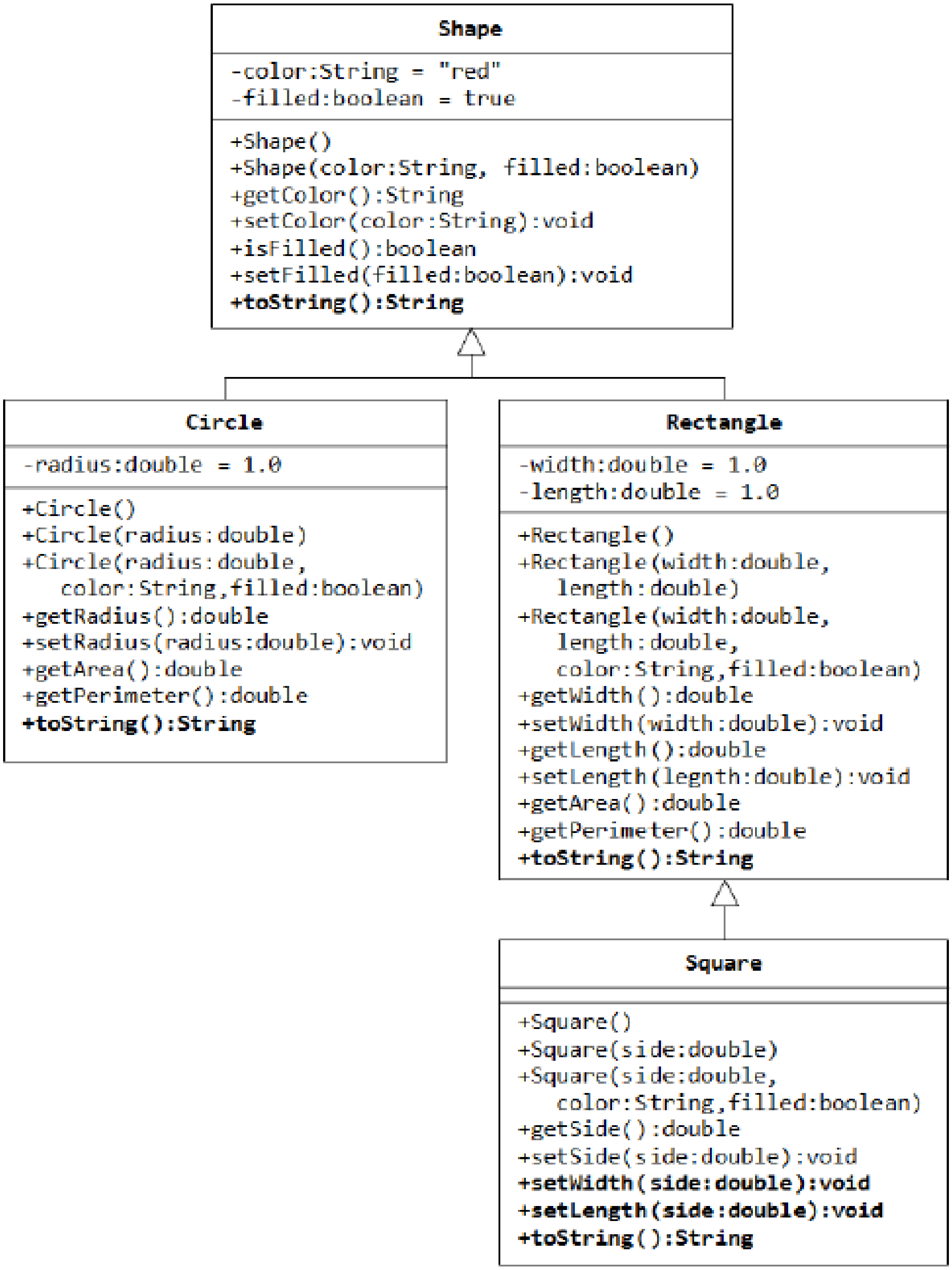


4.3 Point2D and Point3D



4.4 Point and MovablePoint

4.5 Superclass Shape and its subclasses Circle, Rectangle and Square



Write a superclass called `Shape` (as shown in the class diagram), which contains:

- Two instance variables `color(String)` and `filled(boolean)`.

- Two constructors: a `no-arg(no-argument)` constructor that initializes the `color` to `"green"` and `filled` to `true`, and a constructor that initializes the `color` and `filled` to the given values.
- Getter and setter for all the instance variables. By convention, the getter for a `boolean` variable `xxx` is called `isXXX()` (instead of `getXxx()` for all the other types).
- A `toString()` method that returns "`A Shape with color of xxx and filled/Not filled`".

Write a test program to test all the methods defined in `Shape`.

Write two subclasses of `Shape` called `Circle` and `Rectangle`, as shown in the class diagram.

The Circle class contains:

- An instance variable `radius (double)`.
- Three constructors as shown. The `no-arg` constructor initializes the `radius` to `1.0`.
- Getter and setter for the instance variable `radius`.
- Methods `getArea()` and `getPerimeter()`.
- Override the `toString()` method inherited, to return "`A Circle with radius=xxx, which is a subclass of yyy`", where `yyy` is the output of the `toString()` method from the superclass.

The Rectangle class contains:

- Two instance variables `width(double)` and `length(double)`.
- Three constructors as shown. The no-arg constructor initializes the `width` and `length` to `1.0`.
- Getter and setter for all the instance variables.
- Methods `getArea()` and `getPerimeter()`.
- Override the `toString()` method inherited, to return "`A Rectangle with width=xxx and length=zzz, which is a subclass of yyy`", where `yyy` is the output of the `toString()` method from the superclass.

Write a class called `Square`, as a subclass of `Rectangle`. Convince yourself that `Square` can be modeled as a subclass of `Rectangle`. `Square` has no instance variable, but inherits the instance variables `width` and `length` from its superclass `Rectangle`.

- Provide the appropriate constructors (as shown in the class diagram). Hint:

```
public Square(double side) {
      super(side, side); // Call superclass Rectangle(double, double)
}
```

- Override the `toString()` method to return "`A Square with side=xxx, which is a subclass of yyy`", where `yyy` is the output of the `toString()` method from the superclass.
- Do you need to override the `getArea()` and `getPerimeter()`? Try them out.
- Override the `setLength()` and `setWidth()` to change both the `width` and `length`, so as to maintain the square geometry.

5. Exercises on Composition vs Inheritance

They are two ways to reuse a class in your applications: `composition and inheritance`.

5.1 The Point and Line Classes

Let us begin with *composition* with the statement "a line composes of two points".

Complete the definition of the following two classes: `Point` and `Line`. The class `Line` composes 2 instances of class `Point`, representing the beginning and ending points of the line.

Also write test classes for `Point` and `Line` (says `TestPoint` and `TestLine`).

```
public class Point {
	// Private variables
	private int x; // x co-ordinate
	private int y; // y co-ordinate
	// Constructor
	public Point (int x, int y) {......}
		// Public methods
	public String toString() {
		return "Point: (" + x + "," + y + ")";
	}
	public int getX() {......}
	public int getY() {......}
	public void setX(int x) {......}
	public void setY(int y) {......}
	public void setXY(int x, int y) {......}
}
```

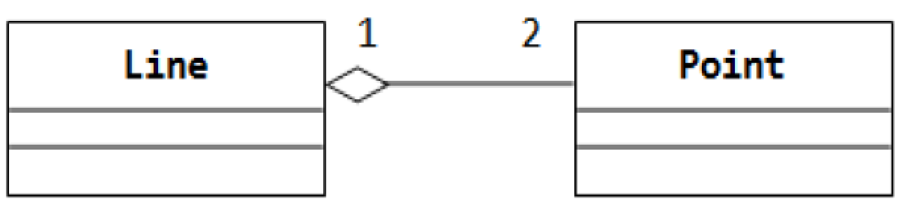
```
public class TestPoint {
	public static void main(String[] args) {
		Point p1 = new Point(10, 20); // Construct a Point
		System.out.println(p1);
		// Try setting p1 to (100, 10).
		......
	}
}
```

```
public class Line {
	// A line composes of two points (as instance variables)
	private Point begin; // beginning point
	private Point end; // ending point
	// Constructors
	public Line (Point begin, Point end) {
		// caller to construct the Points
		this.begin = begin;
		......
	}
```

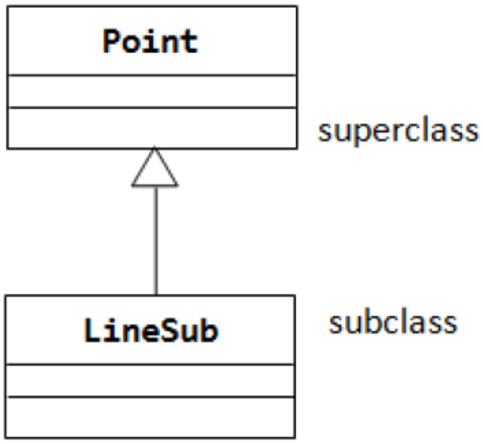
```
    public Line (int beginX, int beginY, int endX, int endY) {
        begin = new Point(beginX, beginY); // construct the Points here
        ......
    }
    // Public methods
    public String toString() { ...... }
    public Point getBegin() { ...... }
    public Point getEnd() { ...... }
    public void setBegin(......) { ...... }
    public void setEnd(......) { ...... }
    public int getBeginX() { ...... }
    public int getBeginY() { ...... }
    public int getEndX() { ...... }
    public int getEndY() { ...... }
    public void setBeginX(......) { ...... }
    public void setBeginY(......) { ...... }
    public void setBeginXY(......) { ...... }
    public void setEndX(......) { ...... }
    public void setEndY(......) { ...... }
    public void setEndXY(......) { ...... }
    public int getLength() { ...... } // Length of the line
    // Math.sqrt(xDiff*xDiff + yDiff*yDiff)
    public double getGradient() { ...... } // Gradient in radians
                                          // Math.atan2(yDiff, xDiff)
}
```

```
public class TestLine {
    public static void main(String[] args) {
        Line l1 = new Line(0, 0, 3, 4);
        System.out.println(l1);
        Point p1 = new Point(...);
        Point p2 = new Point(...);
        Line l2 = new Line(p1, p2);
        System.out.println(l2);
        ...
    }
}
```

The class diagram for *composition* is as follows (where a diamond-hollow-head arrow pointing to its constituents):



Instead of *composition*, we can design a `Line` class using `inheritance`. Instead of "a line composes of two points", we can say that "a line is a point extended by another point", as shown in the following class diagram:



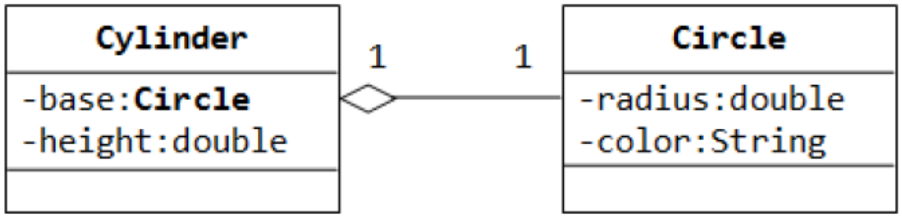
Let us re-design the `Line` class (called `LineSub`) as a subclass of class `Point`. `LineSub` inherits the starting point from its superclass `Point`, and adds an ending point. Complete the class definition. Write a testing class called `TestLineSub` to test `LineSub`.

```
public class LineSub extends Point {
	// A line needs two points: begin and end.
	// The begin point is inherited from its superclass Point.
	// Private variables
	Point end; // Ending point
	// Constructors
	public LineSub (int beginX, int beginY, int endX, int endY) {
		super(beginX, beginY); // construct the begin Point
		this.end = new Point(endX, endY); // construct the end Point
	}
	public LineSub (Point begin, Point end) { // caller to construct the
	Points
		super(begin.getX(), begin.getY());
		// need to reconstruct the begin Point
		this.end = end;
	}
	// Public methods
	// Inherits methods getX() and getY() from superclass Point
```

```
       public String toString() { ... }
       public Point getBegin() { ... }
       public Point getEnd() { ... }
       public void setBegin(...) { ... }
       public void setEnd(...) { ... }
       public int getBeginX() { ... }
       public int getBeginY() { ... }
       public int getEndX() { ... }
       public int getEndY() { ... }
       public void setBeginX(...) { ... }
       public void setBeginY(...) { ... }
       public void setBeginXY(...) { ... }
       public void setEndX(...) { ... }
       public void setEndY(...) { ... }
       public void setEndXY(...) { ... }
       public int getLength() { ... } // Length of the line
       public double getGradient() { ... } // Gradient in radians
}
```

Summary: There are two approaches that you can design a line, `composition` or `inheritance`. "A line composes two points" or "A line is a point extended with another point"". Compare the `Line` and `LineSub` designs: `Line` uses composition and `LineSub` uses inheritance. Which design is better?

5.2 The Circle and Cylinder Classes Using Composition



Try rewriting the `Circle-Cylinder` of the previous exercise using *`composition`* (as shown in the class diagram) instead of *`inheritance`*. That is, "a `cylinder` is composed of a `base circle` and a `height`".

```
public class Cylinder {
      private Circle base; // Base circle, an instance of Circle class
      private double height;
      // Constructor with default color, radius and height
      public Cylinder() {
            base = new Circle(); // Call the constructor to construct the
            Circle
            height = 1.0;
      }
      ......
}
```

Which design (`inheritance` or `composition`) is better?

6. Exercises on Polymorphism, Abstract Classes and Interfaces

6.1 Abstract Superclass Shape and Its Concrete Subclasses

Rewrite the superclass `Shape` and its subclasses `Circle`, `Rectangle` and `Square`, as shown in the class diagram.

In this exercise, `Shape` shall be defined as an `abstract` class, which contains:

- Two `protected` instance variables `color(String)` and `filled(boolean)`. The `protected` variables can be accessed by its subclasses and classes in the same package. They are denoted with a '#' sign in the class diagram.
- Getter and setter for all the instance variables, and `toString()`.
- Two abstract methods `getArea()` and `getPerimeter()` (shown in italics in the class diagram).

The subclasses `Circle` and `Rectangle` shall override the abstract methods `getArea()` and `getPerimeter()` and provide the proper implementation. They also override the `toString()`.

Write a test class to test these statements involving *polymorphism* and explain the outputs. Some statements may trigger compilation errors. Explain the errors, if any.

```
Shape s1 = new Circle(5.5, "RED", false); // Upcast Circle to Shape
System.out.println(s1); // which version?
System.out.println(s1.getArea()); // which version?
System.out.println(s1.getPerimeter()); // which version?
System.out.println(s1.getColor());
System.out.println(s1.isFilled());
System.out.println(s1.getRadius());

Circle c1 = (Circle)s1; // Downcast back to Circle
System.out.println(c1);
System.out.println(c1.getArea());
System.out.println(c1.getPerimeter());
System.out.println(c1.getColor());
System.out.println(c1.isFilled());
System.out.println(c1.getRadius());

Shape s2 = new Shape();

Shape s3 = new Rectangle(1.0, 2.0, "RED", false); // Upcast
System.out.println(s3);
System.out.println(s3.getArea());
System.out.println(s3.getPerimeter());
System.out.println(s3.getColor());
System.out.println(s3.getLength());

Rectangle r1 = (Rectangle)s3; // downcast
System.out.println(r1);
System.out.println(r1.getArea());
```

```
System.out.println(r1.getColor());
System.out.println(r1.getLength());


Shape s4 = new Square(6.6); // Upcast
System.out.println(s4);
System.out.println(s4.getArea());
System.out.println(s4.getColor());
System.out.println(s4.getSide());


// Take note that we downcast Shape s4 to Rectangle,
// which is a superclass of Square, instead of Square
Rectangle r2 = (Rectangle)s4;
System.out.println(r2);
System.out.println(r2.getArea());
System.out.println(r2.getColor());
System.out.println(r2.getSide());
System.out.println(r2.getLength());

// Downcast Rectangle r2 to Square
Square sq1 = (Square)r2;
System.out.println(sq1);
System.out.println(sq1.getArea());
System.out.println(sq1.getColor());
System.out.println(sq1.getSide());
System.out.println(sq1.getLength());
```

What is the usage of the `abstract` method and `abstract` class?

6.2 Polymorphism

Examine the following codes and draw the class diagram.

```
abstract public class Animal {
      abstract public void greeting();
}
```

```
public class Cat extends Animal {
      @Override
      public void greeting() {
            System.out.println("Meow!");
      }
}
```

```
public class Dog extends Animal {
      @Override
      public void greeting() {
            System.out.println("Woof!");
      }
      public void greeting(Dog another) {
            System.out.println("Woooooooooof!");
      }
}
```

```
public class BigDog extends Dog {
      @Override
      public void greeting() {
            System.out.println("Woow!");
      }
      @Override
      public void greeting(Dog another) {
            System.out.println("Woooooowwwww!");
      }
}
```

Explain the outputs (or error) for the following test program.

```
public class TestAnimal {
      public static void main(String[] args) {

            // Using the subclasses
            Cat cat1 = new Cat();
            cat1.greeting();
            Dog dog1 = new Dog();
            dog1.greeting();
            BigDog bigDog1 = new BigDog();
            bigDog1.greeting();
```

```
            // Using Polymorphism
            Animal animal1 = new Cat();
            animal1.greeting();
            Animal animal2 = new Dog();
            animal2.greeting();
            Animal animal3 = new BigDog();
            animal3.greeting();
            Animal animal4 = new Animal();

            // Downcast
            Dog dog2 = (Dog)animal2;
            BigDog bigDog2 = (BigDog)animal3;
            Dog dog3 = (Dog)animal3;
            Cat cat2 = (Cat)animal2;
            dog2.greeting(dog3);
            dog3.greeting(dog2);
            dog2.greeting(bigDog2);
            bigDog2.greeting(dog2);
            bigDog2.greeting(bigDog1);
      }
}
```

<< END OF EXERCISE >>