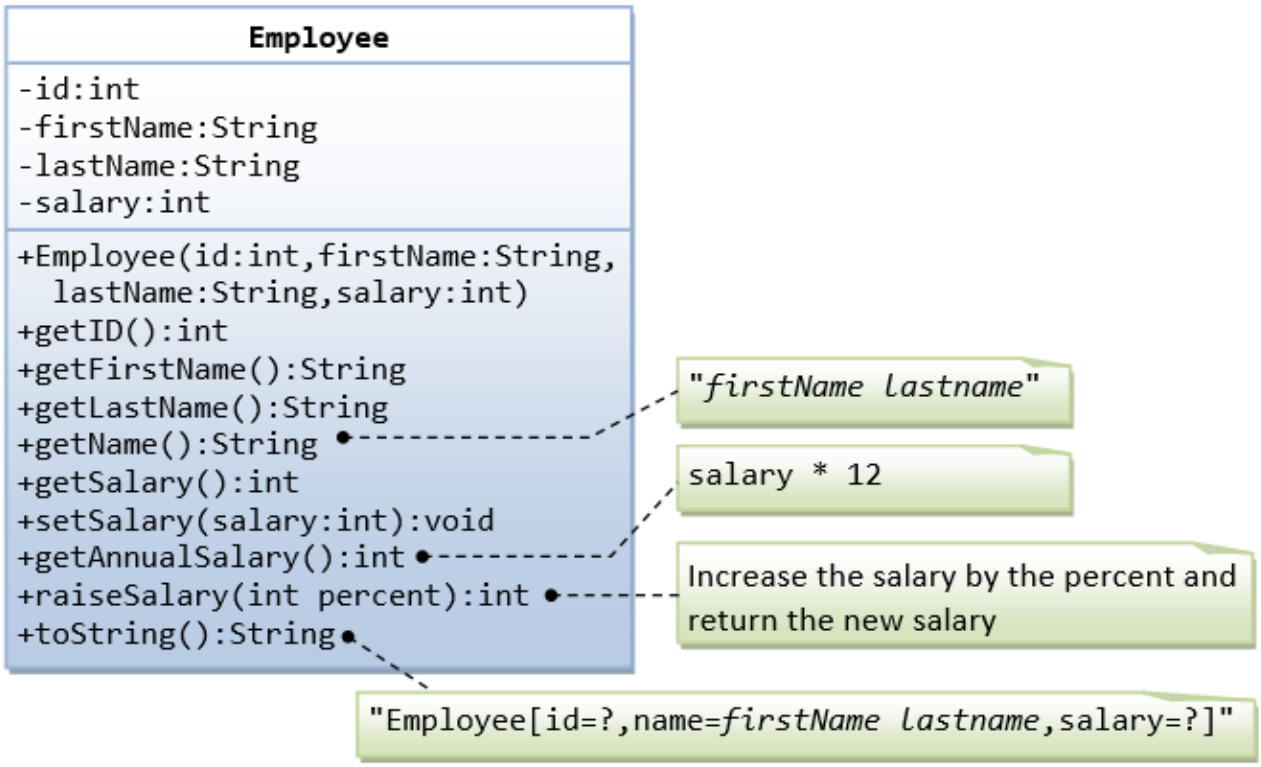
Ex: The `Employee` Class

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.



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());
```

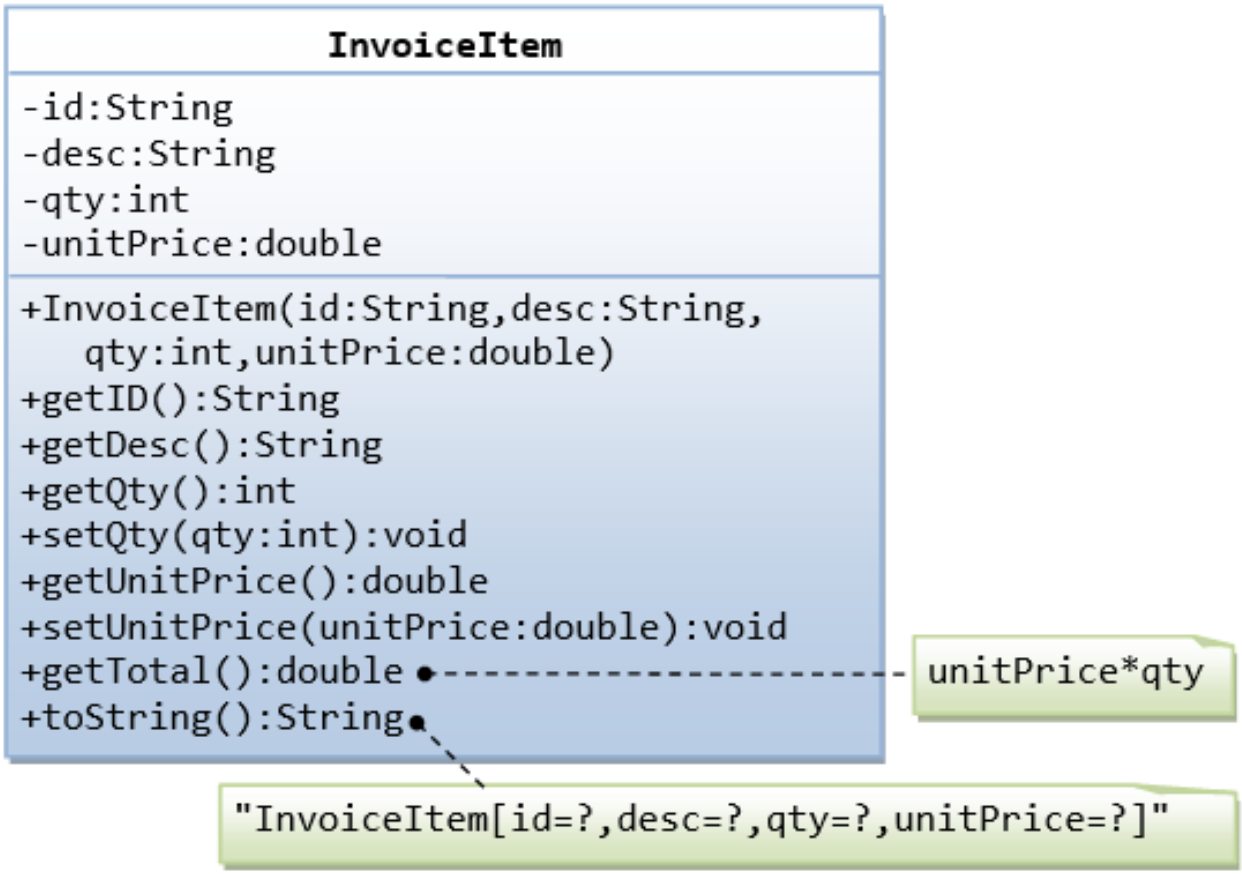
```
System.out.println("lastname is: " +
e1.getLastName());
System.out.println("salary is: " +
e1.getSalary());
System.out.println("name is: " +
e1.getName());
System.out.println("annual salary is: " +
e1.getAnnualSalary()); // Test method
     // Test raiseSalary()
System.out.println(e1.raiseSalary(10));
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();
  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());
```

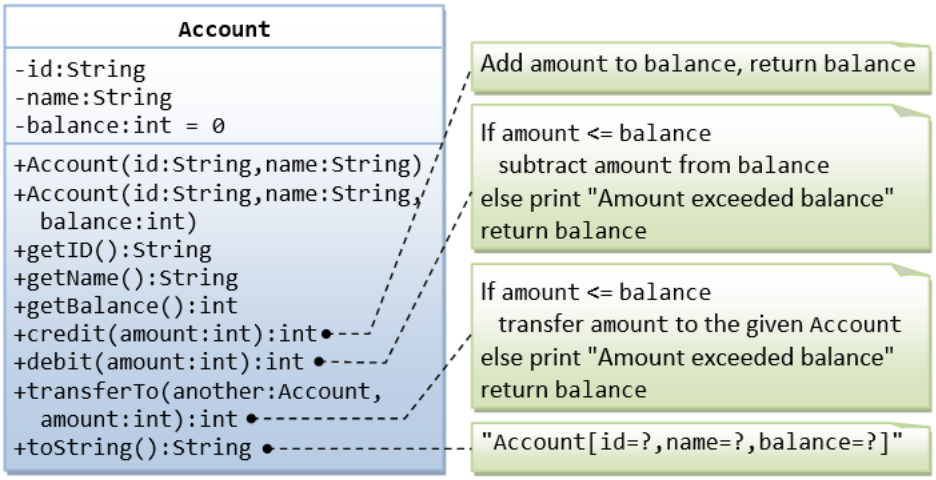
```
System.out.println("The total is: " +
inv1.getTotal());
   }
}
```

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.



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);
   }
}
```

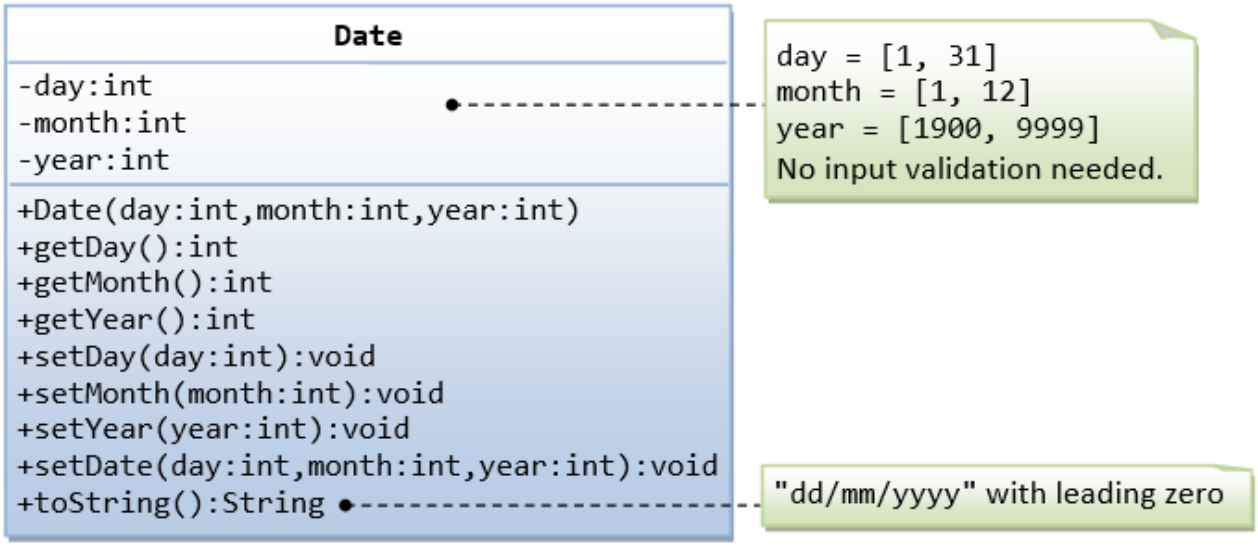
The expected output is:

```
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
```

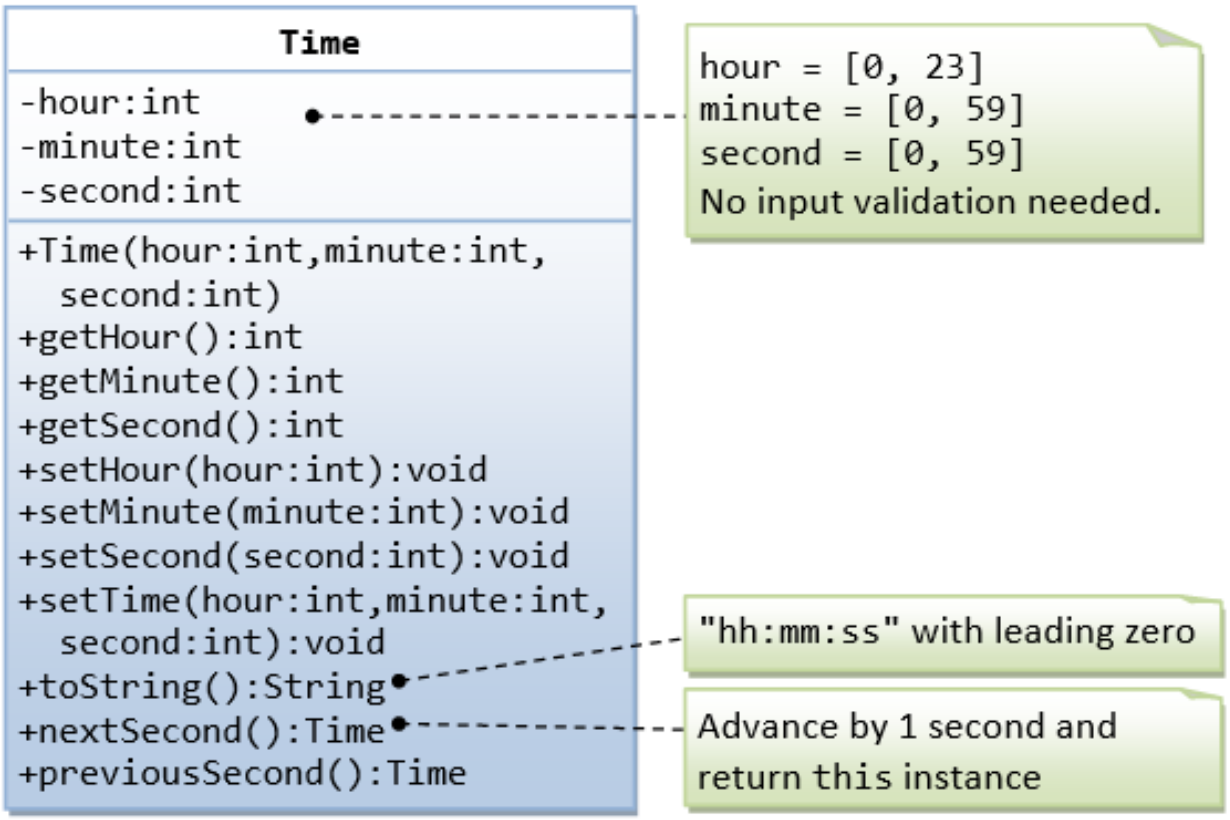
```
      d1.setMonth(12);
      d1.setDay(9);
      d1.setYear(2099);
      System.out.println(d1);  // toString()
            System.out.println("Month:  "  +
d1.getMonth());
             System.out.println("Day:  "  +
d1.getDay());
            System.out.println("Year:  "  +
d1.getYear());

      // 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.



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());
```

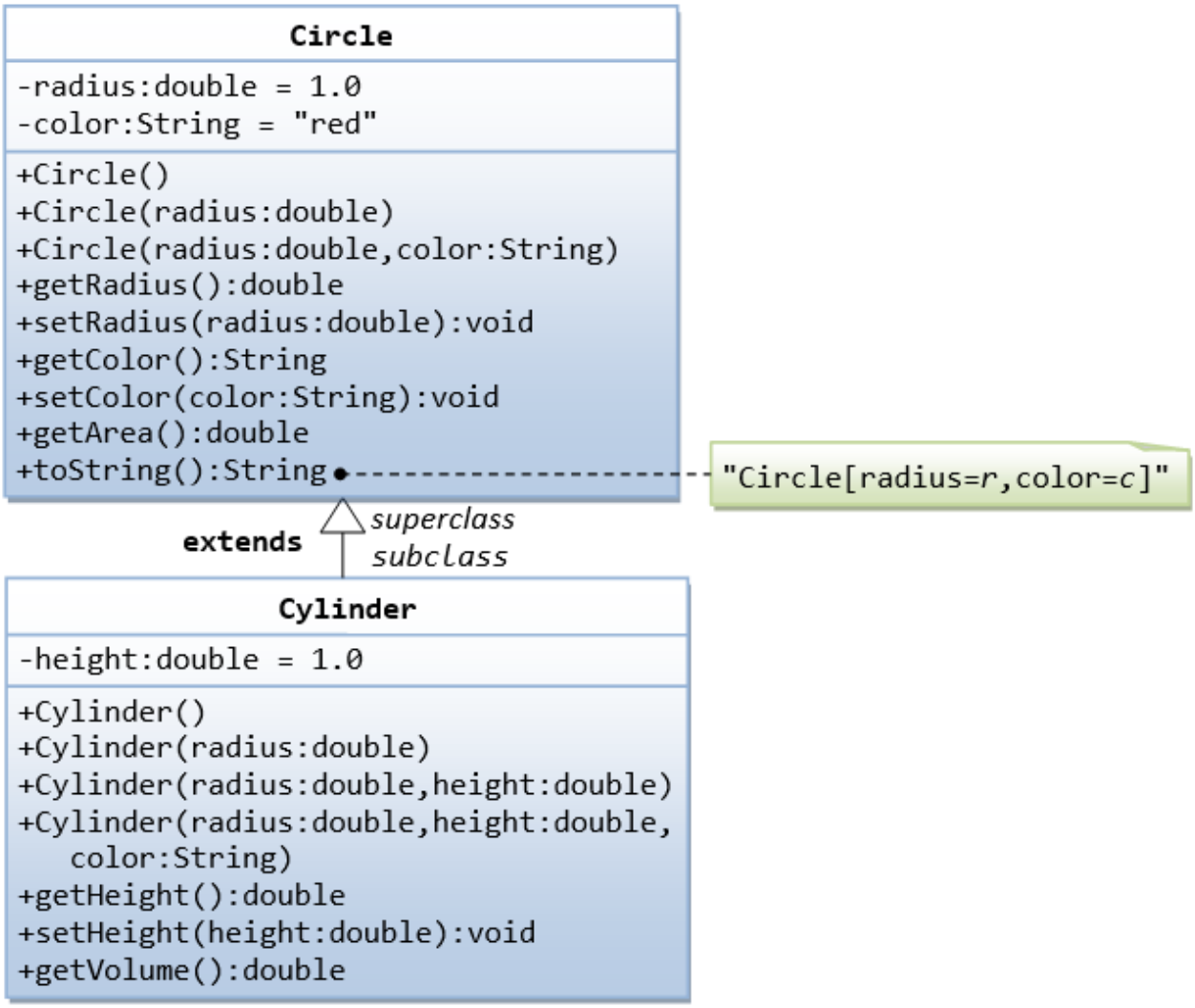
```
 // Test setTime()
 t1.setTime(23, 59, 58);
System.out.println(t1);  // toString()

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

System.out.println(t1.nextSecond().nextSecon
d());

// Test previousSecond()

System.out.println(t1.previousSecond());

System.out.println(t1.previousSecond().previ
ousSecond());
   }
}
```

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
```

An Introduction to OOP Inheritance by Example - 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 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.

```
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:

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

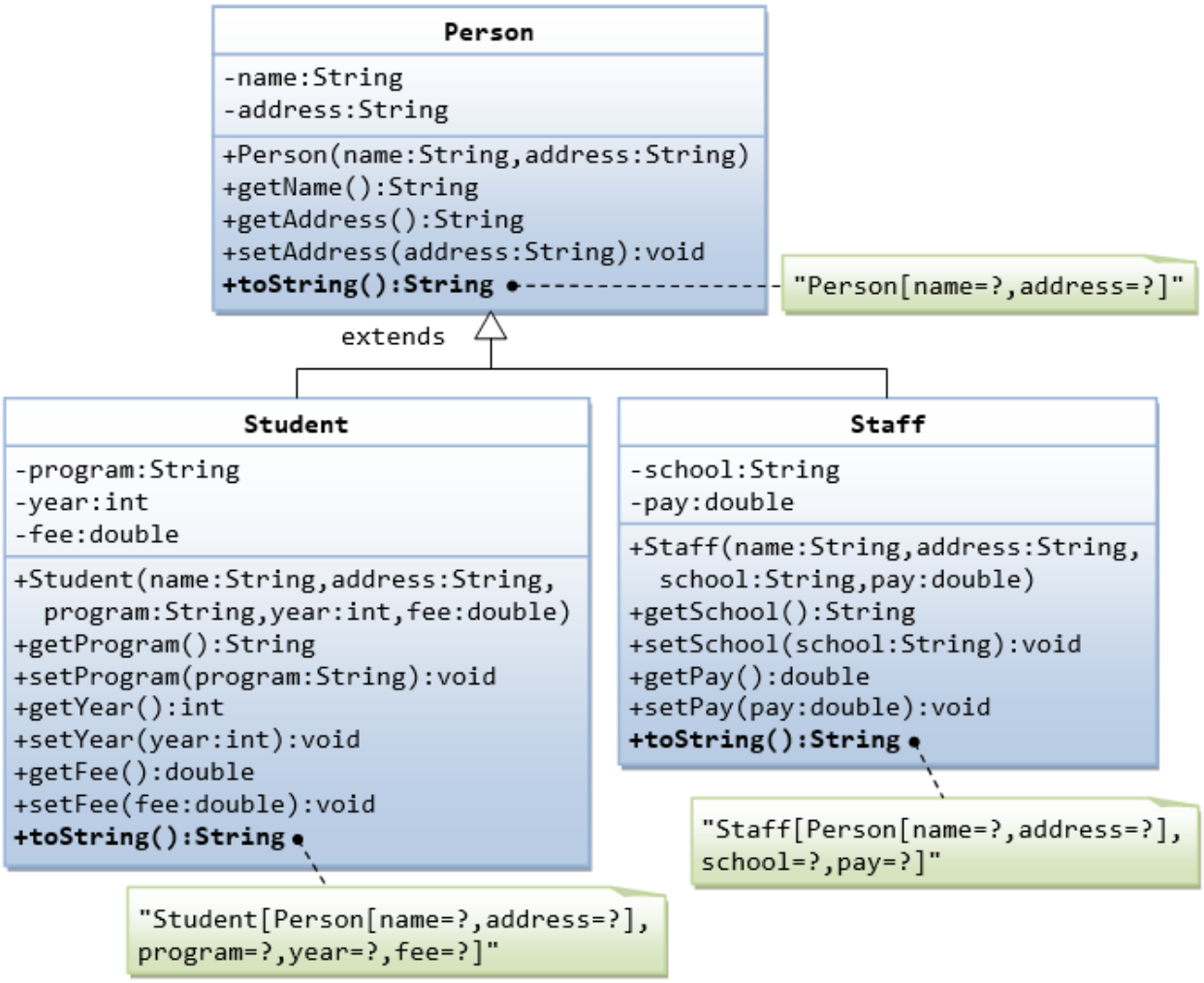
```
}
```

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 Ex: Superclass `Person` and its subclasses

Write the classes as shown in the following class diagram. Mark all the overridden methods with annotation `@Override`.



Using inheritance, one class can acquire the properties of others. Consider the following Animal class:

```
class Animal{
    void walk(){
        System.out.println("I am walking");
    }
}
```

This class has only one method, walk. Next, we want to create a Bird class that also has a fly method. We do this using extends keyword:

```
class Bird extends Animal {
    void fly() {
        System.out.println("I am flying");
```

```
    }
}
```
Finally, we can create a Bird object that can both fly and walk.
```
public class Solution{
   public static void main(String[] args){

      Bird bird = new Bird();
      bird.walk();
      bird.fly();
   }
}
```
The above code will print:
I am walking
I am flying

This means that a Bird object has all the properties that an Animal object has, as well as some additional unique properties.
The code above is provided for you in your editor. You must add a sing method to the Bird class, then modify the main method accordingly so that the code prints the following lines:
I am walking
I am flying
I am singing

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;
```

```
class Animal{
        void walk()
        {
                System.out.println("I am walking");
        }
}
class Bird extends Animal
{
        void fly()
        {
                System.out.println("I am flying");
        }
     void walk(){
         System.out.println("I am walking");
     }
      void sing(){
         System.out.println("I am singing");
     }
}

public class Solution{

   public static void main(String args[]){

          Bird bird = new Bird();
          bird.walk();
          bird.fly();
        bird.sing();

   }
}
```

Write the following code in your editor below:

1. A class named Arithmetic with a method named add that takes integers as parameters and returns an integer denoting their sum.
2. A class named Adder that inherits from a superclass named Arithmetic.

Your classes should not be be .

Input Format

You are not responsible for reading any input from stdin; a locked code stub will test your submission by calling the add method on an Adder object and passing it integer parameters.

Output Format

You are not responsible for printing anything to stdout. Your add method must return the sum of its parameters.

Sample Output

The main method in the Solution class above should print the following:

My superclass is: Arithmetic
42 13 20

```
import java.io.*;
import java.util.*;

public class Solution {

    public static void main(String[] args) {
        Adder a = new Adder();
        System.out.println("My superclass is: Arithmetic");
```

```
        a.call();
    }
}

class Arithmetic{
    int add(int a, int b){
        return  a+b;
    }
}
class Adder extends Arithmetic{
    void call(){
        System.out.print(add(40,2)+" ");
        System.out.print(add(10,3)+" ");
        System.out.print(add(10,10)+" ");
    }
}
```

A Java abstract class is a class that can't be instantiated. That means you cannot create new instances of an abstract class. It works as a base for subclasses. You should learn about Java Inheritance before attempting this challenge. Following is an example of abstract class:

```
abstract class Book{
    String title;
    abstract void setTitle(String s);
    String getTitle(){
        return title;
    }
}
```

If you try to create an instance of this class like the following line you will get an error:

```
Book new_novel=new Book();
```

You have to create another class that extends the abstract class. Then you can create an instance of the new class. Notice that setTitle method is abstract too and has no body. That means you must implement the body of that method in the child class.
In the editor, we have provided the abstract Book class and a Main class. In the Main class, we created an instance of a class called MyBook. Your task is to write just the MyBook class.
Your class mustn't be public.

Sample Input

A tale of two cities

Sample Output

The title is: A tale of two cities

```
abstract class Book{
String title;
abstract void setTitle(String s);
String getTitle(){
return title;
} }
class MyBook extends Book{
void setTitle(String s){
title=s;
} }
public class Solution {
public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    String title = sc.nextLine();

    MyBook new_novel = new MyBook();
    new_novel.setTitle(title);
```

```
    System.out.println("The title is: "+new_novel.getTitle());
    sc.close();
}
}
```

A Java interface can only contain method signatures and fields. The interface can be used to achieve polymorphism. In this problem, you will practice your knowledge on interfaces.
You are given an interface AdvancedArithmetic which contains a method signature int divisor_sum(int n). You need to write a class called MyCalculator which implements the interface.
divisorSum function just takes an integer as input and return the sum of all its divisors. For example divisors of 6 are 1, 2, 3 and 6, so divisor_sum should return 12. The value of n will be at most 1000.
Read the partially completed code in the editor and complete it. You just need to write the MyCalculator class only. Your class shouldn't be public.

Sample Input

6

Sample Output

I implemented: AdvancedArithmetic
12

Explanation

Divisors of 6 are 1,2,3 and 6. 1+2+3+6=12.

```
interface AdvancedArithmetic{
int divisor_sum(int);
}

class MyCalculator implements AdvancedArithmetic{
```

```
    @Override
 public int divisor_sum(int n){
      int sum = 0;
      for(int i=1; i<=n; i++){
          if(n%i==0){
            sum = sum+i;
          }

      }
      return sum;
  }
public static void main(String[] args) {
System.out.println("I implemented: AdvancedArithmetic");
Scanner s=new Scanner(System.in);
int a=s.nextInt();
MyCalculator mc=new MyCalculator();
mc.divisor_sum(a);
}
}
```

When a subclass inherits from a superclass, it also inherits its methods; however, it can also override the superclass methods (as well as declare and implement new ones). Consider the following Sports class:

```
class Sports{
    String getName(){
        return "Generic Sports";
    }
    void getNumberOfTeamMembers(){
        System.out.println( "Each team has n players in " +
getName() );
    }
}
```

Next, we create a Soccer class that inherits from the Sports class. We can override the getName method and return a different, subclass-specific string:

```
class Soccer extends Sports{
    @Override
    String getName(){
        return "Soccer Class";
    }
}
```

Note: When overriding a method, you should precede it with the @Override annotation. The parameter(s) and return type of an overridden method must be exactly the same as those of the method inherited from the supertype.

Task

Complete the code in your editor by writing an overridden getNumberOfTeamMembers method that prints the same statement as the superclass' getNumberOfTeamMembers method, except that it replaces with (the number of players on a Soccer team).

Output Format

When executed, your completed code should print the following:

```
Generic Sports
Each team has n players in Generic Sports
Soccer Class
Each team has 11 players in Soccer Class
```

```
import java.util.*;

class Sports{

        String getName(){
```

```
            return "Generic Sports";
        }

        void getNumberOfTeamMembers(){
            System.out.println( "Each team has n players in " +
getName() );
        }
    }

class Soccer extends Sports{
        @Override
        String getName(){
            return "Soccer Class";
        }

        void getNumberOfTeamMembers(){
            System.out.println( "Each team has 11 players in "
+ getName() );
        }

    }

public class javaOverridemethodwork {

        public static void main(String []args){

           Sports mySportsclass = new Sports();
            Soccer mySoccerclass = new Soccer();

            System.out.println(mySportsclass.getName());
            mySportsclass.getNumberOfTeamMembers();

            System.out.println(mySoccerclass.getName());
```

```
            mySoccerclass.getNumberOfTeamMembers();
        }
    }
```

When a method in a subclass overrides a method in superclass, it is still possible to call the overridden method using **super** keyword. If you write super.func() to call the function func(), it will call the method that was defined in the superclass.
You are given a partially completed code in the editor.
Modify the code so that the code prints the following text:
Hello I am a motorcycle, I am a cycle with an engine.
My ancestor is a cycle who is a vehicle with pedals.

```
import java.util.*;
import java.io.*;

class BiCycle{

    String define_me(){
            return "a vehicle with pedals.";
        }
    }

    class MotorCycle extends BiCycle{
        String define_me(){
            return "a cycle with an engine.";
        }

        MotorCycle(){
            System.out.println("Hello I am a motorcycle, I
am "+ define_me());
```

```
                String temp = super.define_me();


                System.out.println("My ancestor is a cycle who
is "+ temp );
            }


    }

    public class javaMethodoverriding2superWork {
        public static void main(String []args){
            MotorCycle M = new MotorCycle();

        }
    }
```

The Java instanceof operator is used to test if the object or instance is an instanceof the specified type.
In this problem we have given you three classes in the editor:

- Student class
- Rockstar class
- Hacker class

In the main method, we populated an ArrayList with several instances of these classes. count method calculates how many instances of each type is present in the ArrayList. The code prints three integers, the number of instance of Student class, the number of instance of Rockstar class, the number of instance of Hacker class.

But some lines of the code are missing, and you have to fix it by modifying only lines! Don't add, delete or modify any extra line.
To restore the original code in the editor, click on the top left icon in the editor and create a new buffer.

Sample Input

```
5
Student
Student
Rockstar
Student
Hacker
```

Sample Output

```
3 1 1
```

```
import java.util.*;


class Student{}
class Rockstar{}
class Hacker{}


public class javaInstanceofKeywordwork {


   static String count(ArrayList mylist){
      int a = 0, b = 0, c = 0;

      for(int i = 0; i < mylist.size(); i++){

         Object element = mylist.get(i);
```

```
        if(element instanceof Student )
            a++;

        if(element instanceof Rockstar)
            b++;

        if(element instanceof Hacker)
            c++;
      }
      String ret = Integer.toString(a) + " " + Integer.toString(b)
+ " " +  Integer.toString(c);
      return ret;
    }

    public static void main(String []args){
      ArrayList mylist = new ArrayList();

      Scanner scan = new Scanner(System.in);
      int t = scan.nextInt();

      for(int i= 0; i<t; i++){

        String s = scan.next();

        if(s.equals("Student"))mylist.add(new Student());
         if(s.equals("Rockstar"))mylist.add(new Rockstar());
         if(s.equals("Hacker"))mylist.add(new Hacker());
      }

      System.out.println(count(mylist));
    }
}
```

Java Iterator class can help you to iterate through every element in a collection. Here is a simple example:

```
import java.util.*;
public class Example{

    public static void main(String []args){
        ArrayList mylist = new ArrayList();
        mylist.add("Hello");
        mylist.add("Java");
        mylist.add("4");
        Iterator it = mylist.iterator();
        while(it.hasNext()){
            Object element = it.next();
            System.out.println((String)element);
        }
    }
}
```

In this problem you need to complete a method func. The method takes an ArrayList as input. In that ArrayList there is one or more integer numbers, then there is a special string "###", after that there are one or more other strings. A sample ArrayList may look like this:

```
element[0]=>42
element[1]=>10
element[2]=>"###"
element[3]=>"Hello"
element[4]=>"Java"
```

You have to modify the func method by editing at most 2 lines so that the code only prints the elements after the special string "###". For the sample above the output will be:

```
Hello
Java
```

Note: The stdin doesn't contain the string "###", it is added in the main method.
To restore the original code in the editor, click the top left icon on the editor and create a new buffer.

```
import java.util.*;
public class Main{

  static ArrayList func(ArrayList mylist){
    Iterator it=mylist.iterator();
    while(it.hasNext()){
      Object element =it.next();
      if(element instanceof Integer ||
"###".equals((String)element)){
          //Hints: use instanceof operator
          it.remove();
      }
    }
    return mylist;
  }
  @SuppressWarnings({ "unchecked" })
  public static void main(String[] args){
    ArrayList mylist = new ArrayList();
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int m = sc.nextInt();
    for(int i = 0;i<n;i++){
      mylist.add(sc.nextInt());
    }

    mylist.add("###");
    for(int i=0;i<m;i++){
      mylist.add(sc.next());
```

```
        }

        ArrayList newList=func(mylist);
        Iterator it= newList.iterator();
        while(it.hasNext()){
           Object element = it.next();
           System.out.println((String)element);
        }
    }
}
```