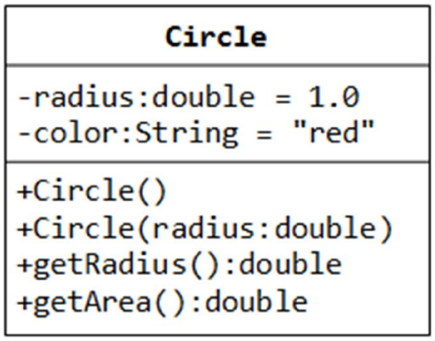
1.1 Exercise: The Circle Class



Circle

-radius:double = 1.0
-color:String = "red"

+Circle()
+Circle(radius:double)
+getRadius():double
+getArea():double

A class called **`circle`** is designed as shown in the following class diagram. It contains:

- Two `private` instance variables: `radius` (of type `double`) and `color` (of type `String`), with default value of 1.0 and "red", respectively.
- Two *overloaded* constructors;
- Two `public` methods: `getRadius()` and `getArea()`.

The source codes for `Circle` is as follows:

```
public class Circle {            // save as "Circle.java"
   // private instance variable, not accessible from outside this class
   private double radius;
   private String color;

   // 1st constructor, which sets both radius and color to default
   public Circle() {
      radius = 1.0;
      color = "red";
   }

   // 2nd constructor with given radius, but color default
   public Circle(double r) {
      radius = r;
      color = "red";
   }

   // A public method for retrieving the radius
   public double getRadius() {
     return radius;
   }

   // A public method for computing the area of circle
   public double getArea() {
      return radius*radius*Math.PI;
   }
}
```

Compile "`Circle.java`". Can you run the `Circle` class? Why? This `Circle` class does not have a `main()` method. Hence, it cannot be run directly. This `Circle` class is a “building block” and is meant to be used in another program.

Let us write a *test program* called `TestCircle` which uses the `Circle` class, as follows:

```
public class TestCircle {        // save as "TestCircle.java"
   public static void main(String[] args) {
      // Declare and allocate an instance of class Circle called c1
      //  with default radius and color
      Circle c1 = new Circle();
      // Use the dot operator to invoke methods of instance c1.
      System.out.println("The circle has radius of "
         + c1.getRadius() + " and area of " + c1.getArea());

      // Declare and allocate an instance of class circle called c2
      //  with the given radius and default color
      Circle c2 = new Circle(2.0);
      // Use the dot operator to invoke methods of instance c2.
      System.out.println("The circle has radius of "
         + c2.getRadius() + " and area of " + c2.getArea());
   }
}
```

Now, run the `TestCircle` and study the results.

TRY:

1. Constructor: Modify the class `Circle` to include a third constructor for constructing a `Circle` instance with the given `radius` and `color`.
2.
```
// Construtor to construct a new instance of Circle with the given
radius and color
public Circle (double r, String c) {......}
```

   Modify the test program `TestCircle` to construct an instance of `Circle` using this constructor.

3. Getter: Add a getter for variable `color` for retrieving the `color` of a `Circle` instance.
4.
```
// Getter for instance variable color
public String getColor() {......}
```

   Modify the test program to test this method.

5. public vs. private: In `TestCircle`, can you access the instance variable `radius` directly (e.g., `System.out.println(c1.radius)`); or assign a new value to `radius` (e.g., `c1.radius=5.0`)? Try it out and explain the error messages.
6. Setter: Is there a need to change the values of `radius` and `color` of a `Circle` instance after it is constructed? If so, add two `public` methods called *setters* for changing the `radius` and `color` of a `Circle` instance as follows:

```
7. // Setter for instance variable radius
8. public void setRadius(double r) {
9.    radius = r;
10.  }
11.
12.  // Setter for instance variable color
   public void setColor(String c) { ...... }
```

   Modify the `TestCircle` to test these methods, e.g.,

```
Circle c3 = new Circle();   // construct an instance of Circle
c3.setRadius(5.0);          // change radius
c3.setColor(...);           // change color
```

13. Keyword "this": Instead of using variable names such as `r` (for `radius`) and `c` (for `color`) in the methods' arguments, it is better to use variable names `radius` (for `radius`) and `color` (for `color`) and use the special keyword "`this`" to resolve the conflict between instance variables and methods' arguments. For example,

```
14.  // Instance variable
15.  private double radius;
16.
17.  // Setter of radius
18.  public void setRadius(double radius) {
19.     this.radius = radius;   // "this.radius" refers to the instance
   variable
20.                             // "radius" refers to the method's
   argument
   }
```

Modify ALL the constructors and setters in the `Circle` class to use the keyword "`this`".

21. Method toString(): Every well-designed Java class should contain a `public` method called `toString()` that returns a short description of the instance (in a return type of `String`). The `toString()` method can be called explicitly (via ***instanceName***`.toString()`) just like any other method; or implicitly through `println()`. If an instance is passed to the `println(`***anInstance***`)` method, the `toString()` method of that instance will be invoked implicitly. For example, include the following `toString()` methods to the `Circle` class:

```
22.  public String toString() {
23.     return "Circle: radius=" + radius + " color=" + color;
   }
```

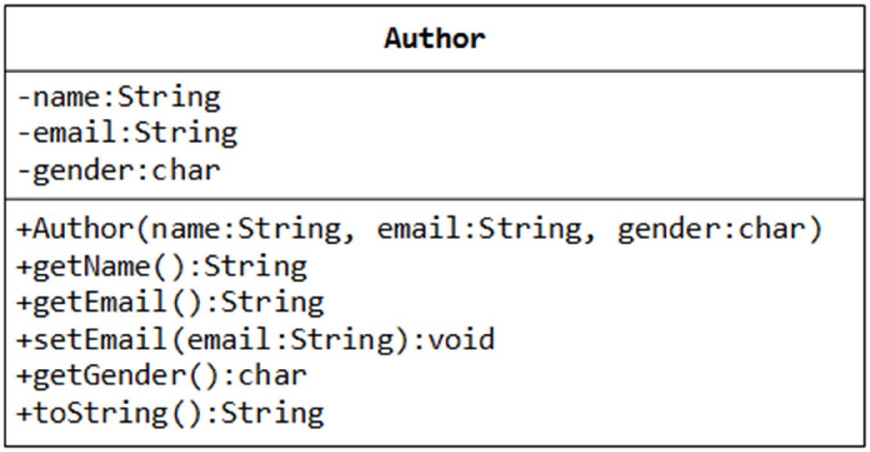
Try calling `toString()` method explicitly, just like any other method:

```
Circle c1 = new Circle(5.0);
System.out.println(c1.toString());   // explicit call
```

`toString()` is called implicitly when an instance is passed to `println()` method, for example,

```
Circle c2 = new Circle(1.2);
System.out.println(c2.toString());  // explicit call
System.out.println(c2);             // println() calls toString()
implicitly, same as above
System.out.println("Operator '+' invokes toString() too: " + c2);  //
'+' invokes toString() too
```

1.2 Exercise: The Author and Book Classes



Author

-name:String
-email:String
-gender:char

+Author(name:String, email:String, gender:char)
+getName():String
+getEmail():String
+setEmail(email:String):void
+getGender():char
+toString():String

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

- Three `private` instance variables: `name` (`String`), `email` (`String`), and `gender` (`char` of either `'m'` or `'f'`);
- One constructor to initialize the `name`, `email` and `gender` with the given values;

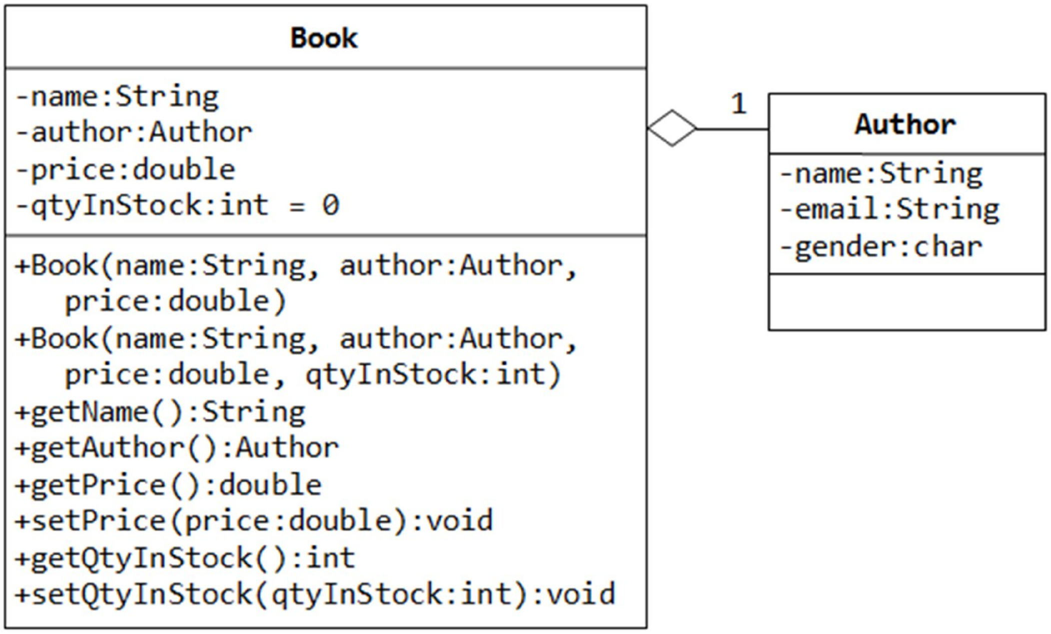
  ```
  public Author (String name, String email, char gender) {......}
  ```

  (There is no default constructor for Author, as there are no defaults for name, email and gender.)

- `public` getters/setters: `getName()`, `getEmail()`, `setEmail()`, and `getGender()`; (There are no setters for `name` and `gender`, as these attributes cannot be changed.)
- A `toString()` method that returns "*author-name (gender) at email*", e.g., "Tan Ah Teck (m) at ahTeck@somewhere.com".

Write the `Author` class. Also write a *test program* called `TestAuthor` to test the constructor and `public` methods. Try changing the `email` of an author, e.g.,

```
Author anAuthor = new Author("Tan Ah Teck", "ahteck@somewhere.com", 'm');
System.out.println(anAuthor);   // call toString()
anAuthor.setEmail("paul@nowhere.com")
System.out.println(anAuthor);
```



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

- Four `private` instance variables: `name` (`String`), `author` (of the class `Author` you have just created, assume that each book has one and only one author), `price` (`double`), and `qtyInStock` (`int`);
- Two constructors:
- `public Book (String name, Author author, double price) {...}`

-
  ```
  public Book (String name, Author author, double price,
       int qtyInStock) {...}
  ```
- public methods `getName()`, `getAuthor()`, `getPrice()`, `setPrice()`, `getQtyInStock()`, `setQtyInStock()`.
- `toString()` that returns "*'book-name' by author-name (gender) at email*". (Take note that the `Author`'s `toString()` method returns "*author-name (gender) at email*".)

Write the class `Book` (which uses the `Author` class written earlier). Also write a test program called `TestBook` to test the constructor and `public` methods in the class `Book`. Take Note that you have to construct an instance of `Author` before you can construct an instance of `Book`. E.g.,

```
Author anAuthor = new Author(......);
Book aBook = new Book("Java for dummy", anAuthor, 19.95, 1000);
// Use an anonymous instance of Author
Book anotherBook = new Book("more Java for dummy", new Author(......),
29.95, 888);
```

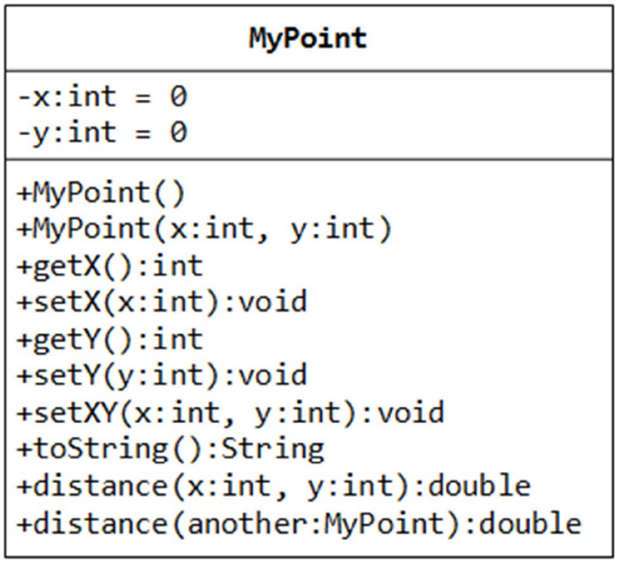
Take note that both `Book` and `Author` classes have a variable called `name`. However, it can be differentiated via the referencing instance. For a `Book` instance says `aBook`, `aBook.name` refers to the `name` of the book; whereas for an `Author`'s instance say `auAuthor`, `anAuthor.name` refers to the `name` of the author. There is no need (and not recommended) to call the variables `bookName` and `authorName`.

TRY:

1. Printing the `name` and `email` of the author from a `Book` instance. (Hint: `aBook.getAuthor().getName()`, `aBook.getAuthor().getEmail()`).
2. Introduce new methods called `getAuthorName()`, `getAuthorEmail()`, `getAuthorGender()` in the `Book` class to return the `name`, `email` and `gender` of the author of the book. For example,

   ```
   public String getAuthorName() { ...... }
   ```

1.3 Exercise: The MyPoint Class



A class called `MyPoint`, which models a 2D point with `x` and `y` coordinates, is designed as shown in the class diagram. It contains:

- Two instance variables `x` (`int`) and `y` (`int`).
- A "no-argument" (or "no-arg") constructor that construct a point at `(0, 0)`.
- A constructor that constructs a point with the given `x` and `y` coordinates.
- Getter and setter for the instance variables `x` and `y`.
- A method `setXY()` to set both `x` and `y`.
- A `toString()` method that returns a string description of the instance in the format "`(`*`x`*`, `*`y`*`)`".
- A method called `distance(int x, int y)` that returns the distance from *this* point to another point at the given `(x, y)` coordinates.
- An overloaded `distance(MyPoint another)` that returns the distance from *this* point to the given `MyPoint` instance `another`.

You are required to:

1. Write the code for the class `MyPoint`. Also write a test program (called `TestMyPoint`) to test all the methods defined in the class.
   Hints:

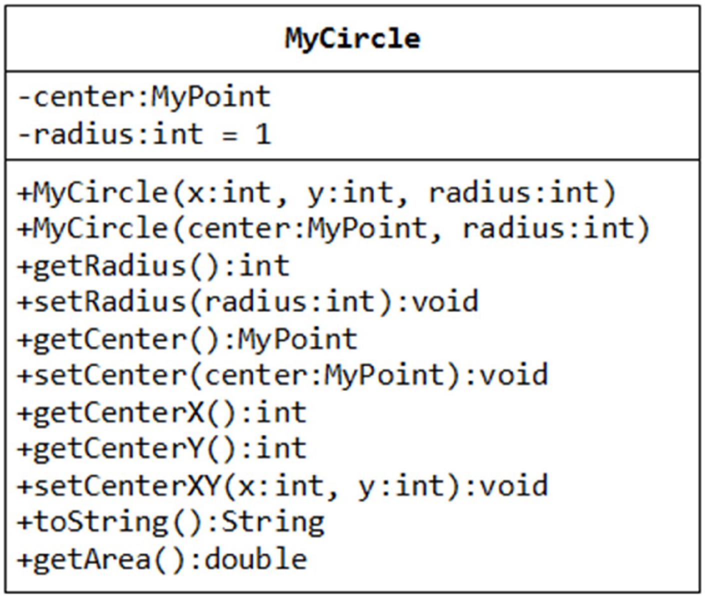
```
2. // Overloading method distance()
3. public double distance(int x, int y) {   // this version takes two ints as arguments
4.    int xDiff = this.x – x;
5.    int yDiff = ......
6.    return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
7. }
8.
9. public double distance(MyPoint another) { // this version takes a MyPoint instance as argument
10.    int xDiff = this.x – another.x;
11.    .......
12.  }
13.
14.  // Test program
15.  MyPoint p1 = new MyPoint(3, 0);
16.  MyPoint p2 = new MyPoint(0, 4);
17.  ......
18.  // Testing the overloaded method distance()
19.  System.out.println(p1.distance(p2));    // which version?
20.  System.out.println(p1.distance(5, 6));  // which version?
   .....
```

21. Write a program that allocates `10` points in an array of `MyPoint`, and initializes to `(1, 1)`, `(2, 2)`, ... `(10, 10)`.
    Hints: You need to allocate the array, as well as each of the ten `MyPoint` instances.

```
22.  MyPoint[] points = new MyPoint[10]; // Declare and allocate an array of MyPoint
23.  for (......) {
24.     points[i] = new MyPoint(...);    // Allocate each of MyPoint instances
   }
```

Notes: Point is such a common entity that JDK certainly provided for in all flavors.

1.4 Exercise: The MyCircle Class



```
MyCircle
-center:MyPoint
-radius:int = 1
+MyCircle(x:int, y:int, radius:int)
+MyCircle(center:MyPoint, radius:int)
+getRadius():int
+setRadius(radius:int):void
+getCenter():MyPoint
+setCenter(center:MyPoint):void
+getCenterX():int
+getCenterY():int
+setCenterXY(x:int, y:int):void
+toString():String
+getArea():double
```

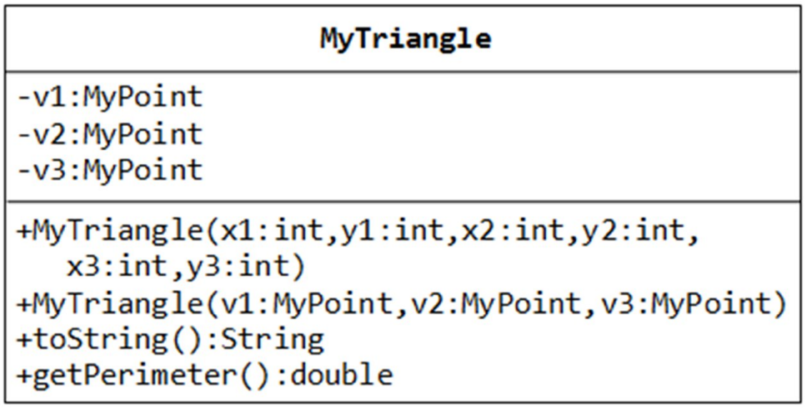
A class called `MyCircle`, which models a circle with a `center` (x, y) and a `radius`, is designed as shown in the class diagram. The `MyCircle` class uses an instance of `MyPoint` class (created in the previous exercise) as its `center`.

The class contains:

- Two `private` instance variables: `center` (an instance of `MyPoint`) and `radius` (`int`).
- A constructor that constructs a circle with the given center's (`x`, `y`) and `radius`.
- An overloaded constructor that constructs a `MyCircle` given a `MyPoint` instance as `center`, and `radius`.
- Various getters and setters.
- A `toString()` method that returns a string description of this instance in the format "Circle @ (*x*, *y*) radius=*r*".
- A `getArea()` method that returns the area of the circle in `double`.

Write the `MyCircle` class. Also write a test program (called `TestMyCircle`) to test all the methods defined in the class.

1.5 Exercise: The MyTriangle Class



```
MyTriangle
-v1:MyPoint
-v2:MyPoint
-v3:MyPoint
+MyTriangle(x1:int,y1:int,x2:int,y2:int,
   x3:int,y3:int)
+MyTriangle(v1:MyPoint,v2:MyPoint,v3:MyPoint)
+toString():String
+getPerimeter():double
```

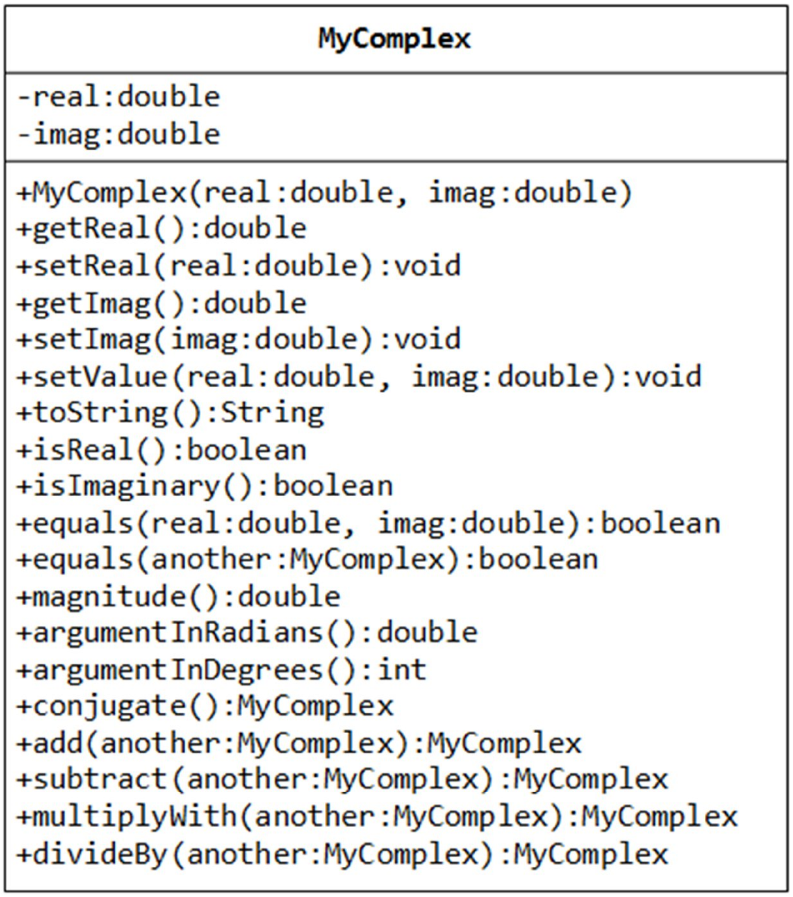
A class called `MyTriangle`, which models a triangle with 3 vertices, is designed as follows. The `MyTriangle` class uses three `MyPoint` instances (created in the earlier exercise) as the three vertices.

The class contains:

- Three `private` instance variables `v1`, `v2`, `v3` (instances of `MyPoint`), for the three vertices.
- A constructor that constructs a `MyTriangle` with three points `v1=(x1, y1)`, `v2=(x2, y2)`, `v3=(x3, y3)`.
- An overloaded constructor that constructs a `MyTriangle` given three instances of `MyPoint`.
- A `toString()` method that returns a string description of the instance in the format "Triangle @ (x1, y1), (x2, y2), (x3, y3)".
- A `getPerimeter()` method that returns the length of the perimeter in double. You should use the `distance()` method of `MyPoint` to compute the perimeter.
- A method `printType()`, which prints "equilateral" if all the three sides are equal, "isosceles" if any two of the three sides are equal, or "scalene" if the three sides are different.

Write the `MyTriangle` class. Also write a test program (called `TestMyTriangle`) to test all the methods defined in the class.

1.6 Exercise: The MyComplex class



```
MyComplex
-real:double
-imag:double
+MyComplex(real:double, imag:double)
+getReal():double
+setReal(real:double):void
+getImag():double
+setImag(imag:double):void
+setValue(real:double, imag:double):void
+toString():String
+isReal():boolean
+isImaginary():boolean
+equals(real:double, imag:double):boolean
+equals(another:MyComplex):boolean
+magnitude():double
+argumentInRadians():double
+argumentInDegrees():int
+conjugate():MyComplex
+add(another:MyComplex):MyComplex
+subtract(another:MyComplex):MyComplex
+multiplyWith(another:MyComplex):MyComplex
+divideBy(another:MyComplex):MyComplex
```

A class called `MyComplex`, which models complex numbers `x+yi`, is designed as shown in the 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.
- 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. Hint:

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

- A method `equals(double real, double imag)` that returns `true` if *this* complex number is equal to the given complex number of (real, imag).
- An overloaded `equals(MyComplex another)` that returns `true` if *this* complex number is equal to the given `MyComplex` instance `another`.

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

  ```
  magnitude(x+yi) = Math.sqrt(x2 + y2)
  ```

- Methods `argumentInRadians()` and `argumentInDegrees()` that returns the argument of this complex number in radians (in `double`) and degrees (in `int`) respectively.

  ```
  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`.

- A method `conjugate()` that returns a new `MyComplex` instance containing the complex conjugate of this instance.

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

  Hint:

  ```
  return new MyComplex(real, -imag);  // construct a new instance and
  return the constructed instance
  ```

- Methods `add(MyComplex another)` and `subtract(MyComplex another)` that adds and subtract this instance with the given `MyComplex` instance `another`, and returns a new `MyComplex` instance containing the result.

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

- Methods `multiplyWith(MyComplex another)` and `divideBy(MyComplex another)` that multiplies and divides this instance with the given `MyComplex` instance `another`, keep 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)] / (c2 + d2)
  ```

  Hint:

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

You are required to:

1. Write the `MyComplex` class.
2. Write a test program to test all the 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.

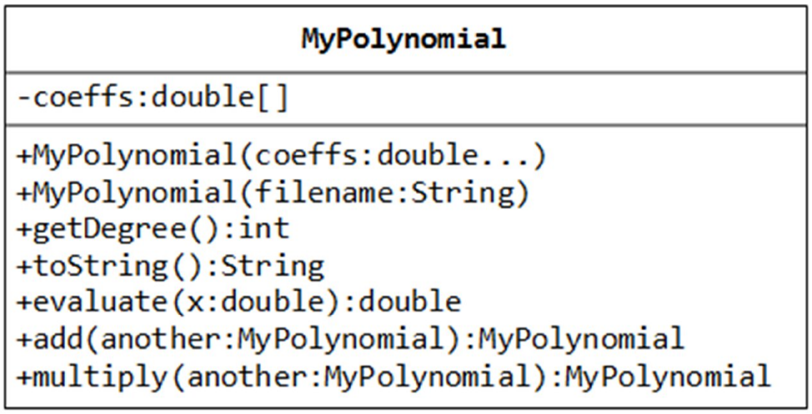
```
4. Enter complex number 1 (real and imaginary part): 1.1 2.2
5. Enter complex number 2 (real and imaginary part): 3.3 4.4
6.
7. Number 1 is: (1.1 + 2.2i)
```

```
8. (1.1 + 2.2i) is NOT a pure real number
9. (1.1 + 2.2i) is NOT a pure imaginary number
10.
11.  Number 2 is: (3.3 + 4.4i)
12.  (3.3 + 4.4i) is NOT a pure real number
13.  (3.3 + 4.4i) is NOT a pure imaginary number
14.
15.  (1.1 + 2.2i) is NOT equal to (3.3 + 4.4i)
16.  (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 `add()`, `subtract()`, and `conjugate()` produce new instances, whereas `multiplyWith()` and `divideBy()` modify this instance. There is inconsistency in the design (introduced for teaching purpose).
- Unusual to have both `argumentInRadians()` and `argumentInDegrees()`.

1.7 Exercise: The MyPolynomial Class


MyPolynomial
-coeffs:double[]
+MyPolynomial(coeffs:double...)
+MyPolynomial(filename:String)
+getDegree():int
+toString():String
+evaluate(x:double):double
+add(another:MyPolynomial):MyPolynomial
+multiply(another:MyPolynomial):MyPolynomial

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

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

The class contains:

- An instance variable named `coeffs`, which stores the coefficients of the n-degree polynomial in a `double` array of size `n+1`, where c_0 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 c_0. 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);
```

- Another constructor that takes coefficients from a file (of the given `filename`), having this format:
- `Degree-n(int)`
- `c0(double)`
- `c1(double)`
- `......`
- `......`
- `cn-1(double)`
- `cn(double)`
- `(end-of-file)`

  Hints:

```
public MyPolynomial(String filename) {
   Scanner in = null;
   try {
      in = new Scanner(new File(filename));  // open file
   } catch (FileNotFoundException e) {
      e.printStackTrace();
   }
   int degree = in.nextInt();      // read the degree
   coeffs = new double[degree+1];  // allocate the array
   for (int i=0; i<coeffs.length; ++i) {
      coeffs[i] = in.nextDouble();
   }
}
```

- A method `getDegree()` that returns the degree of this polynomial.
- A method `toString()` that returns "c_nx^n+c_{n-1}x^(n-1)+...+c_1x+c_0".
- 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 a new `MyPolynomial` instance that contains the result.

Write the `MyPolynomial` class. Also write a test program (called `TestMyPolynomial`) to test all the 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?

1.8 Exercise: 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`.

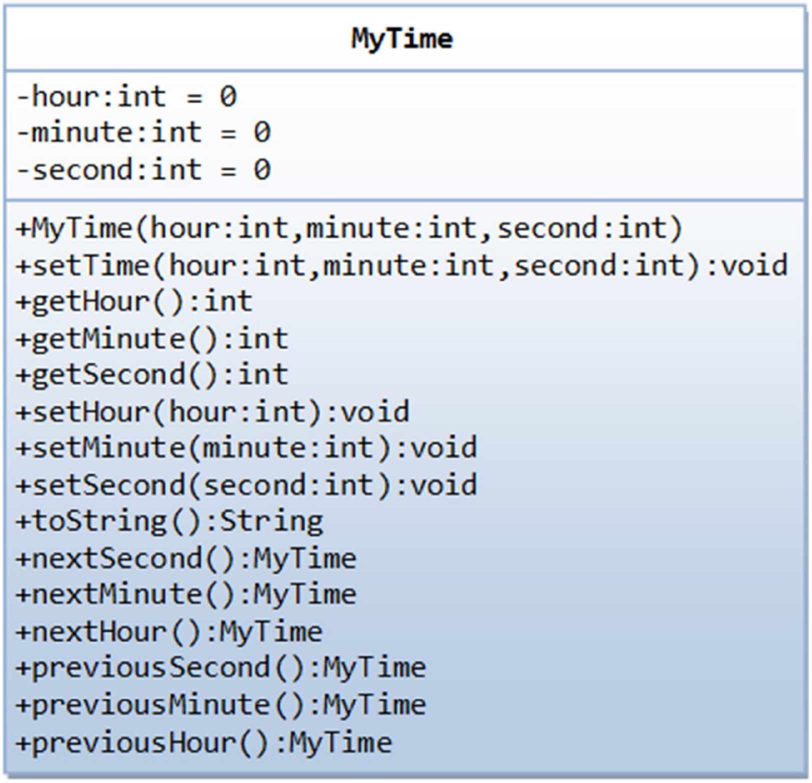
Write a program called `TestBigInteger` that:

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));
      .......
   }
}
```

1.9 Exercise: The MyTime Class



```
MyTime
-hour:int = 0
-minute:int = 0
-second:int = 0
+MyTime(hour:int,minute:int,second:int)
+setTime(hour:int,minute:int,second:int):void
+getHour():int
+getMinute():int
+getSecond():int
+setHour(hour:int):void
+setMinute(minute:int):void
+setSecond(second:int):void
+toString():String
+nextSecond():MyTime
+nextMinute():MyTime
+nextHour():MyTime
+previousSecond():MyTime
+previousMinute():MyTime
+previousHour():MyTime
```

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.

The constructor shall invoke the `setTime()` method (to be described later) to set the instance variable.

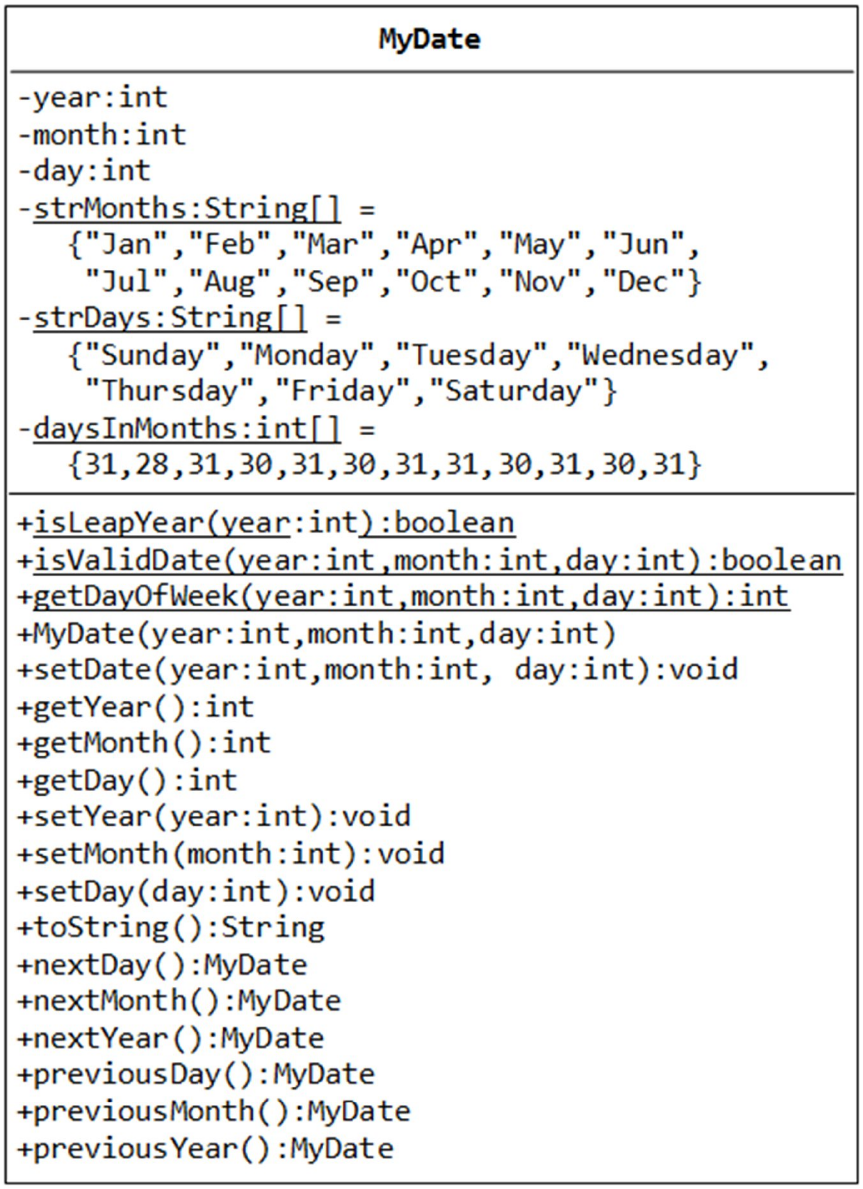
It contains the following `public` methods:

- `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 program (called `TestMyTime`) to test all the methods defined in the `MyTime` class.

1.10 Exercise: The MyDate Class



```
MyDate
-year:int
-month:int
-day:int
-strMonths:String[] =
   {"Jan","Feb","Mar","Apr","May","Jun",
    "Jul","Aug","Sep","Oct","Nov","Dec"}
-strDays:String[] =
   {"Sunday","Monday","Tuesday","Wednesday",
    "Thursday","Friday","Saturday"}
-daysInMonths:int[] =
   {31,28,31,30,31,30,31,31,30,31,30,31}
+isLeapYear(year:int):boolean
+isValidDate(year:int,month:int,day:int):boolean
+getDayOfWeek(year:int,month:int,day:int):int
+MyDate(year:int,month:int,day:int)
+setDate(year:int,month:int, day:int):void
+getYear():int
+getMonth():int
+getDay():int
+setYear(year:int):void
+setMonth(month:int):void
+setDay(day:int):void
+toString():String
+nextDay():MyDate
+nextMonth():MyDate
+nextYear():MyDate
+previousDay():MyDate
+previousMonth():MyDate
+previousYear():MyDate
```

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 (or Wiki "Determination of 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., "Tuesday 14 Feb 2012".
- `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.

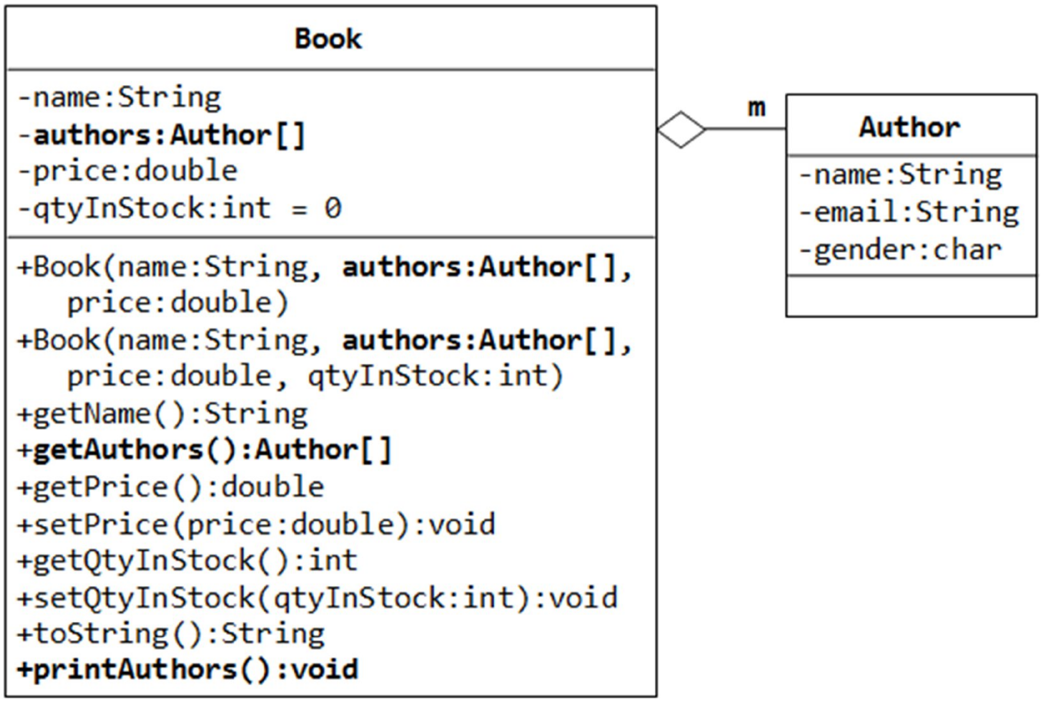
Write the code for the `MyDate` class.

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`.

1.11 Exercise: Book and Author Classes Again - An Array of Objects as an Instance Variable



In the earlier exercise, a book is written by one and only one author. In reality, a book can be written by one or more author. Modify the `Book` class to support one or more authors by changing the instance variable `authors` to an `Author` array. Reuse the Author class written earlier.

Notes:

- The constructors take an array of `Author` (i.e., `Author[]`), instead of an `Author` instance.
- The `toString()` method shall return "book-name by *n* authors", where *n* is the number of authors.
- A new method `printAuthors()` to print the names of all the authors.

You are required to:

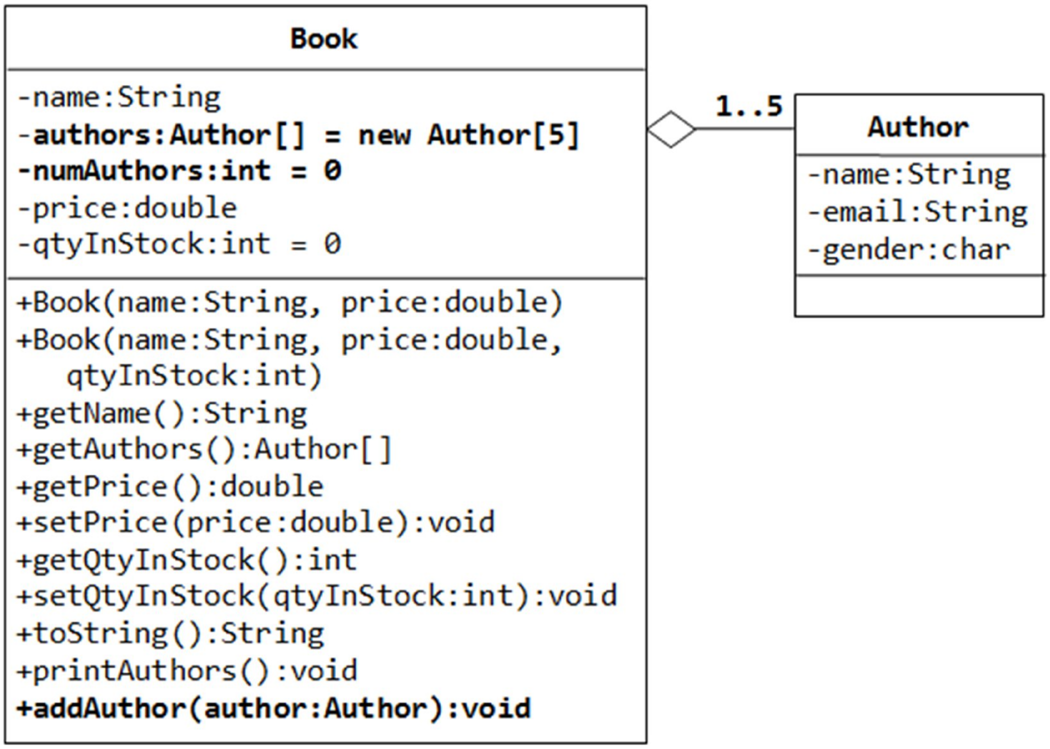
1. Write the code for the `Book` class. You shall re-use the `Author` class written earlier.
2. Write a test program (called `TestBook`) to test the `Book` class.

Hints:

```
// Declare and allocate an array of Authors
Author[] authors = new Author[2];
authors[0] = new Author("Tan Ah Teck", "AhTeck@somewhere.com", 'm');
authors[1] = new Author("Paul Tan", "Paul@nowhere.com", 'm');

// Declare and allocate a Book instance
```

```
Book javaDummy = new Book("Java for Dummy", authors, 19.99, 99);
System.out.println(javaDummy);  // toString()
System.out.print("The authors are: ");
javaDummy.printAuthors();
```

1.12 Exercise: Book and Author Classes Once More - A Fixed-length Array of Objects as an Instance Variable



In the above exercise, the number of authors cannot be changed once a `Book` instance is constructed. Suppose that we wish to allow the user to add more authors (which is really unusual but presented here for academic purpose).

We shall remove the `authors` from the constructors, and add a new method called `addAuthor()` to add the given `Author` instance to this `Book`.

We also need to pre-allocate an `Author` array, with a fixed length (says 5 - a book is written by 1 to 5 authors), and use another instance variable `numAuthors` (`int`) to keep track of the actual number of authors.

You are required to:

1. Modify your `Book` class to support this new requirement.
   Hints:

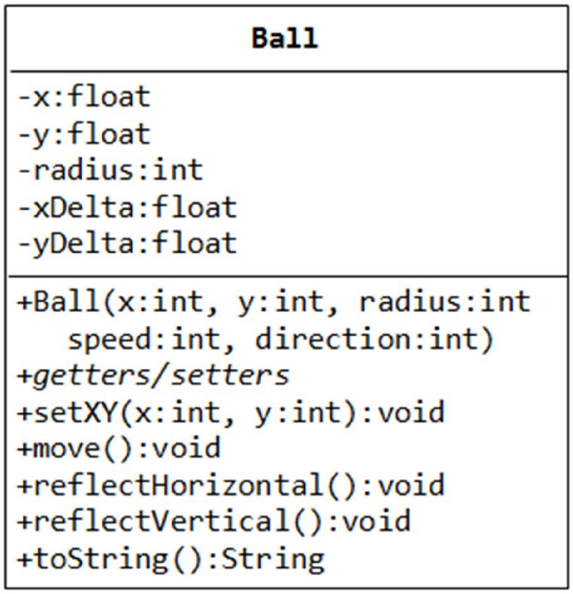
```
public class Book {
   // private instance variable
   private Author[] authors = new Author[5]; // declare and allocate the array
                                             // BUT not the element's instance
   private int numAuthors = 0;

   ......
   ......

   public void addAuthor(Author author) {
      authors[numAuthors] = author;
      ++numAuthors;
   }
}

// Test program
Book javaDummy = new Book("Java for Dummy", 19.99, 99);
System.out.println(javaDummy);  // toString()
System.out.print("The authors are: ");
javaDummy.printAuthors();

javaDummy.addAuthor(new Author("Tan Ah Teck", "AhTeck@somewhere.com", 'm'));
javaDummy.addAuthor(new Author("Paul Tan", "Paul@nowhere.com", 'm'));
System.out.println(javaDummy);  // toString()
System.out.print("The authors are: ");
javaDummy.printAuthors();
```

27. Try writing a method called `removeAuthorByName(authorName)`, that remove the author from this `Book` instance if `authorName` is present. The method shall return `true` if it succeeds.

```
boolean removeAuthorByName(String authorName)
```

Advanced Note: Instead of using a fixed-length array in this case, it is better to be a dynamically allocated array (e.g., `ArrayList`), which does not have a fixed length.

1.13 Exercise: Bouncing Balls - Ball and Container Classes



```
Ball
-x:float
-y:float
-radius:int
-xDelta:float
-yDelta:float
+Ball(x:int, y:int, radius:int
   speed:int, direction:int)
+getters/setters
+setXY(x:int, y:int):void
+move():void
+reflectHorizontal():void
+reflectVertical():void
+toString():String
```

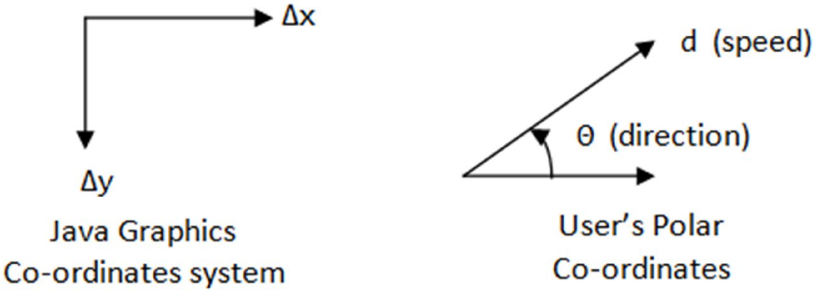
A class called `Ball` is designed as shown in the class diagram.

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.



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

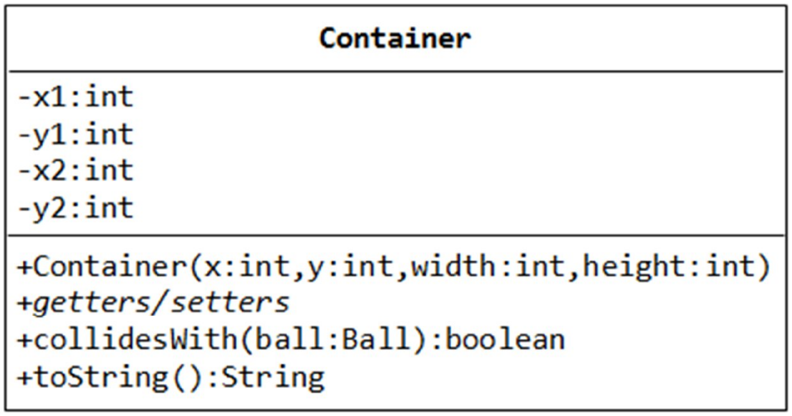
- Getter and setter for all the instance variables.
- A method `move()` which move the ball by one step.
- `x += Δx`
  `y += Δy`

- `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)
+getters/setters
+collidesWith(ball:Ball):boolean
+toString():String

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
}
```