Homework 4. OOP Exercises

Writing Good Programs

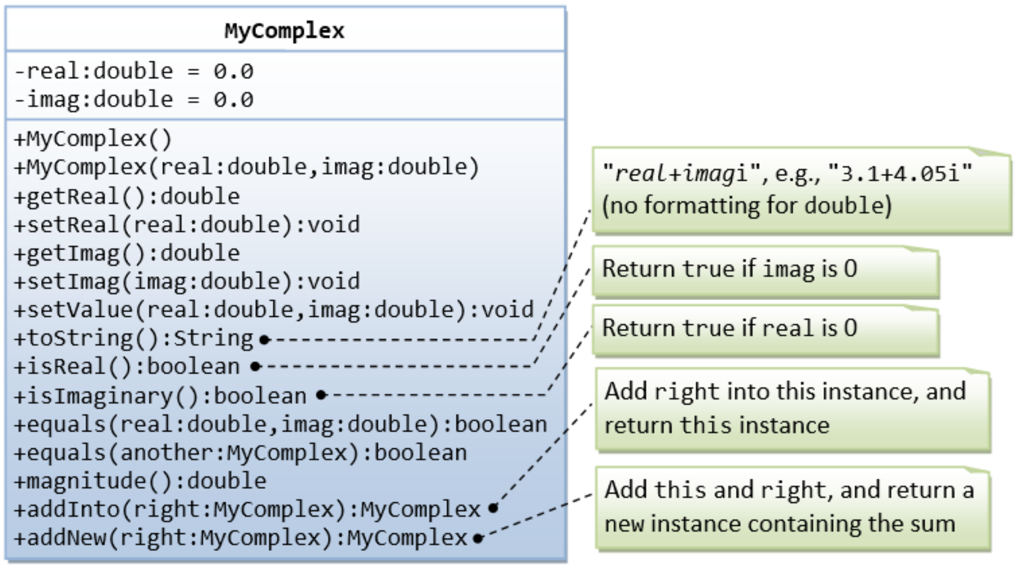
The only way to learn programming is program, program and program. Learning programming is like learning cycling, swimming or any other sports. You can't learn by watching or reading books. Start to program immediately. On the other hands, to improve your programming, you need to read many books and study how the masters program.
It is easy to write programs that work. It is much harder to write programs that not only work but also easy to maintain and understood by others – I call these good programs. In the real world, writing program is not meaningful. You have to write good programs, so that others can understand and maintain your programs.
Pay particular attention to:

1. **Coding style**:
   - Read Java code convention: "Java Style and Commenting Guide".
   - Follow the Java Naming Conventions for variables, methods, and classes STRICTLY. Use CamelCase for names. Variable and method names begin with lowercase, while class names begin with uppercase. Use nouns for variables (e.g., radius) and class names (e.g., Circle). Use verbs for methods (e.g., getArea(), isEmpty()).
   - **Use Meaningful Names**: Do not use names like a, b, c, d, x, x1, x2, and x1688 - they are meaningless. Avoid single-alphabet names like i, j, k. They are easy to type, but usually meaningless. Use single-alphabet names only when their meaning is clear, e.g., x, y, z for co-ordinates and i for array index. Use meaningful names like row and col (instead of x and y, i and j, x1 and x2), numStudents (not n), maxGrade, size (not n), and upperbound (not n again). Differentiate between singular and plural nouns (e.g., use books for an array of books, and book for each item).
   - Use consistent indentation and coding style. Many IDEs (such as Eclipse / NetBeans) can re-format your source codes with a single click.
2. **Program Documentation**: Comment! Comment! and more Comment to explain your code to other people and to yourself three days later.
3. The only way to learn programming is program, program and program on challenging problems. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests (such as International Collegiate Programming Contest (ICPC), International Olympiad in Informatics (IOI)).

1 More Exercises on Classes

1.1 The MyComplex class

A class called MyComplex, which models a complex number with real and imaginary parts, is designed as shown in the class diagram.


MyComplex
-real:double = 0.0
-imag:double = 0.0
+MyComplex()
+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
+addInto(right:MyComplex):MyComplex
+addNew(right:MyComplex):MyComplex
"real+imagi", e.g., "3.1+4.05i" (no formatting for double)
Return true if imag is 0
Return true if real is 0
Add right into this instance, and return this instance
Add this and right, and return a new instance containing the sum

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

```java
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

  ```java
  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

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

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

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

- Methods *addInto(MyComplex right)* that adds and subtract the given MyComplex instance (called right) into this instance and returns this instance.

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

  Hints

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

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

  Hints

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

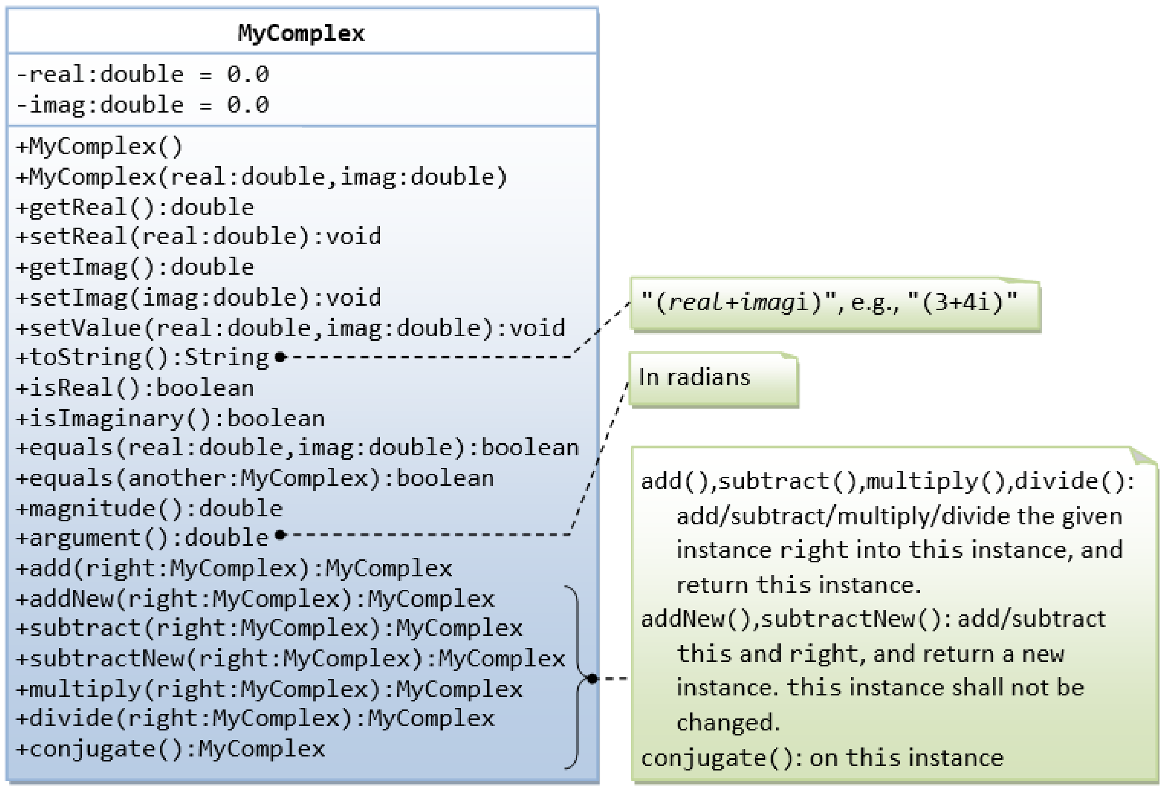
You are required to:

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.

   Command window

   ```
   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)
   ```

Try
A (more) complete design of MyComplex class is shown below:



```
MyComplex
-real:double = 0.0
-imag:double = 0.0
+MyComplex()
+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
+argument():double
+add(right:MyComplex):MyComplex
+addNew(right:MyComplex):MyComplex
+subtract(right:MyComplex):MyComplex
+subtractNew(right:MyComplex):MyComplex
+multiply(right:MyComplex):MyComplex
+divide(right:MyComplex):MyComplex
+conjugate():MyComplex
```

"(*real*+*imag*i)", e.g., "(3+4i)"

In radians

add(),subtract(),multiply(),divide(): add/subtract/multiply/divide the given instance right into this instance, and return this instance.
addNew(),subtractNew(): add/subtract this and right, and return a new instance. this instance shall not be changed.
conjugate(): on this instance

- 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.

- The method *addInto()* is renamed *add()*. Also added *subtract()* and *subtractNew()*.

- 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.

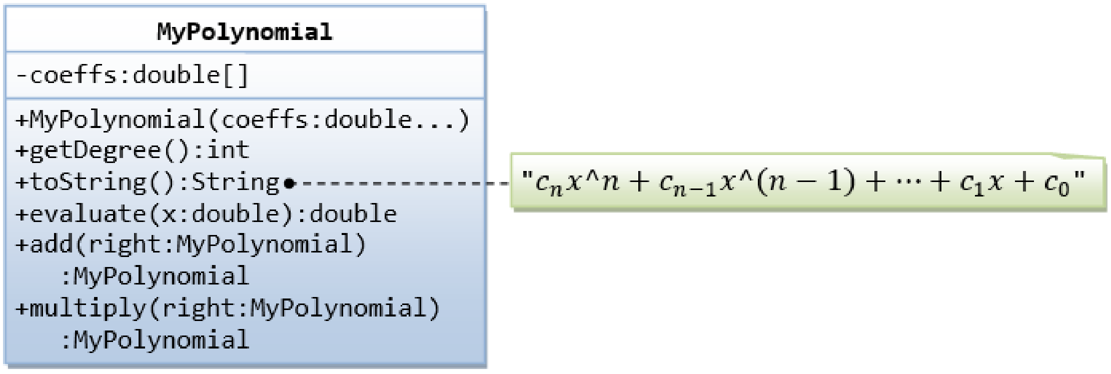
```
conjugate(x+yi) = x - yi
```

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., *complex1.add(complex2).add(complex3)* (same as *(complex1.add(complex2)).add(complex3))*, or *complex1.add(complex2).subtract(complex3)*.

1.2 The MyPolynomial Class


MyPolynomial
-coeffs:double[]
+MyPolynomial(coeffs:double...)
+getDegree():int
+toString():String
+evaluate(x:double):double
+add(right:MyPolynomial) :MyPolynomial
+multiply(right:MyPolynomial) :MyPolynomial
"c_nx^$n + c_{n-1}x$^$(n-1) + \cdots + c_1x + c_0$"

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

$$c_nx^n + c_{n-1}x^{n-1} + \cdots + c_1x + c_0.$$

It contains:

- 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

```java
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 polynomial1 = new MyPolynomial(1.1, 2.2, 3.3);
MyPolynomial polynomial1 = 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 polynomial2 = new MyPolynomial(coeffs);
```

- A method *getDegree()* that returns the degree of this polynomial.
- A method *toString()* that returns "$c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$.".
- 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 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?

1.3 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:

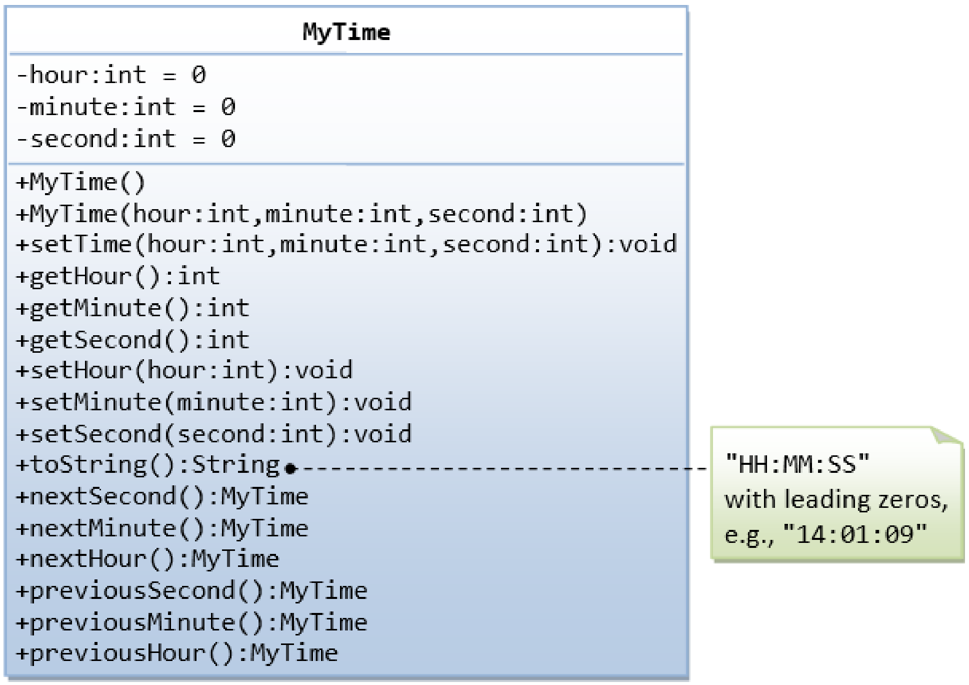
1. adds ”11111111111111111111111111111111111111111111111111111111111111”
   to ”22222222222222222222222222222222222222222222222222” and prints the result.

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

Hints

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

1.4 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.

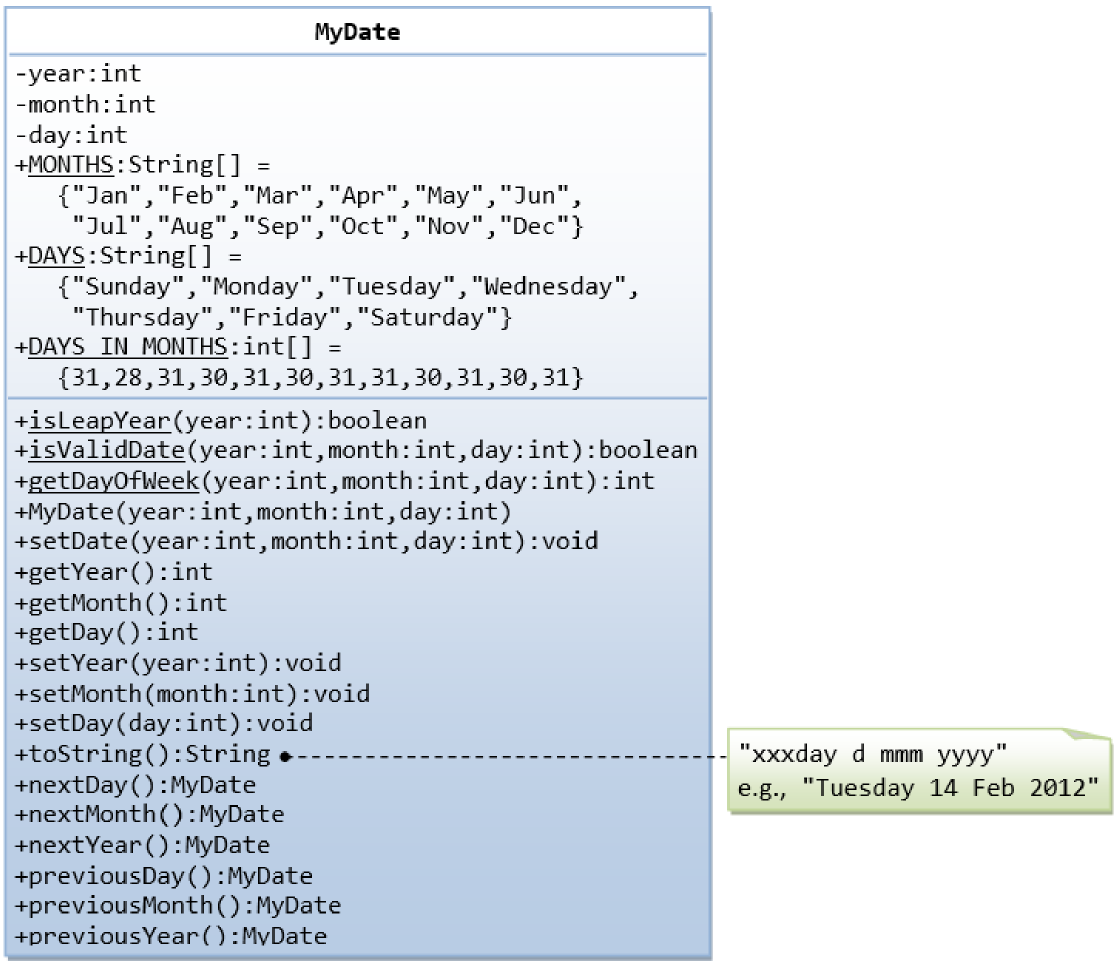
You are required to perform input validation.
It contains the following public methods:

- *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.

1.5 The MyDate Class


MyDate
-year:int
-month:int
-day:int
+MONTHS:String[] =
{"Jan","Feb","Mar","Apr","May","Jun",
"Jul","Aug","Sep","Oct","Nov","Dec"}
+DAYS:String[] =
{"Sunday","Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday"}
+DAYS_IN_MONTHS: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
"xxxday d mmm yyyy"
e.g., "Tuesday 14 Feb 2012"

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 public static final variables (drawn with underlined in the class diagram):

- MONTHS (*String*[ ]), DAYS (*String*[ ]), and DAY_IN_MONTHS (*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.)

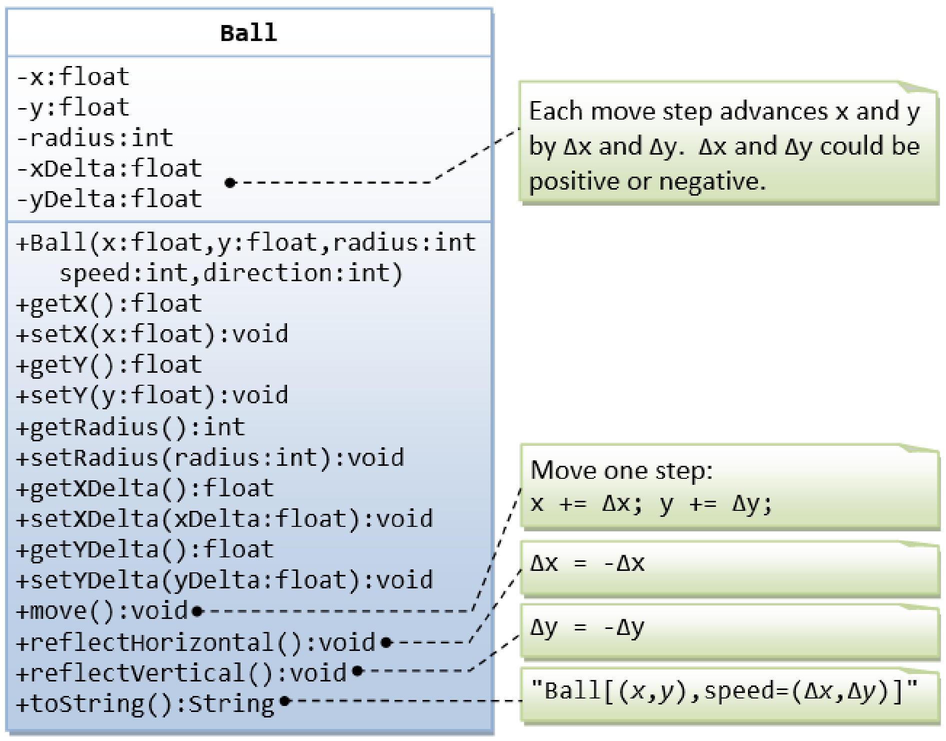
- *previousDay()*, *previousMonth()*, *previousYear()*: similar to the above.

Write the code for the MyDate class.
Use the following test statements to test the MyDate class:

```java
MyDate date1 = new MyDate(2012, 2, 28);
System.out.println(date1);               // Tuesday 28 Feb 2012
System.out.println(date1.nextDay());     // Wednesday 29 Feb 2012
System.out.println(date1.nextDay());     // Thursday 1 Mar 2012
System.out.println(date1.nextMonth());   // Sunday 1 Apr 2012
System.out.println(date1.nextYear());    // Monday 1 Apr 2013

MyDate date2 = new MyDate(2012, 1, 2);
System.out.println(date2);                   // Monday 2 Jan 2012
System.out.println(date2.previousDay());     // Sunday 1 Jan 2012
System.out.println(date2.previousDay());     // Saturday 31 Dec 2011
System.out.println(date2.previousMonth());   // Wednesday 30 Nov 2011
System.out.println(date2.previousYear());    // Tuesday 30 Nov 2010

MyDate date3 = new MyDate(2012, 2, 29);
System.out.println(date3.previousYear());    // Monday 28 Feb 2011

// MyDate date4 = new MyDate(2099, 11, 31); // Invalid year, month, or
    ↪ day!
// MyDate date5 = 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.

1.6 Bouncing Balls - Ball and Container Classes


Ball
-x:float
-y:float
-radius:int
-xDelta:float
-yDelta:float
+Ball(x:float,y:float,radius:int
speed:int,direction:int)
+getX():float
+setX(x:float):void
+getY():float
+setY(y:float):void
+getRadius():int
+setRadius(radius:int):void
+getXDelta():float
+setXDelta(xDelta:float):void
+getYDelta():float
+setYDelta(yDelta:float):void
+move():void
+reflectHorizontal():void
+reflectVertical():void
+toString():String
Each move step advances x and y by Δx and Δy. Δx and Δy could be positive or negative.
Move one step: x += Δx; y += Δy;
Δx = -Δx
Δy = -Δy
"Ball[(x,y),speed=(Δx,Δy)]"

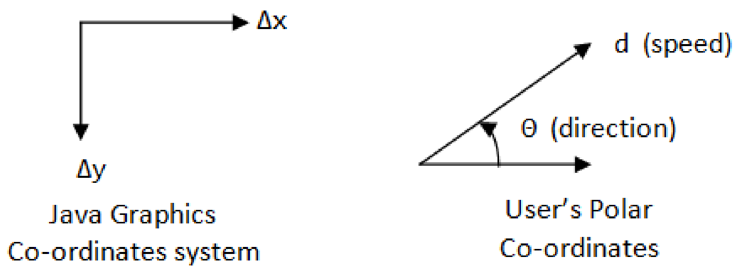
A class called Ball is designed as shown in the class diagram.

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 $(\Delta x, \Delta y)$ 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.



$$\Delta x = d\cos(\theta)$$
$$\Delta y = d\sin(\theta)$$

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

$$x \mathrel{+}= \Delta x$$
$$y \mathrel{+}= \Delta y$$

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

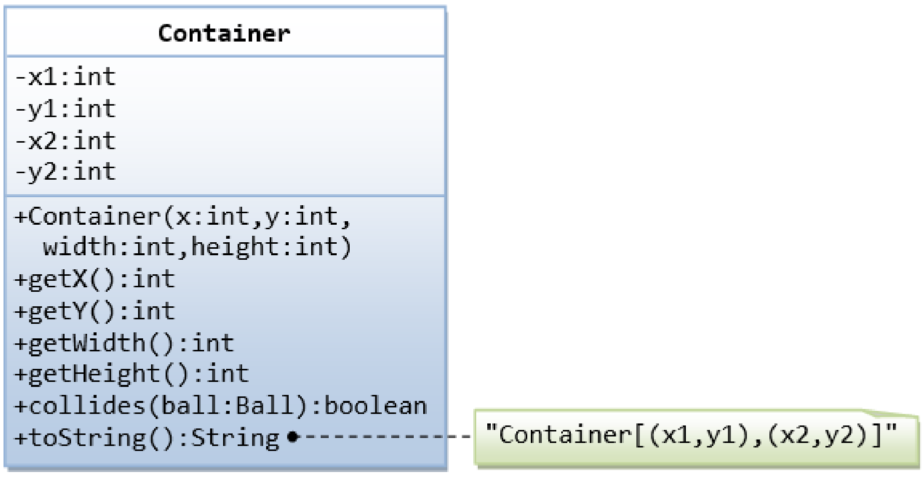
$$\begin{aligned}\Delta x &= -\Delta x \\ \Delta y &\ \text{no changes}\end{aligned}$$

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

$$\begin{aligned}\Delta x &\ \text{no changes} \\ \Delta y &= -\Delta y\end{aligned}$$

- *toString()* which prints the message "Ball at (x, y) of velocity $(\Delta x, \Delta y)$".

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



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.

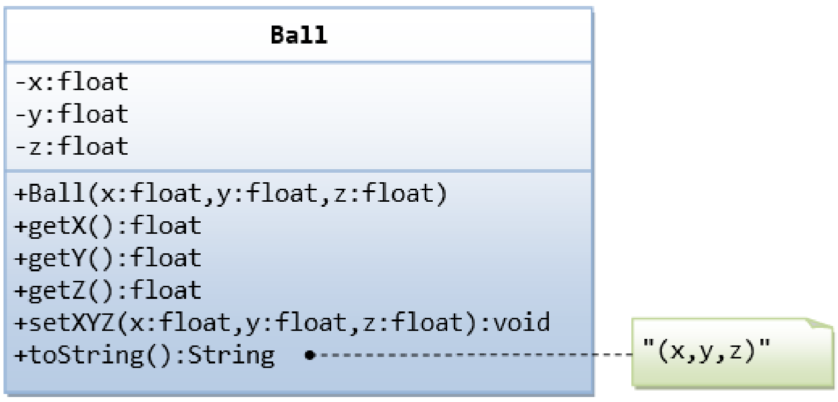
- A constructor which accepts (x, y) of the top-left corner, width and height as argument, and converts them into the internal representation (i.e., $x2 = x1 + \ width \ - 1$). Width and height is used in the argument for safer operation (there is no need to check the validity of $x2 > x1$, etc.).
- 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.

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

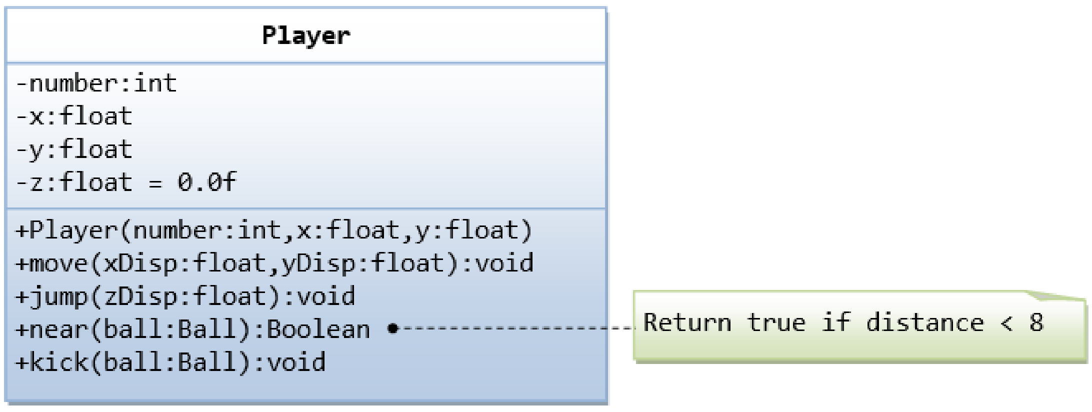
Use the following statements to test your program:

```java
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
}
```

1.7 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?