CS–313
Object Oriented Analysis and Design (OOA&D)
Lab # 4

JavaDocs, Inheritance, Polymorphism, Abstract Classes

Objectives
Understanding the concepts of
- Using JavaDocs
- Inheritance
- Abstract Classes
- Polymorphism

4.1. Using JavaDocs

4.1.1. Introduction
Have you ever had an experience of reading your code months after you last worked on it? You probably had problems remembering exactly what the code does. We can actually do two things to avoid such a situation. One is to strive to make your program simple and readable. The other is to write good documentation.

Javadoc is a convenient, standard way to document your Java code. Javadoc is actually a special format of comments. There are some utilities that read the comments, and then generate HTML document based on the comments. HTML files give us the convenience of hyperlinks from one document to another. Most class libraries, both commercial and open source, provide Javadoc documents.

In this tutorial, we are going to learn several ways to write Javadoc comments with the assistance of Eclipse, and generate HTML files with the Javadoc Export WIzard. There are several tags for Javadoc comments. However, you don't have to try to memorize them all.

Note: Most frequently used tags can be generated by the code template, and you can find others with Content Assist by pressing Ctrl-SPACE.

4.1.2. Types of Javadoc
There are two kinds of Javadoc comments: class-level comments, and member-level comments. Class-level comments provide the description of the classes, and member-level comments describe the purposes of the members. Both types of comments start with /** and end with */. For example, this is a Javadoc comment:

```
/** This is a Javadoc comment */
```

4.1.2.1. Class-level Comments

Class-level comments provide a description of the class, and they are placed right above the code that declares the class. Class-level comments generally contain author tags, and a description of the class. An example class-level comment is below:

```
/**
 * @author OOAD_Instructor
 *
 * The Inventory class contains the amounts of all the
 * inventory in the CoffeeMaker system.  The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 */
public class Inventory {

   //Inventory code here

}
```

4.1.2.2. Member-level Comments

Member-level comments describe the fields, methods, and constructors. Method and constructor comments may contain tags that describe the parameters of the method. Method comments may also contain return tags. An example of these member-level comments are below:

```
/**
 * @author OOAD_Instructor
 *
 * The Inventory class contains the amounts of all the
 * inventory in the CoffeeMaker system. The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 */
public class Inventory {

   /**
    * Inventory for coffee
    */
   private int amtCoffee;

   /**
    * Default constructor for Inventory
    * Sets all ingredients to 15 units
    */
   public Inventory() {
      this.coffee = 15;
   }

   /**
    * Returns the units of coffee in the Inventory
    *
    * @return int
    */
   public int getAmtCoffee() {
      return coffee;
   }

   /**
    * Sets the units of coffee in the Inventory
    *
    * @param int new units coffee
    */
   public void setAmtCoffee(int newCoffee) {
      this.coffee = newCoffee;
   }
}
```

4.1.3. Tags

Tags are keywords recognized by Javadoc which define the type of information that follows. Tags come after the description (separated by a new line). Here are some common pre-defined tags:

- ***@author [author name]*** - identifies author(s) of a class or interface.
- ***@version [version]*** - version info of a class or interface.
- ***@param [argument name] [argument description]*** - describes an argument of method or constructor.
- ***@return [description of return]*** - describes data returned by method (unnecessary for constructors and void methods).
- ***@exception [exception thrown] [exception description]*** - describes exception thrown by method.
- ***@throws [exception thrown] [exception description]*** - same as ***@exception***.

4.1.4. Javadoc Templates in Eclipse

Eclipse generates Javadoc everytime you create a class, method, or field, using templates built into the IDE. These templates can be modified to create Javadoc in the format that you want. Or you can turn off the Javadoc generation property.

The Javadoc templates are found by selecting **Window > Preferences**. Under the tree on the left of the Preferences dialog, open up **Java > Code Style > Code Templates**.

There is a check box at the bottom of the Code Templates page that says **Automatically add comments for new methods and types.** If you uncheck this box, Javadoc comments will not be generated.

4.1.5. Generate HTML Document

- Right click on the project and select **Export...**. Select **Javadoc**, and click **Next**.
- Make sure the **Javadoc command** refers to the Javadoc command line tool. This tool is provided with JDK. The name is usually *javadoc.exe*. In the lab, you can find *javadoc.exe* in *C:\jdk1.6.1\bin\* (that is, the Javadoc command is *C:\jdk1.6.1\bin\ javadoc.exe* .)

- Select the types that you want to create Javadoc for.
- Make sure that the **Use Standard Doclet** radio button is chosen, and **Browse** for a destination folder.
- Click **Next** for more options if you wish; otherwise click **Finish.**

You can find the documentation of the Java API at http://docs.oracle.com/javase/6/docs/api/

4.2. Inheritance

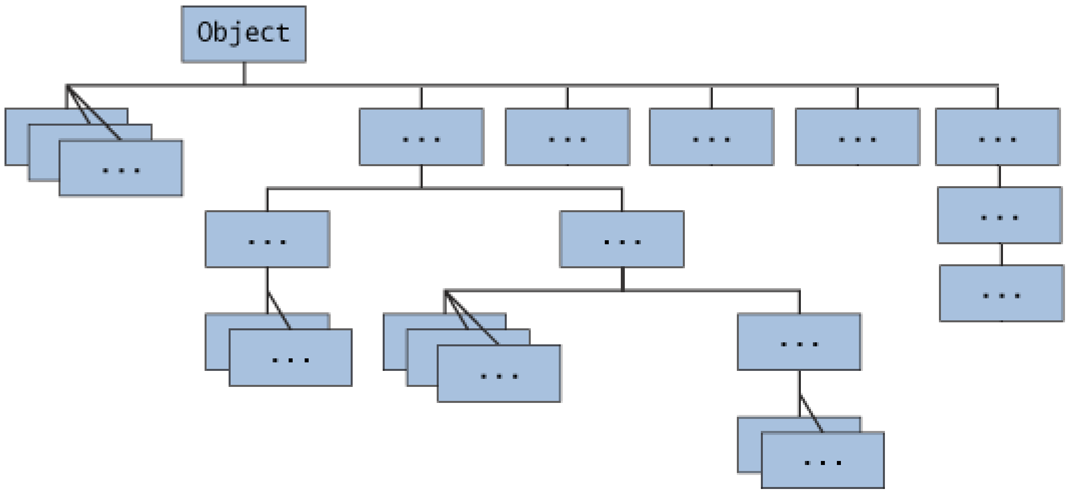
The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself. A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

The *Object* class sits at the top of the class hierarchy tree in the Java development environment. Every class in the Java system is a descendent (direct or indirect) of the `Object` class. The `Object` class defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, and to return the object's class. Excepting `Object`, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to `Object`. A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

4.2.1. The Java Platform Class Hierarchy

The *Object* class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, `Object` is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

4.2.2. What You Can Do in a Subclass

A subclass inherits all members of its parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

The following sections in this lesson will expand on these topics.

4.2.3. Inheritance and Access

When inheritance is used to create a new (derived) class from an existing (base) class, everything in the base class is also in the derived class

- it may not be accessible, however - the access in the derived class depends on the access in the base class:

base class access	accessibility in derived class
public	public
protected	protected
private	inaccessible

Note that private elements become inaccessible to the derived class - this does not mean that they disappear, or that that there is no way to affect their values, just that they can't be referenced by name in code within the derived class

4.2.4. Inheritance and Constructors

Since a derived class object contains the elements of a base class object, it is reasonable to want to use the base class constructor as part of the process of constructing a derived class object. But constructors are "not inherited". Within a derived class constructor, however, you can use super(*parameterList*) to call a base class constructor

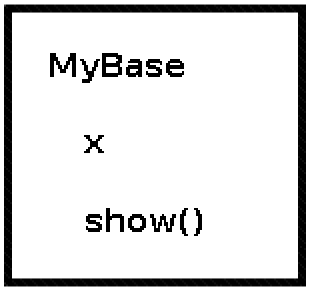
- it must be done as the first line of a constructor
- if you do not explicitly invoke a form of super-constructor, then super() (the form that takes no parameters) will run
- and for the superclass, its constructor will either explicitly or implicitly run a constructor for its superclass
- so, when an instance is created, *one constructor will run at every level of the inheritance chain*, all the way from Object up to the current class

4.2.5. Inheritance Example - A Derived Class

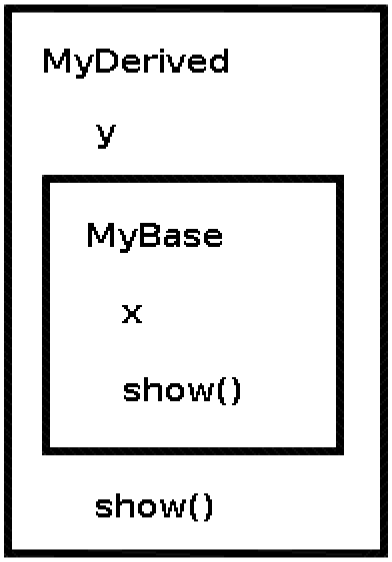
When a derived class is created, an object of the new class will in effect contain a complete object of the base class within it

The following maps out the relation between the derived class and the base class.

```
public class MyBase {
 public int x;
 public void show() {
  System.out.println("x =" + x);
}
```



```
class MyDerived extends MyBase {
 public int y;
 public void show() {
  System.out.println("x = " + x);
  System.out.println("y = " + y);
 }
}
```



Since everything in MyBase is public, code in the MyDerived class has free access to the x value from the MyBase object inside it, as well as y and show() from itself.

- the show() method from MyBase is also available, but only within MyDerived class code (but some work is required to get it, since it is hidden by the show() method added with MyDerived) - code in other classes cannot invoke the MyBase version of show() at all

4.2.6. Example

```
class MyBase {
      private int x;
      public MyBase(int x) {
            this.x = x;
      }
      public int getX() {
            return x;
      }
      public void show() {
            System.out.println("x=" + x);
      }
}

class MyDerived extends MyBase {
      private int y;
       public MyDerived(int x) {
             super(x);
       }
      public MyDerived(int x, int y) {
            super(x);
            this.y = y;
      }
      public int getY() {
```

```
                return y;
        }
        public void show() {
                System.out.println("x = " + getX());
                System.out.println("y = " + y);
        }
}
public class Inheritance1 {
        public static void main(String[] args) {
                MyBase b = new MyBase(2);
                b.show();
                MyDerived d = new MyDerived(3, 4);
                d.show();
        }
}
```

4.2.7. Method Overriding

As we saw before, you can create a method in the derived class with the same name as a base class method

- the new method *overrides* (and hides) the original method
- you can still call the base class method from within the derived class if necessary, by adding the super keyword and a dot in front of the method name
- the base class version of the method is not available to outside code
- you can view the super term as providing a reference to the base class object buried inside the derived class
- you cannot do super.super. to back up two levels
- you cannot change the return type when overriding a method, since this would make polymorphism impossible

4.2.8. The Order of Construction under Inheritance

Note that when you construct an object, the default base class constructor is called implicitly, before the body of the derived class constructor is executed. So, objects are constructed top-down under inheritance. Since every object inherits from the original Object class, the Object() constructor is always called implicitly. However, you can call a superclass constructor explicitly using the built-in super keyword, as long as it is the first statement in a constructor.

For example, most Java exception objects inherit from the java.lang.Exception class. If you wrote your own exception class, say SomeException, you might write it as follows:

```
public class SomeException extends Exception {
        public SomeException() {
                super(); // calls Exception(), which calls Object()
        }
        public SomeException(String s) {
                super(s); // calls Exception(String),
        }
        public SomeException (int error_code) {
                this("error”); // calls super(String)
                System.err.println(error_code);
        }
}
```

4.3. Abstract Classes and Methods

Java provides a special type of class called an abstract class which helps us to organize our classes based on common methods. An abstract class lets you put the common method names in one abstract class without having to write the actual implementation code. An abstract class can be extended into sub-classes, these sub-classes usually provide implementations for all of the abstract methods.

Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. You can require that certain methods be overridden by subclasses by specifying the abstract type modifier, they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

So an abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.

```
public abstract class GraphicObject {
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

4.3.1. An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—

for example, resize or draw. All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, GraphicObject, as shown in the following figure.

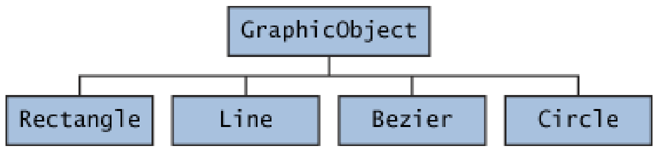


Figure 1: Classes Rectangle, Line, Bezier, and Circle inherit from GraphicObject

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject {
   int x, y;
   ...
   void moveTo(int newX, int newY) {
      ...
   }
   abstract void draw();
   abstract void resize();
}
```

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

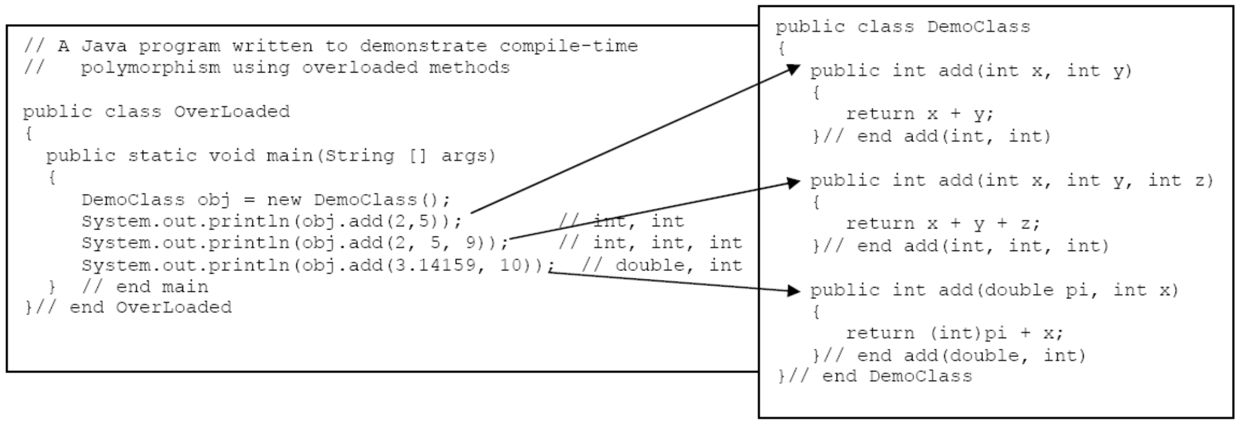
```
class Circle extends GraphicObject {
   void draw() {
      ...
   }
   void resize() {
      ...
   }
}
class Rectangle extends GraphicObject {
   void draw() {
      ...
   }
   void resize() {
      ...
   }
}
```

4.4. Polymorphism

Generally, polymorphism refers to the ability to appear in many forms. Polymorphism in a Java program is the ability of a reference variable to change behavior according to what object instance it is holding. This allows multiple objects of different subclasses to be treated as objects of a single super class, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to

For example, given a base class shape, polymorphism enables the programmer to define different area methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the area method to it will return the correct results.

Polymorphism means "a method the same as another in spelling but with different behavior." The computer differentiates between (or among) methods depending on either the method signature (after compile) or the object reference (at run time).
In the example below polymorphism is demonstrated by the use of multiple add methods. The computer differentiates among them by the method signatures (the list of parameters: their number, their types, and the order of the types.)



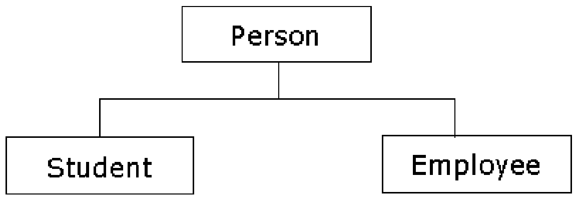
```
// A Java program written to demonstrate compile-time
//   polymorphism using overloaded methods

public class OverLoaded
{
  public static void main(String [] args)
  {
     DemoClass obj = new DemoClass();
     System.out.println(obj.add(2,5));          // int, int
     System.out.println(obj.add(2, 5, 9));      // int, int, int
     System.out.println(obj.add(3.14159, 10));  // double, int
  }  // end main
}// end OverLoaded
```

```
public class DemoClass
{
   public int add(int x, int y)
   {
      return x + y;
   }// end add(int, int)

   public int add(int x, int y, int z)
   {
      return x + y + z;
   }// end add(int, int, int)

   public int add(double pi, int x)
   {
      return (int)pi + x;
   }// end add(double, int)
}// end DemoClass
```

This form of polymorphism is called **early-binding** (or **compile-time**) **polymorphism** because the computer knows after the compile to the byte code which of the add methods it will execute. That is, after the compile process when the code is now in byte-code form, the computer will "know" which of the add methods it will execute. If there are two actual *int* parameters the computer will know to execute the add method with two formal *int* parameters, and so on. Methods whose headings differ in the number and type of formal parameters are said to be *overloaded* methods. The parameter list that differentiates one method from another is said to be the method signature list.

There is another form of polymorphism called **late-binding** (or **run-time**) **polymorphism** because the computer does not know at compile time which of the methods are to be executed. It will not know that until "run time." Run-time polymorphism is achieved through what are called *overridden* methods (while compile-time polymorphism is achieved with overloaded methods). Sometimes run-time polymorphism is referred to as **dynamic binding**.

Example:
Given the parent class **Person** and the child class **Student**, we add another subclass of **Person** which is **Employee**. Below is the class hierarchy.



In Java, we can create a reference that is of type super class, Person, to an object of its subclass, Student.

```
public static main( String[] args ) {
        Student studentObject = new Student();
        Employee employeeObject = new Employee();

        Person ref = studentObject;     // Person reference points to a Student object

        // Calling getName() of the Student object instance
        String name = ref.getName();
}
```

Now suppose we have a getName method in our super class Person, and we override this method in both Student and Employee subclasses

```
public class Student {
        public String getName(){
                System.out.println("Student Name:" + name);
                return name;
        }
}
public class Employee {
        public String getName(){
                System.out.println("Employee Name:" + name);
                return name;
        }
}
```

Going back to our main method, when we try to call the *getName* method of the reference Person *ref*, the *getName* method of the *Student* object will be called. Now, if we assign *ref* to an *Employee* object, the *getName* method of *Employee* will be called.

```
public static main( String[] args ) {

        Student studentObject = new Student();
        Employee employeeObject = new Employee();

        Person ref = studentObject;     //Person ref. points to a Student object
        // getName() method of Student class is called
        String temp= ref.getName();
        System.out.println( temp );

        ref = employeeObject;           //Person ref. points to an Employee object

        //getName() method of Employee class is called
        String temp = ref.getName();
        System.out.println( temp );
    }
```

4.4.1. Benefits of polymorphism

1) Simplicity

– If you need to write code that deals with a family of types, the code can ignore type-specific details and just interact with the base type of the family
– Even though the code thinks it is using an object of the base class, the object's class could actually be the base class or any one of its subclasses
– This makes your code easier for you to write and easier for others to understand

2) Extensibility

– Other subclasses could be added later to the family of types, and objects of those new subclasses would also work with the existing code

Class Tasks

Task 1:

Imagine a publishing company that markets both books and audiocassette versions of its works. Create a class ***publication*** that stores the ***title*** (a string) and ***price*** (type float) of a publication. From this class, derive two classes: ***book***, which adds a ***page count*** (type int); and ***tape*** which adds a ***playing time in minutes*** (type float). Each of these three classes should have a ***getdata()*** function to get its data from the user at the keyboard, and a ***putdata()*** function to display its data.
Write a main() program to test the book and tape classes by creating instances of them, asking to fill in data with getdata() and displaying the data with putdata().

Task 2:

Generate Javadoc for the above classes. Use all the recognized tags given in the tutorial.

Home Assignment

Task 1:

Using an inheritance hierarchy, design a Java program to model 3-dimensional shapes (square pyramid, sphere, rectangular prism, cube, cylinder, circular cone). Make a top level shape class with methods for getting the area and the volume. Next, build classes and subclasses for the above 3dimensional shapes with functionality to calculate area and volume. Make sure that you place common behavior in superclasses whenever possible. Also, use abstract classes as appropriate. Add methods to subclasses to represent unique behavior particular to each 3-dimensional shape. Write the definitions of these classes and a test program showing them in use.

Task 2:

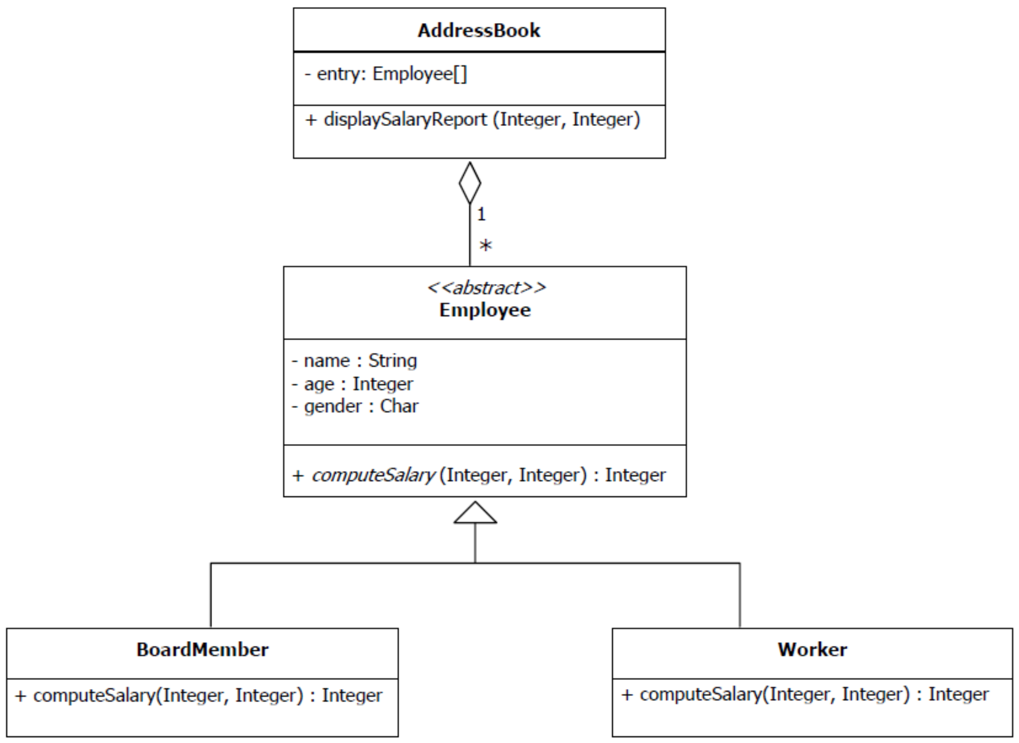

AddressBook
- entry: Employee[]
+ displaySalaryReport (Integer, Integer)
1
*
<<abstract>>
Employee
- name : String
- age : Integer
- gender : Char
+ computeSalary (Integer, Integer) : Integer
BoardMember
+ computeSalary(Integer, Integer) : Integer
Worker
+ computeSalary(Integer, Integer) : Integer

Figure 2: Class Diagram. Note not all of the attributes and methods are shown here. You may add any number of attributes and methods according to your requirements.

Your task is to implement an address book that contains information about the employees of a company. There are two types of employees in the company i.e. Board Members and Workers. Implement your program according to the UML diagram given above. *computeSalary(….)* is an abstract function that works differently for board managers and workers. Use the following formulas for calculating the salaries.

For the BoardMember class : salary = baseSalary + days*35 + age*100
For the Worker class : salary = baseSalary + days*45 + age*50

where salary is the Integer value to be returned by the method; baseSalary is the first input parameter, an Integer value indicating the base salary amount for the employees; days is the second parameter, an Integer value indicating the number of days the salary employees has been working without a payment; and age is the age of the employee whose salary is being calculated.

The displaySalaryReport(baseSalary: Integer, days: Integer) method should call the computeSalary method of each one of the employee objects the AddressBook object contains and display the name of the employee and its corresponding salary, a list similar to:

John Smith	$ 3500
Jennifer Pole	$ 2980
Lucia Anaya	$ 4560