**Inheritance and Interfaces**

By the end of this assignment, you should understand the following terms:

* Inheritance
* Aggregation
* Superclass
* Subclass
* "extends" keyword
* "implements" keyword
* "protected" keyword
* "interface" keyword
* Polymorphism

As you have seen Java is an object-oriented programming language. Emphasis is placed on modularity through classes and their instantiation as objects. One of the most powerful concepts in object-oriented programming is inheritance.

The strength of object-oriented programming is that is helps the programmer to organize the programs they write in an intuitive way. We are used to thinking of the world in terms of objects, their attributes and capabilities and the relationships between objects.

Two of the most intuitive relationships that exist between objects are **aggregation** and **inheritance**.

**Aggregation:** It is simply the idea that some objects (and we can include abstract concepts as objects) are **composed of** other objects. An animal is composed of body parts (arms, legs, head, etc), a car is composed of car parts (engine, wheels, etc), a computer program is composed of instructions and data.

We also tend to classify objects in the real world based on another kind of relationship. Dogs are a **kind of** mammal. A Doctor **is a** human being. Java **is a kind of** programming language. This idea is called **inheritance**. The class being inherited from is called the superclass, the class inheriting is called the subclass. In the example above the doctor is a subclass of human. Human is a superclass of doctor.

In Java a subclass contains all the methods and variables defined in the super-class plus the subclass's own methods and variables. Java implements inheritance through the **extends** keyword. The doctor has all the attributes of being human plus extra attributes picked up in medical school.

A doctor is a human.  
A human is a mammal.  
Therefore a doctor is also a mammal.

In java we would represent this as:

class human extends mammal  
class doctor extends human.

The doctor class gets all the class variables and methods defined in class mammal, class human, and class doctor.

However, imagine we have a class called employee that doesn't extend any other class. Java does not let doctor extend employee and human at the same time. Only one class can be extended by a subclass. This is called single inheritance (some other languages allow multiple inheritance).

Things start to get a little complicated (and useful) when we consider **polymorphism**. Polymorphism just means we can treat an object as if it were any of the subclasses it inherits from. For example, we can treat the doctor object as being of class mammal, class human, or class doctor.

This makes things easy in some cases, for example, when is do not matter to you that a mammal object is also a doctor.

In Java we write:

|  |
| --- |
| //In Mammal.java  class Mammal  {  protected int age;  protected float weight;    Mammal(float w, int a)  {      age = a;      weight = w;  }    public float getWeight()  {      return weight;  }    public int getAge()  {      return age;  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | //in Human.java:    class Human extends Mammal  {  protected float height;  protected String nationality;    Human(float h, int a)  {      age = a;      height = h;  }    public float getHeight()  {      return height;  }    public int getNationality()  {      return nationality;  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | //in Doctor.java:    class Doctor extends Human  {  private School school; // School is a class not shown  private Specialty specialty; // Specialty is a class not shown    Doctor(School sc, Specialty sp)  {      school = sc;      specialty = sp;  }    public float getSchool()  {      return height;  }    public int getSpecialty()  {      return nationality;  }    public float calcSalary()  {      return school.getValue() + specialty.getValue + getAge();  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Soldier extends Human  {  private Branch branch; // Branch is a class not shown  private Rank rank; // Rank is a class not shown    Soldier(Rank r, Branch a)  {      rank = a;      branch = b;  }    public float getRank()  {      return height;  }    public int getBranch()  {      return branch;  }    public float calcSalary()  {      return rank.getValue() + branch.getValue + getAge();  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //Inheritance allows us to do the following:    Doctor doc  = new Doctor();  Soldier sol  = new Soldier();    String spec = doc.getSpecialty();  String height= doc.getHeight();  String age = doc.getAge();    String salary = sol.calcSalary();  String height= sol.getHeight();  String age = sol.getAge(); |

Notice the keyword **protected** in the block of code above. We have already seen that class variables and class methods can be declared **private** or **public**. Private means the method or variable can only be used inside the class. Public means that the methods and variables can be called from anywhere. A variable declared to be protected can only be used inside the class and inside any of the class's subclasses. In the above code, age can be use inside the classes Mammal, Human, and Doctor but:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void main(String[] args)  {  Doctor doc = new Doctor();  int docs\_age = doc.age;  } |

Would fail with a compiler error because the main method is not inside Mammal or any of its subclasses and age was declared **protected**.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void main(String[] args)  {  Doctor doc = new Doctor();  int docs\_age = doc.getAge();  } |

Would work because getAge() is declared **public**.

### **Interfaces**

An interface is a template that specifies the methods that must be included by any class that implements the interface.

In large systems this helps to keep the classes "honest" and prevents the program from trying to use an object for a purpose it was not intended to fill. For example a graphics system might try to draw an object using its draw method. This wouldn't work very well if the programmer forgot to implement a draw method. You can also think of an interface as corresponding to the **can** relationship. A doctor **can** diagnose your illness, a human may or may not be able to do so. A **Drawable** object in Java **can** draw itself, some other object that does not **implement** the Drawable interface may or may not be able to draw itself.

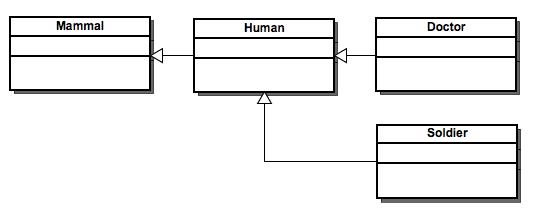
In Java we might write:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Doctor implements CanDiagnose  {      ...        Doctor();      public String diagnose( String[] symptoms )      {      ...      }        ...  } |

The interface would be:

|  |  |
| --- | --- |
| 1  2  3  4 | interface CanDiagnose  {      public String diagnose( String[] symptoms );  } |

The connections between objects can become overwhelming so people often use UML (unified modeling language) diagrams to represent the relationships.



### **Assignment**

For this assignment you will write a program to represent geometric shapes and some operations that can be performed on them. The idea here is that shapes in higher dimensions inherit data from lower dimensional shapes. For example a cube is a three dimensional square. A sphere is a three dimensional circle and a glome is a four dimensional circle. A cylinder is another kind of three dimensional circle. The circle, sphere, cylinder, and glome all share the attribute radius. The square and cube share the attribute side length. There are various ways to use inheritance to relate these shapes but please follow the inheritance described in the table below.

All shapes inherit getName() from the superclass Shape.

**Specification:**

The program should have the following classes: **Shape**, **Circle**, **Square**, **Cube**, **Sphere**, **Cylinder**, and **Glome** and two interfaces **Area** and **Volume** (Area.java and Volume.java are given below).

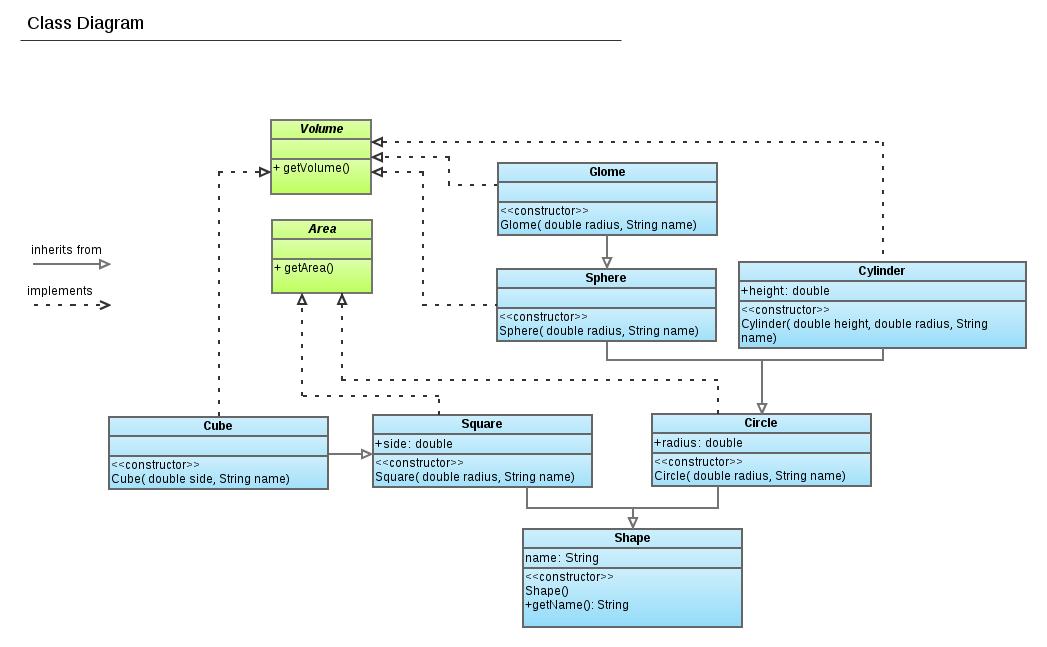
The volume of a glome is 0.5(π2)r4 where r is the radius.

Your classes may **only** have the class variable specified in the table below and the methods defined in the two interfaces Area and Volume. You will implement the methods specified in the Area and Volume interfaces and have them return the appropriate value for each shape. Class Shape will have a single public method called getName that returns a string.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Class Variable** | **Constructor** | **Extends** | **Implements** |
| Shape | String name | Shape() |  |  |
| Circle | double radius | Circle( double r, String n ) | Shape | Area |
| Square | double side | Square( double s, String n ) | Shape | Area |
| Cylinder | double height | Cylinder(double h, double r, String n ) | Circle | Volume |
| Sphere | None | Sphere( double r, String n ) | Circle | Volume |
| Cube | None | Cube( double s, String n ) | Square | Volume |
| Glome | None | Glome( double r, String n ) | Sphere | Volume |

Your program will use the following:

* [Area.java](https://fricke.co.uk/Teaching/CS251/lab4/Area.java)
* [Volume.java](https://fricke.co.uk/Teaching/CS251/lab4/Volume.java)



**Submission:**

**Don’t copy**

**Honestly is very important.**

Submit a file called **assignment\_Inh.zip** containing your seven class source files: Demo.java,Shape.java, Square.java, Circle.java, Sphere.java, Cylinder.java, Cube.java, and Glome.java on CANVAS at the due date.