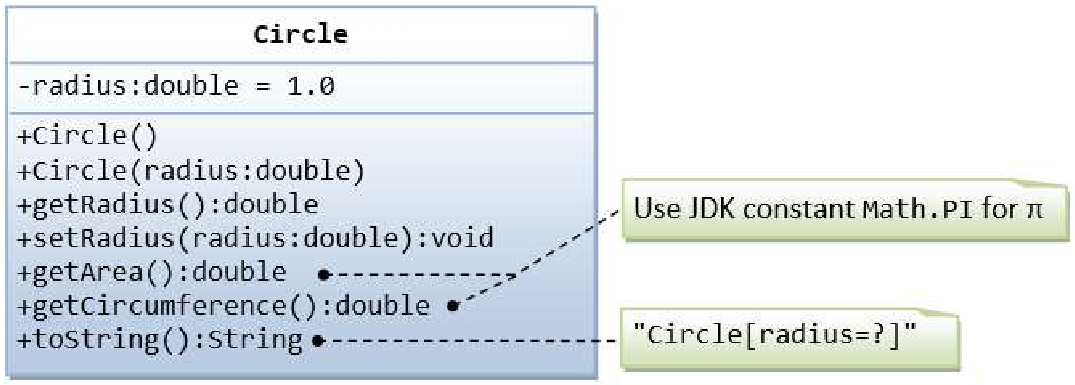
1.1 An Introduction to Classes and Instances by Example - The `Circle` Class

This first exercise shall lead you through all the *basic concepts* in OOP.



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

- Two `private` instance variables: `radius` (of the type `double`) and `color` (of the type `String`), with default value of `1.0` and "`red`", respectively.
- Two *overloaded* constructors - a *default* constructor with no argument, and a constructor which takes a double argument for radius.
- Two `public` methods: `getRadius()` and `getArea()`, which return the radius and area of this instance, respectively.

The source codes for `Circle.java` is as follows:

```
/**
 * The Circle class models a circle with a radius and color.
 */
public class Circle {  // Save as "Circle.java"
   // private instance variable, not accessible from outside this class
   private double radius;
   private String color;

   // Constructors (overloaded)
   /** Constructs a Circle instance with default value for radius and color */
   public Circle() {  // 1st (default) constructor
      radius = 1.0;
      color = "red";
   }

   /** Constructs a Circle instance with the given radius and default color */
   public Circle(double r) {  // 2nd constructor
      radius = r;
      color = "red";
   }

   /** Returns the radius */
   public double getRadius() {
     return radius;
   }

   /** Returns the area of this Circle instance */
   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` (in another source file called `TestCircle.java`) which uses the `Circle` class, as follows:

```
/**
 *  A Test Driver for the Circle class
 */
public class TestCircle {  // Save as "TestCircle.java"
   public static void main(String[] args) {
      // Declare an instance of Circle class called c1.
      // Construct the instance c1 by invoking the "default" constructor
      // which sets its radius and color to their default value.
      Circle c1 = new Circle();
      // Invoke public methods on instance c1, via dot operator.
      System.out.println("The circle has radius of "
         + c1.getRadius() + " and area of " + c1.getArea());
      //The circle has radius of 1.0 and area of 3.141592653589793

      // Declare an instance of class circle called c2.
      // Construct the instance c2 by invoking the second constructor
      // with the given radius and default color.
      Circle c2 = new Circle(2.0);
      // Invoke public methods on instance c2, via dot operator.
      System.out.println("The circle has radius of "
         + c2.getRadius() + " and area of " + c2.getArea());
      //The circle has radius of 2.0 and area of 12.566370614359172
   }
}
```

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

More Basic OOP Concepts

1. **Constructor:** Modify the class `Circle` to include a third constructor for constructing a `Circle` instance with two arguments - a `double` for `radius` and a `String` for `color`.

   ```
   // 3rd constructor 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.

2. **Getter:** Add a getter for variable `color` for retrieving the `color` of this instance.

   ```
   // Getter for instance variable color
   public String getColor() { ...... }
   ```

   Modify the test program to test this method.

3. **`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.

4. **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:

   ```
   // Setter for instance variable radius
   public void setRadius(double newRadius) {
      radius = newRadius;
   }

   // Setter for instance variable color
   public void setColor(String newColor) { ...... }
   ```

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

```
Circle c4 = new Circle();   // construct an instance of Circle
c4.setRadius(5.5);          // change radius
System.out.println("radius is: " + c4.getRadius()); // Print radius via getter
c4.setColor("green");       // Change color
System.out.println("color is: " + c4.getColor());   // Print color via getter

// You cannot do the following because setRadius() returns void, which cannot be printed
System.out.println(c4.setRadius(4.4));
```

5. **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,

```
// Instance variable
private double radius;

/** Constructs a Circle instance with the given radius and default color */
public Circle(double radius) {
   this.radius = radius;   // "this.radius" refers to the instance variable
                           // "radius" refers to the method's parameter
   color = "red";
}

/** Sets the radius to the given value */
public void setRadius(double radius) {
   this.radius = radius;   // "this.radius" refers to the instance variable
                           // "radius" refers to the method's argument
}
```

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

6. **Method `toString()`:** Every well-designed Java class should contain a `public` method called `toString()` that returns a description of the instance (in the 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:

```
/** Return a self-descriptive string of this instance in the form of Circle[radius=?,color=?] */
public String toString() {
   return "Circle[radius=" + radius + " color=" + color + "]";
}
```

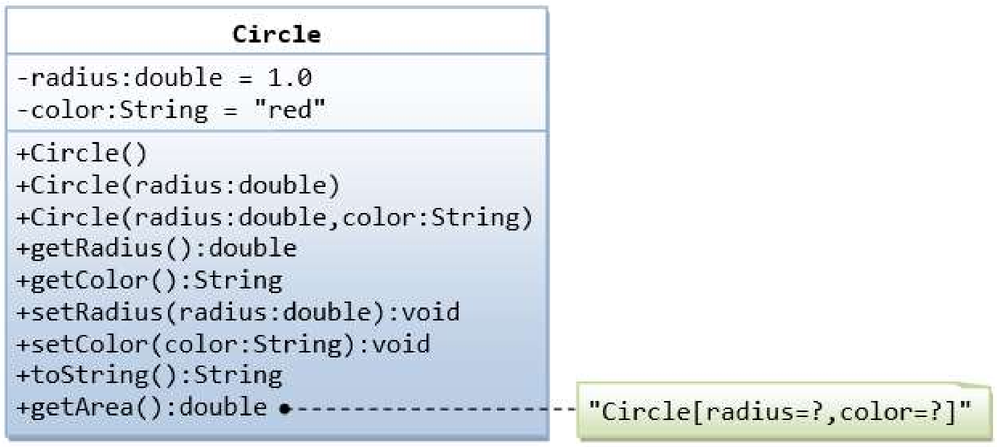
   Try calling `toString()` method explicitly, just like any other method:

```
Circle c5 = new Circle(5.5);
System.out.println(c5.toString());   // explicit call
```

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

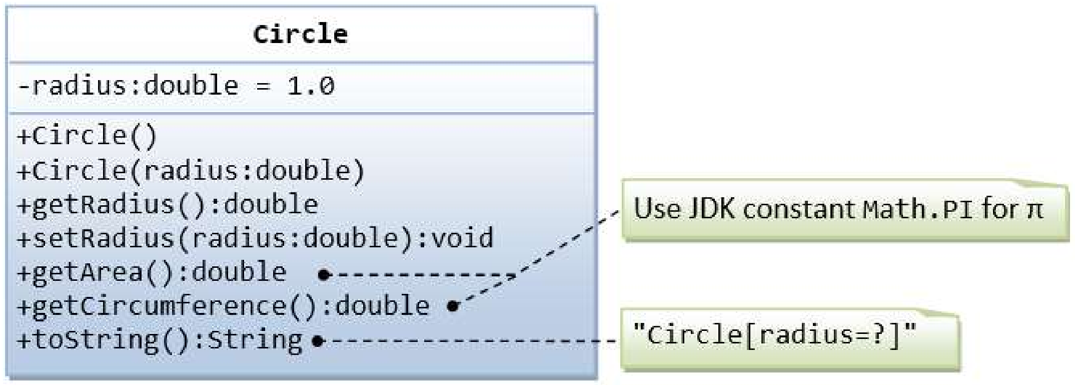
```
Circle c6 = new Circle(6.6);
System.out.println(c6.toString());  // explicit call
System.out.println(c6);             // println() calls toString() implicitly, same as above
System.out.println("Operator '+' invokes toString() too: " + c6);  // '+' invokes toString() too
```

The final class diagram for the `Circle` class is as follows:



1.2 Ex: Yet Another `Circle` Class

A class called `Circle`, which models a circle with a radius, is designed as shown in the following class diagram. Write the `Circle` class.



Below is a Test Driver to test your `Circle` class.

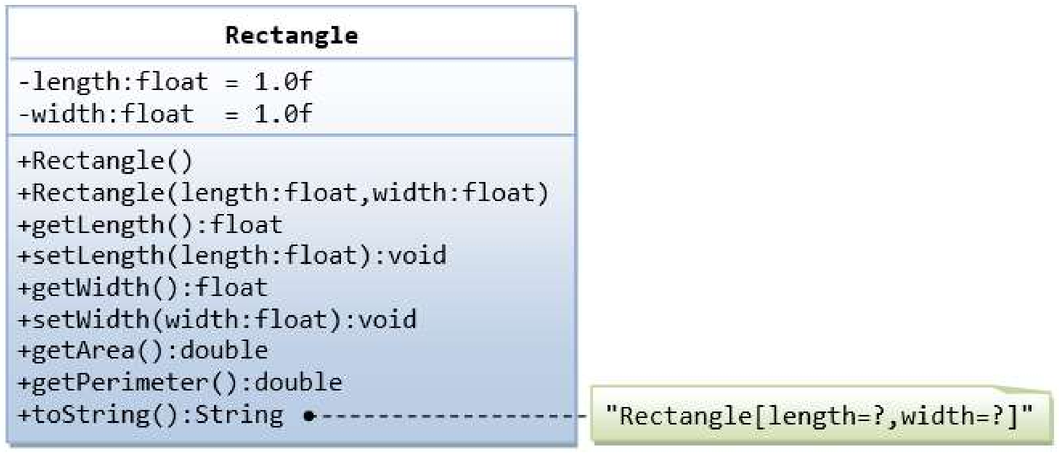
```
public class TestMain {
   public static void main(String[] args) {
      // Test Constructors and toString()
      Circle c1 = new Circle(1.1);
      System.out.println(c1);   // toString()
      Circle c2 = new Circle(); // default constructor
      System.out.println(c2);

      // Test setter and getter
      c1.setRadius(2.2);
      System.out.println(c1);      // toString()
      System.out.println("radius is: " + c1.getRadius());

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

The expected output is:

```
Circle[radius=1.1]
Circle[radius=1.0]
Circle[radius=2.2]
radius is: 2.2
area is: 15.21
circumference is: 13.82
```

1.3 Ex: The Rectangle Class

A class called Rectangle, which models a rectangle with a length and a width (in float), is designed as shown in the following class diagram. Write the Rectangle class.



Below is a test driver to test the Rectangle class:

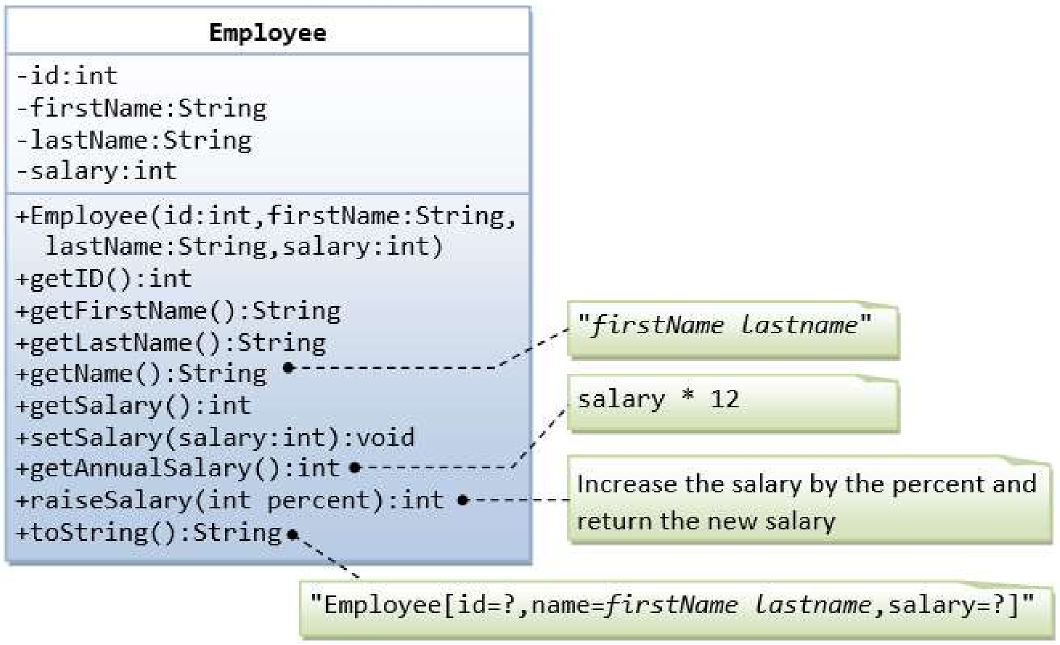
```
public class TestMain {
   public static void main(String[] args) {
      // Test constructors and toString()
      // You need to append a 'f' or 'F' to a float literal
      Rectangle r1 = new Rectangle(1.2f, 3.4f);
      System.out.println(r1);  // toString()
      Rectangle r2 = new Rectangle();  // default constructor
      System.out.println(r2);

      // Test setters and getters
      r1.setLength(5.6f);
      r1.setWidth(7.8f);
      System.out.println(r1);  // toString()
      System.out.println("length is: " + r1.getLength());
      System.out.println("width is: " + r1.getWidth());

      // Test getArea() and getPerimeter()
      System.out.printf("area is: %.2f%n", r1.getArea());
      System.out.printf("perimeter is: %.2f%n", r1.getPerimeter());
   }
}
```

The expected output is:

```
Rectangle[length=1.2,width=3.4]
Rectangle[length=1.0,width=1.0]
Rectangle[length=5.6,width=7.8]
length is: 5.6
width is: 7.8
area is: 43.68
perimeter is: 26.80
```

1.4 Ex: The Employee Class

A class called Employee, which models an employee with an ID, name and salary, is designed as shown in the following class diagram. The method raiseSalary(percent) increases the salary by the given percentage. Write the Employee class.


Employee
-id:int
-firstName:String
-lastName:String
-salary:int
+Employee(id:int,firstName:String,
lastName:String,salary:int)
+getID():int
+getFirstName():String
+getLastName():String
+getName():String
+getSalary():int
+setSalary(salary:int):void
+getAnnualSalary():int
+raiseSalary(int percent):int
+toString():String
"firstName lastname"
salary * 12
Increase the salary by the percent and return the new salary
"Employee[id=?,name=firstName lastname,salary=?]"

Below is a test driver to test the `Employee` class:

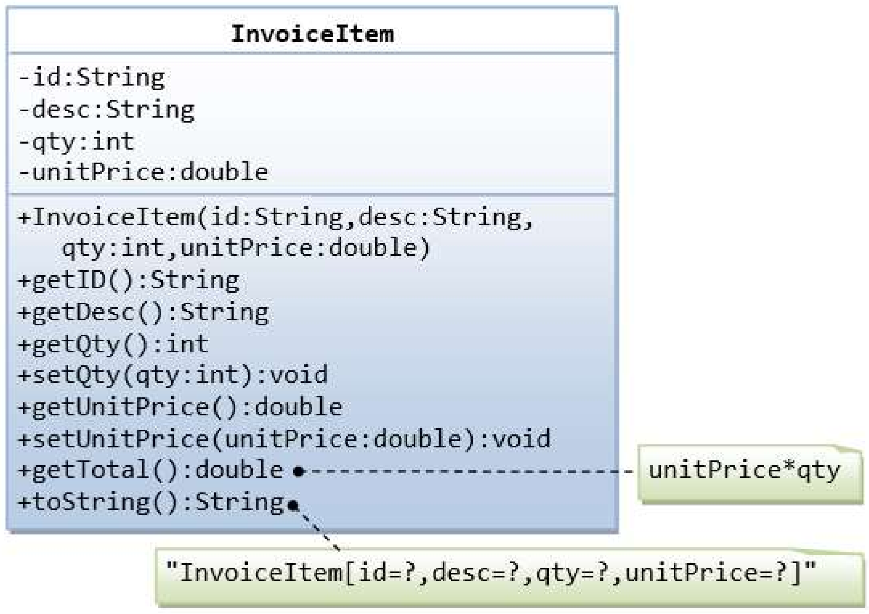
```
public class TestMain {
   public static void main(String[] args) {
      // Test constructor and toString()
      Employee e1 = new Employee(8, "Peter", "Tan", 2500);
      System.out.println(e1);  // toString();

      // Test Setters and Getters
      e1.setSalary(999);
      System.out.println(e1);  // toString();
      System.out.println("id is: " + e1.getID());
      System.out.println("firstname is: " + e1.getFirstName());
      System.out.println("lastname is: " + e1.getLastName());
      System.out.println("salary is: " + e1.getSalary());

      System.out.println("name is: " + e1.getName());
      System.out.println("annual salary is: " + e1.getAnnualSalary()); // Test method

      // Test raiseSalary()
      System.out.println(e1.raiseSalary(10));
      System.out.println(e1);
   }
}
```

The expected out is:

```
Employee[id=8,name=Peter Tan,salary=2500]
Employee[id=8,name=Peter Tan,salary=999]
id is: 8
firstname is: Peter
lastname is: Tan
salary is: 999
name is: Peter Tan
annual salary is: 11988
1098
Employee[id=8,name=Peter Tan,salary=1098]
```

1.5 Ex: The `InvoiceItem` Class

A class called `InvoiceItem`, which models an item of an invoice, with ID, description, quantity and unit price, is designed as shown in the following class diagram. Write the `InvoiceItem` class.



Below is a test driver to test the `InvoiceItem` class:

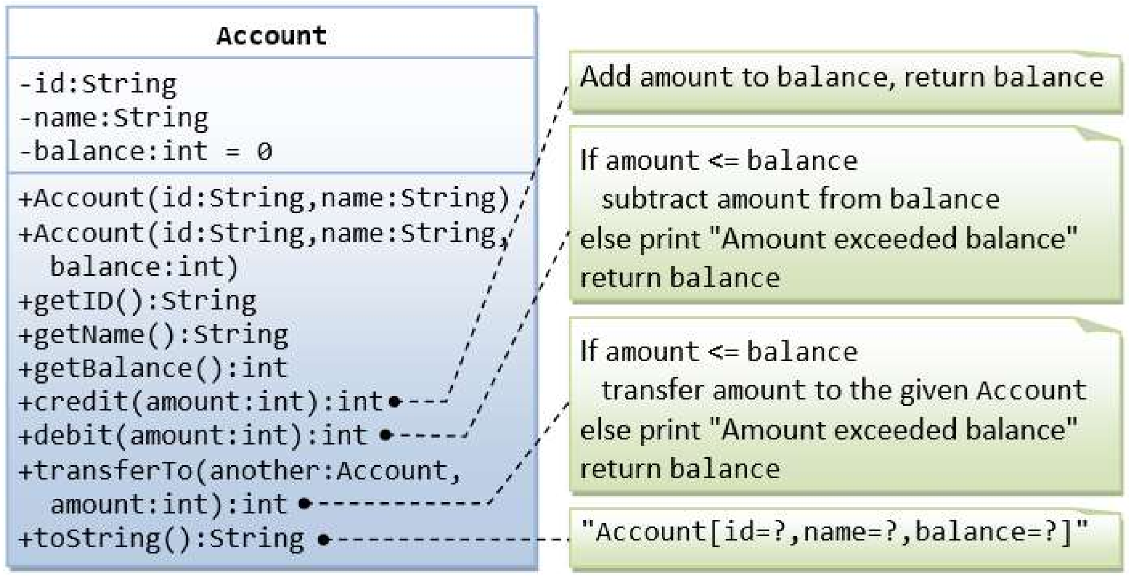
```
public class TestMain {
   public static void main(String[] args) {
      // Test constructor and toString()
      InvoiceItem inv1 = new InvoiceItem("A101", "Pen Red", 888, 0.08);
      System.out.println(inv1);  // toString();

      // Test Setters and Getters
      inv1.setQty(999);
      inv1.setUnitPrice(0.99);
      System.out.println(inv1);  // toString();
      System.out.println("id is: " + inv1.getID());
      System.out.println("desc is: " + inv1.getDesc());
      System.out.println("qty is: " + inv1.getQty());
      System.out.println("unitPrice is: " + inv1.getUnitPrice());

      // Test getTotal()
      System.out.println("The total is: " + inv1.getTotal());
   }
}
```

The expected output is:

```
InvoiceItem[id=A101,desc=Pen Red,qty=888,unitPrice=0.08]
InvoiceItem[id=A101,desc=Pen Red,qty=999,unitPrice=0.99]
id is: A101
desc is: Pen Red
qty is: 999
unitPrice is: 0.99
The total is: 989.01
```

1.6 Ex: The Account Class

A class called `Account`, which models a bank account of a customer, is designed as shown in the following class diagram. The methods `credit(amount)` and `debit(amount)` add or subtract the given `amount` to the `balance`. The method `transferTo(anotherAccount, amount)` transfers the given `amount` from this `Account` to the given `anotherAccount`. Write the `Account` class.


Account
-id:String
-name:String
-balance:int = 0
+Account(id:String,name:String)
+Account(id:String,name:String, balance:int)
+getID():String
+getName():String
+getBalance():int
+credit(amount:int):int
+debit(amount:int):int
+transferTo(another:Account, amount:int):int
+toString():String
Add amount to balance, return balance
If amount <= balance subtract amount from balance else print "Amount exceeded balance" return balance
If amount <= balance transfer amount to the given Account else print "Amount exceeded balance" return balance
"Account[id=?,name=?,balance=?]"

Below is a test driver to test the Account class:

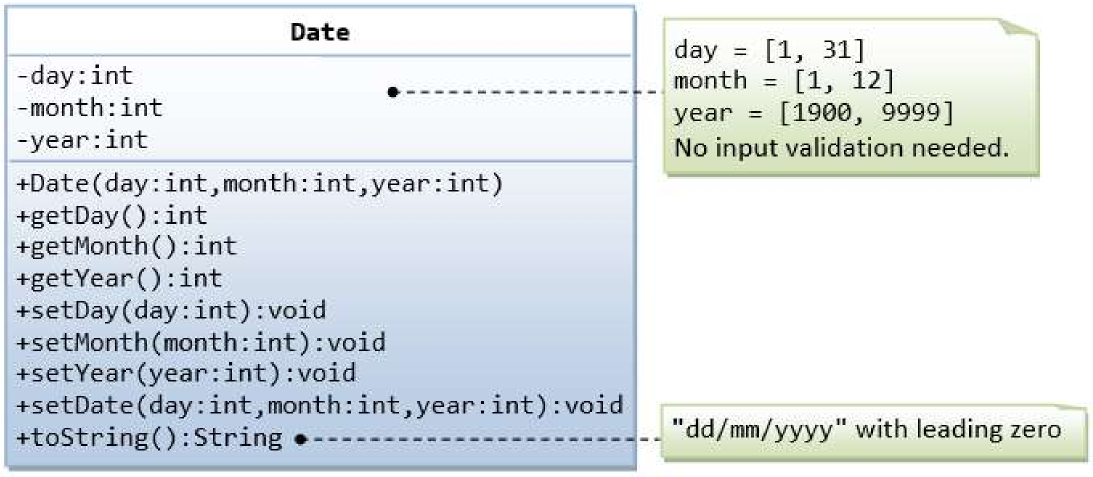
```
public class TestMain {
   public static void main(String[] args) {
      // Test constructor and toString()
      Account a1 = new Account("A101", "Tan Ah Teck", 88);
      System.out.println(a1);  // toString();
      Account a2 = new Account("A102", "Kumar"); // default balance
      System.out.println(a2);

      // Test Getters
      System.out.println("ID: " + a1.getID());
      System.out.println("Name: " + a1.getName());
      System.out.println("Balance: " + a1.getBalance());

      // Test credit() and debit()
      a1.credit(100);
      System.out.println(a1);
      a1.debit(50);
      System.out.println(a1);
      a1.debit(500);  // debit() error
      System.out.println(a1);

      // Test transfer()
      a1.transferTo(a2, 100);  // toString()
      System.out.println(a1);
      System.out.println(a2);
   }
}
```

The expected output is:

```
Account[id=A101,name=Tan Ah Teck,balance=88]
Account[id=A102,name=Kumar,balance=0]
ID: A101
Name: Tan Ah Teck
Balance: 88
Account[id=A101,name=Tan Ah Teck,balance=188]
Account[id=A101,name=Tan Ah Teck,balance=138]
Amount exceeded balance
Account[id=A101,name=Tan Ah Teck,balance=138]
Account[id=A101,name=Tan Ah Teck,balance=38]
Account[id=A102,name=Kumar,balance=100]
```

1.7 Ex: The Date Class

A class called Date, which models a calendar date, is designed as shown in the following class diagram. Write the Date class.



Below is a test driver to test the Date class:

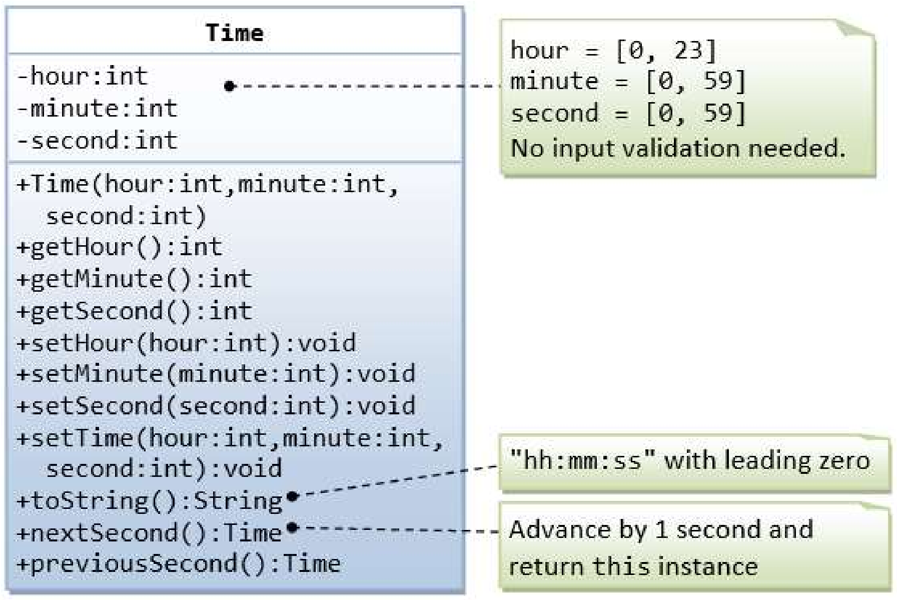
```
public class TestMain {
   public static void main(String[] args) {
      // Test constructor and toString()
      Date d1 = new Date(1, 2, 2014);
      System.out.println(d1);  // toString()

      // Test Setters and Getters
      d1.setMonth(12);
      d1.setDay(9);
      d1.setYear(2099);
      System.out.println(d1);  // toString()
      System.out.println("Month: " + d1.getMonth());
      System.out.println("Day: " + d1.getDay());
      System.out.println("Year: " + d1.getYear());

      // Test setDate()
      d1.setDate(3, 4, 2016);
      System.out.println(d1);  // toString()
   }
}
```

The expected output is:

```
01/02/2014
09/12/2099
Month: 12
Day: 9
Year: 2099
03/04/2016
```

1.8 Ex: The Time Class

A class called Time, which models a time instance, is designed as shown in the following class diagram. The methods nextSecond() and previousSecond() shall advance or rewind this instance by one second, and return this instance, so as to support chaining operation such as t1.nextSecond().nextSecond(). Write the Time class.


Time
-hour:int
-minute:int
-second:int
+Time(hour:int,minute:int, second:int)
+getHour():int
+getMinute():int
+getSecond():int
+setHour(hour:int):void
+setMinute(minute:int):void
+setSecond(second:int):void
+setTime(hour:int,minute:int, second:int):void
+toString():String
+nextSecond():Time
+previousSecond():Time
hour = [0, 23]
minute = [0, 59]
second = [0, 59]
No input validation needed.
"hh:mm:ss" with leading zero
Advance by 1 second and return this instance

Below is a test driver for testing the `Time` class:

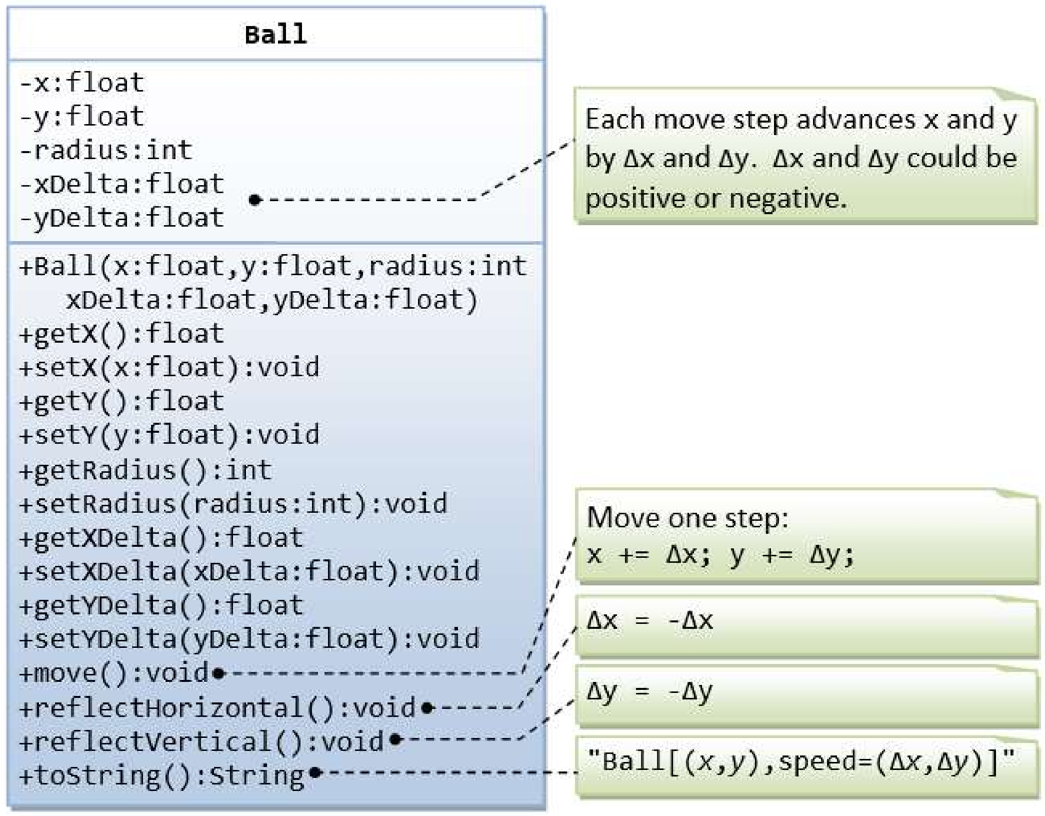
```
public class TestMain {
   public static void main(String[] args) {
      // Test constructors and toString()
      Time t1 = new Time(1, 2, 3);
      System.out.println(t1);  // toString()

      // Test Setters and Getters
      t1.setHour(4);
      t1.setMinute(5);
      t1.setSecond(6);
      System.out.println(t1);  // toString()
      System.out.println("Hour: " + t1.getHour());
      System.out.println("Minute: " + t1.getMinute());
      System.out.println("Second: " + t1.getSecond());

      // Test setTime()
      t1.setTime(23, 59, 58);
      System.out.println(t1);  // toString()

      // Test nextSecond();
      System.out.println(t1.nextSecond());
      System.out.println(t1.nextSecond().nextSecond());

      // Test previousSecond()
      System.out.println(t1.previousSecond());
      System.out.println(t1.previousSecond().previousSecond());
   }
}
```

The expected output is:

```
01:02:03
04:05:06
Hour: 4
Minute: 5
Second: 6
23:59:58
23:59:59
00:00:01
00:00:00
23:59:58
```

1.9 Ex: The `Ball` Class

A class called Ball, which models a bouncing ball, is designed as shown in the following class diagram. It contains its radius, x and y position. Each move-step advances the x and y by delta-x and delta-y, respectively. delta-x and delta-y could be positive or negative. The reflectHorizontal() and reflectVertical() methods could be used to bounce the ball off the walls. Write the Ball class. Study the test driver on how the ball bounces.


Ball
-x:float
-y:float
-radius:int
-xDelta:float
-yDelta:float
+Ball(x:float,y:float,radius:int xDelta:float,yDelta:float)
+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)]"

Below is a test driver:

```
public class TestMain {
   public static void main(String[] args) {
      // Test constructor and toString()
      Ball ball = new Ball(1.1f, 2.2f, 10, 3.3f, 4.4f);
      System.out.println(ball);  // toString()

      // Test Setters and Getters
      ball.setX(80.0f);
      ball.setY(35.0f);
      ball.setRadius(5);
      ball.setXDelta(4.0f);
      ball.setYDelta(6.0f);
      System.out.println(ball);  // toString()
      System.out.println("x is: " + ball.getX());
      System.out.println("y is: " + ball.getY());
      System.out.println("radius is: " + ball.getRadius());
      System.out.println("xDelta is: " + ball.getXDelta());
      System.out.println("yDelta is: " + ball.getYDelta());

      // Bounce the ball within the boundary
      float xMin = 0.0f;
      float xMax = 100.0f;
      float yMin = 0.0f;
      float yMax = 50.0f;
      for (int i = 0; i < 15; i++) {
         ball.move();
         System.out.println(ball);
         float xNew = ball.getX();
         float yNew = ball.getY();
         int radius = ball.getRadius();
         // Check boundary value to bounce back
```

```
            if ((xNew + radius) > xMax || (xNew - radius) < xMin) {
               ball.reflectHorizontal();
            }
            if ((yNew + radius) > yMax || (yNew - radius) < yMin) {
               ball.reflectVertical();
            }
         }
      }
   }
```

The expected output is:

```
Ball[(1.1,2.2),speed=(3.3,4.4)]
Ball[(80.0,35.0),speed=(4.0,6.0)]
x is: 80.0
y is: 35.0
radius is: 5
xDelta is: 4.0
yDelta is: 6.0
Ball[(84.0,41.0),speed=(4.0,6.0)]
Ball[(88.0,47.0),speed=(4.0,6.0)]
Ball[(92.0,41.0),speed=(4.0,-6.0)]
Ball[(96.0,35.0),speed=(4.0,-6.0)]
Ball[(92.0,29.0),speed=(-4.0,-6.0)]
Ball[(88.0,23.0),speed=(-4.0,-6.0)]
Ball[(84.0,17.0),speed=(-4.0,-6.0)]
Ball[(80.0,11.0),speed=(-4.0,-6.0)]
Ball[(76.0,5.0),speed=(-4.0,-6.0)]
Ball[(72.0,-1.0),speed=(-4.0,-6.0)]
Ball[(68.0,5.0),speed=(-4.0,6.0)]
Ball[(64.0,11.0),speed=(-4.0,6.0)]
Ball[(60.0,17.0),speed=(-4.0,6.0)]
Ball[(56.0,23.0),speed=(-4.0,6.0)]
Ball[(52.0,29.0),speed=(-4.0,6.0)]
```

Try: Modify the constructor to take in speed and direction (in polar coordinates) instead of delta-x and delta-y (in cartesian coordinates), which is more convenient for the users.

```
public Ball(float x, float y, int radius, int speed, int directionInDegree)
```