Chapter 5:

OOP concepts

5.1. Introduction

One of the main object oriented principles is Encapsulation, Bringing data and methods in a single unit. Binding or grouping of data and methods is implemented using a “class” construct. The other pillars of Object-oriented programming are: Inheritance, Polymorphism, and Communication with messages. Inheritance and communication with message are the subject matters of this chapter.

5.2. Encapsulation

- It separates the implementation details from the interface.
- Object orientation uses classes to encapsulate (i.e. Wrap together) data and methods.
- It is the mechanism by which implementation details are hidden. Encapsulation enables objects to hide their implementation from other objects, a principle called information hiding. This is done through security and visibility.
- The interface (method) is visible to the entire world, and we can hide other variables and methods so that they cannot be accessed. Methods are provided to allow to variables (State) and to change the state. The only access to the object’s variables is through method calls to check their state and there is no direct access to object’s variables.

Access modifiers: are keywords used to determine the level of access to class members.

- *Private*: Variables or methods declared with access modifier private are accessible only to methods of the class in which they are declared. Only other members of the same class can have access. It indicates that instance variables are accessible only to methods of the class; this is known as data hiding.
- *Public*: Variables or methods declared public are accessible where ever the program has a reference to an object of the class members.
- *Default*: within the package (sometimes called friendly). If none of the other modifiers are used.

- *Protected*: Offers an intermediate level of access b/n public and private. A super class' protected members can be accessed by members that super class, by members of any classes derived from that superclass and by members of other classes in the same package. Access is within the package and derived classes.

E.g. Let us modify the circle class.

```
class Circle{
        private float rad;
        private int xCoord, yCoord;
        public void showArea(){
                float  area = Math.PI*rad*rad;
                System.out.println("The area is:" + area);
        }
        public void showCircumference(){
                float circum=2* Math.PI*rad;
                System.out.println("The circumference is:" + circum);
        }
        public void setRadius(float r){
                rad = (r<0.0f?0.0f:r);
        }
        public float getRadius(){
                return rad;
        }
}
class DriveClass{
        public static void main(String args[]){
        Circle c1 = new Circle();
        c1.setRadius(10);
        c1.showArea();
        float r = c1.getRadius();
        }
}
```

Setter methods - used to assign values to attributes.

Getter methods – used to retrieve values.

Private methods are known as utility methods or helper methods b/c they can be called only by other methods of that class and are used to support the operation of those methods. Methods other than the above three methods are called service methods.

Exercise: Define the point class given the driver class.

```
class PointTest{
        public static void main(String args[]){
                Point p1 = new Point();
                p1.setCoordinates(1,5);
                p1.addToXCoordinate(3);
                p1.showCoordinates();
                Point p2;
                p2 = p1.CopyMe();
        }
}
```

Class modifiers

Allowed: abstract, final, extends, implements, public.

Not Allowed: protected, private, native, static, synchronized.

- The **current object** is an object that actively executes a method or it is an object receiving a message. '***this***' keyword can be used inside any method to refer to the current object.
- Each object can access a reference to 'this', (sometimes called the 'this' reference). In a method, the 'this' reference can be used implicitly and explicitly to refer to the instance variables and other methods of the object on which the method was called.
- If a method contains a local variable with the same name as a field of that class, that method will refer to the local variable, rather than the field. In this case the method can use 'this' to refer to the shadowed field explicitly.
- If a method declares a local variable with the same name as a variable declared in the method's enclosing class the variable declared in the class' scope is shadowed (hidden) by the local variable.

E.g.
```
void setCoordinates(int xCoord, int yCoord){
        this.xCoord = xCoord;
}
```

5.3. Inheritance

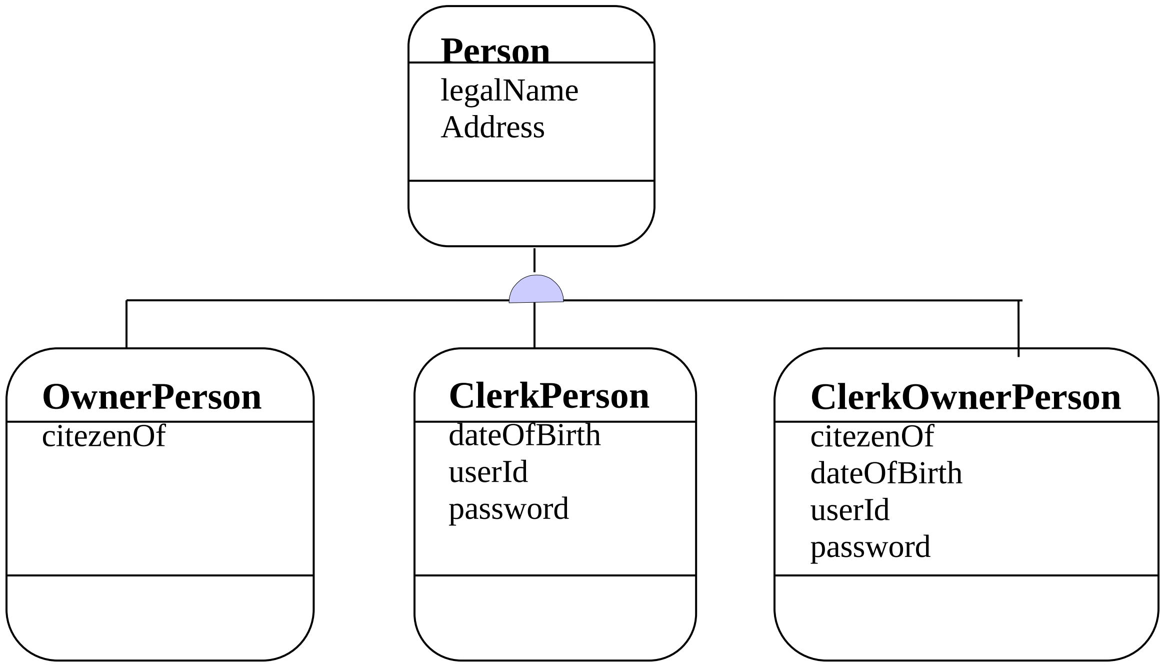
- Human-being organizes his/her knowledge in two ways: (or abstract ways)
  1. Generalization/Specialization (Gen-Spec)
     - Relationship or 'is a" relationship.
     - Generalization is a relationship between a class and one or more refined versions of it.
     - Example:

       "Oak <u>is a</u> tree."
       - Here we imply that an oak is a tree and inherits many of the general features that we associate with the tree.
       - Oak has some specialized features of its own like corn.
     - Defines as inheritance relationship/hierarchy of classes
  2. Whole-part
     - Part of or "has a" relationship.
     - Whole-part is another way of organizing knowledge.
     - It is a grouping relationship
     - Example:

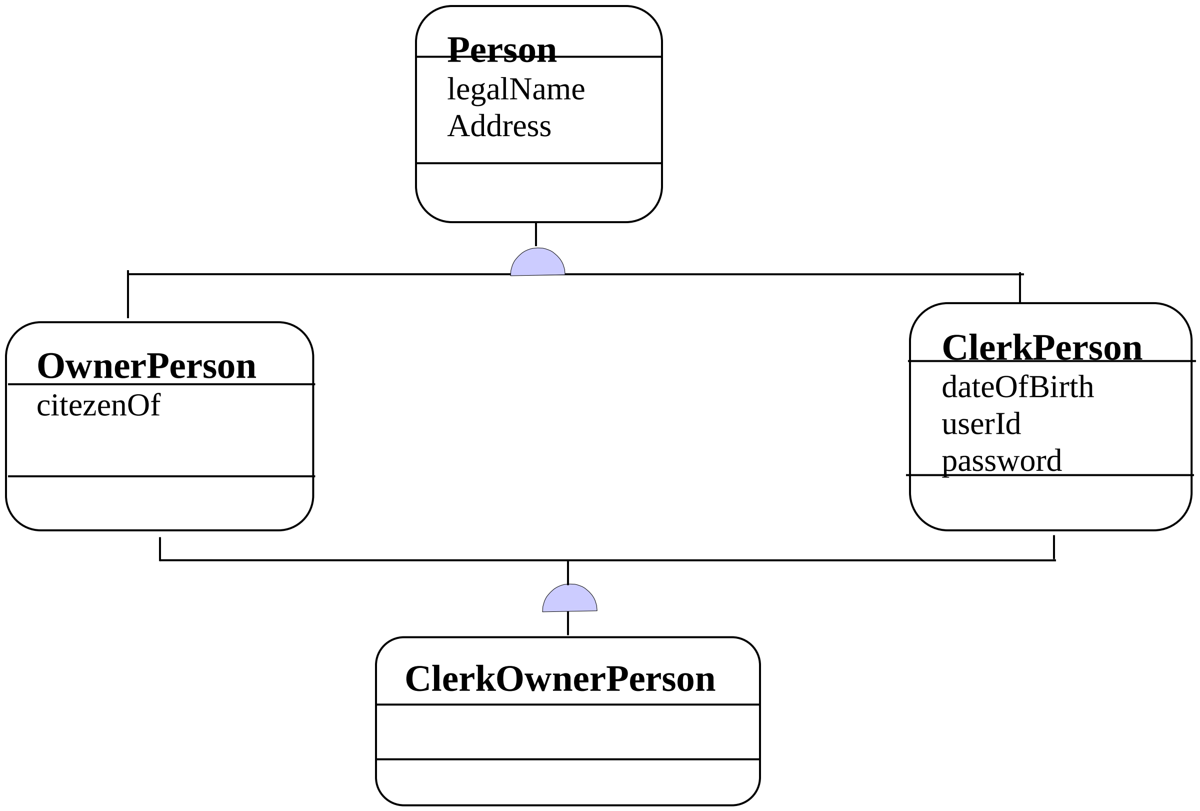
       Tree has branches
     - Three are three places where you may find whole-part
       - Assembly-Part
       - Container-Contents
       - Collection-Members

<u>Example</u>:

Consider owner (of a vehicle) and clerk (working in vehicle registration). A clerk may own a vehicle as well. Gen-Spec structure showing the relationship:

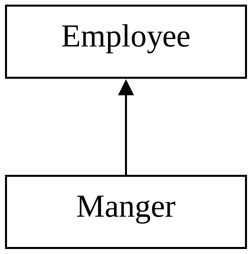


- Note that we have some redundancies across the specializations. To avoid this problem lets redefine the class relationship.



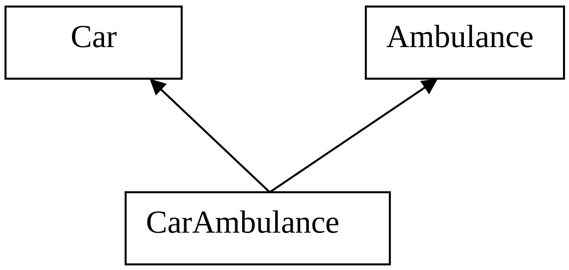
- We can see here multiple inheritances in which the class ClerkOwnerPerson inherits from more than one class (i.e tow classes).
- Inheritance is realized by the Generalization specialization relationship. It allows deriving more specialized classes from the existing one (the new class should be inherit the members from an existing class).
- The existing class is called the super class (In C++ base class).
- The new class that inherits 'all' the properties of the super class by adding behaviors specific to it is called the Subclass (In C++, derived class).
- The Object class is the ancestor of all Java classes (the root class).
- All other classes are implicitly the subclass of the Object class.
- Inheritance is the mechanism for code reuse.
- It allows us to extend the functionality of an object, in other words, we can create new types with extended properties from the original type.
- Generally there are tow types of inheritance in OOP.
    1. **Single Inheritance** – A class inherits from only one super class.

       Example:



    2. **Multiple Inheritances**: A class inherits from more than one superclass. Java doesn't support multiple inheritance but C++ supports.

       Example:

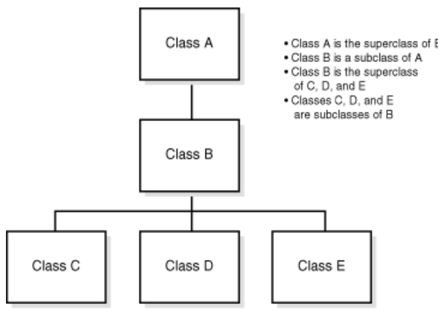


- A subclass is a class that extends another class.
- A subclass inherits state and behavior from all of its ancestors.
- The term "superclass" refers to a class's direct ancestor as well as to all of its ascendant classes.
- A subclass inherits variables and methods from its superclass and all of its ancestors.
- The subclass can use these members as is, or it can hide the member variables or override the methods.
- A subclass inherits all of the members in its superclass that are accessible to that subclass unless the subclass explicitly hides a member variable or overrides a method.
- Note that constructors are not members and are not inherited by subclasses.
- The following list itemizes the members that are inherited by a subclass:
    - Subclasses inherit those superclass members declared as public or protected.
    - Subclasses inherit those superclass members declared with no access specifiers as long as the subclass is in the same package as the superclass.
    - Subclasses don't inherit a superclass's member if the subclass declares a member with the same name. In the case of member variables, the member variable in the subclass hides the one in the superclass. In the case of methods, the method in the subclass overrides the one in the superclass.

Syntax:

```
class subclassName extends superClassName{
        …
}
```

Example1:


Class A
Class B
Class C
Class D
Class E
• Class A is the superclass of
• Class B is a subclass of A
• Class B is the superclass of C, D, and E
• Classes C, D, and E are subclasses of B

```
class A{
                int x;
                void showX()
                {
                        System.out.println(x);
                }
}
class B extends A{
                int y;
                void showXY(){
                        System.out.println(x);
                        System.out.println(y);
                }
}
```

Example2:

```
class ClassSuper{
        int x,y;
        private int p;
void displayXYP(){
                System.out.println("x & y in superclass:" + x+" "+ y);
                System.out.println("P is private:"+p);
        }

        ClassSuper(){
                P = 5;
                x = 0; y = 0;
        }
}
class ClassSub extends ClassSuper{
        int z;
        void showz(){
                z = x + y + p; //p is private and not inherited
                z = x + y;
                System.out.println("x+y = " + z);
        }
}
//Driver class
```

```
        class TestInheritance{
                public static void main(String args[]){
                        ClassSuper objSup = new ClassSuper();
                        ClassSub objSub = new ClassSub();
                        objSup.x = 10; objSup.y = 20;
                        objSub.displayXYP();
                        objzdub.showz();
                }
        }
```

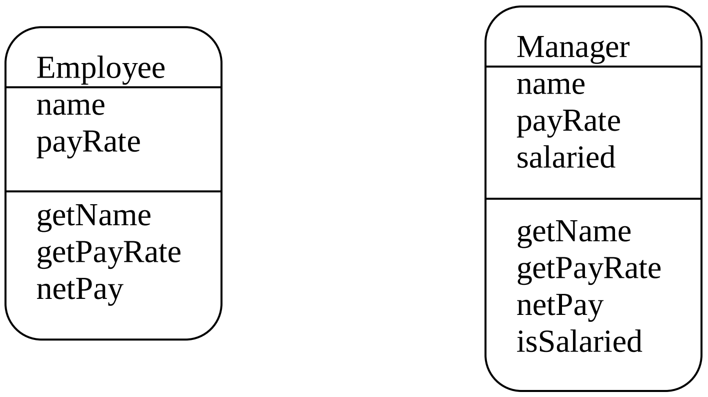
Output: x & y in super: 10 20

```
        p is private: 5
        x + Y = 30
```

- Note that private member p of the ClassSuper can not be accessed by using a reference to object of the ClassSub and also by member methods of the ClassSub.
- Allowing private members of a super class to be accessed by subclass, would propagate access to what should be private data, and the benefit of data/information hiding would be lost.

Referencing a subclass object using a super class variable

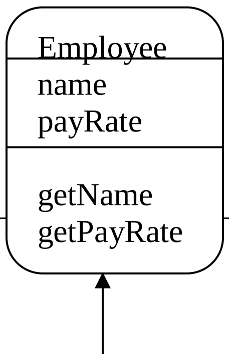
- Consider the relationship between an Employee and a Manager object.



- A manager is an Employee (Employee is generic and a manager is a specialized Employee).

//Here is a class definition for the generic Employee

```
public class Employee{
        protected String name;
        protected float payRate;
```



```
        public Employee(String nm, float pr){
                name = nm;
                payRate = pr;
        }
        public String getName(){
                return name;
          }

        public float getPayRate(){
                return payRate;
        }
          public float netRate(float hrsWorked){
                return hrsWorked * payRate;
          }
}
//The Manager Subclass
public class Manager extends Employee{
        protected boolean salaried;
        public Manager(String nm, float pr, boolean isSal){
                                                          ⎧ name = nm;
                super(nm, pr);    ◄───────────────────────⎨ payRate = pr;
                salaried = isSal;                         ⎩
        }
        public boolean getSalaried(){
                return salaried;
        }
        public float netPay(float hw){
                if(salaried )
                        return payRate;
                return (super.netPay(hw));
        }
}
```

Note: Because a Manager is a kind of Employee, a Manager can be used as an Employee.

However, an Employee can not be used as a Manager.

Example: Manager mg1 = new Manager();

```
Employee emp1 = new Employee();
Manager mg2 = new Employee();      //Wrong
Employee emp2 = new Manager();
emp1 = mg1;
```

- If we want the object referenced by emp1 to be referenced by a reference to a manager object mg1 (i.e if want to consider an Employee object a Manager object), we need type casting.

```
mg1 = emp1;   //Wrong they are incompatible types
mg1 = (Manager) emp1;
```

- Down casting is a process of casting a super class reference that is referring to subclass object to subclass reference.

  Syntax:

```
<Sub_Ref.> = (SubClassName)<sup_Ref.> ;
```

Example:

```
Employee emp = new Manager ();
Manager mgr = (Manager) emp;
```

NB: To assign super class reference variable to a subclass variable we use down casting, but the super class reference variable should refer to a subclass object. Otherwise it will compile correctly but there will be an error at runtime.

Example:

```
Employee emp = new Employee();
Manager mgr = (Manager) emp;
```

Hiding Member Variables

- As mentioned in the previous section, member variables defined in the subclass hide member variables that have the same name in the superclass.
- While this feature of the Java language is powerful and convenient, it can be a fruitful source of errors. When naming your member variables, be careful to hide only those member variables that you actually wish to hide.
- One interesting feature of Java member variables is that a class can access a hidden member variable through its superclass. Consider the following superclass and subclass pair

Using the super keyword

- **super** is a Java language keyword that allows a method to refer to hidden variables and overridden methods of the superclass.

```
class Super {
   float aNumber;
}
class Subbie extends Super {
   float aNumber;
}
```

The aNumber variable in Subbie hides aNumber in Super. But you can access Super's aNumber from Subbie with:

```
super.aNumber
```

- It is possible for a subclass constructor to call a constructor in the superclass. We use the super(…) together with the appropriate parameter list.

  Syntax:

```
super(arg. list);
```

Note: super(…) must always be the first statement executed inside a subclass constructor.

- Java allows us to use super keyword to invoke a super class's constructor for two reasons:
  i) Not to have a duplicate code in the subclass while it is already found in the super class.
  ii) The super class might want to keep the details of its implementation to itself by making data members private.
- To refer to a super class's method from a subclass, especially if there are overridden methods, we use super together with dot (.) operator to invoke superclass's method.

  Example: super.netPay(hw);

Overriding Methods

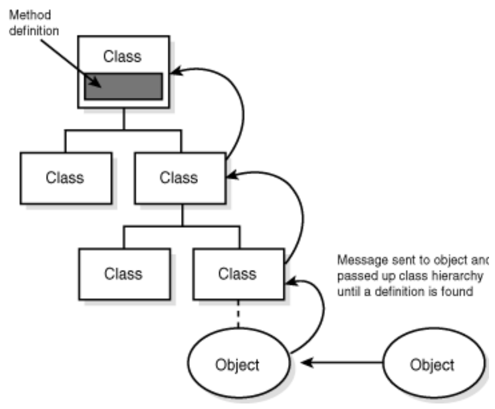
Going from the general cases to its specializations, some behaviors might be implemented differently. Whenever there is such a difference in implementation of behavior and the base class implementation is wrong in the subclass's context, we use method overriding.

Definition: When a method in a super class has the same name as and type signature as a method in the superclass, then the subclass method is said to override the method in the superclass. The return type, method name, and number and type of the parameters for the overriding method must match those in the overridden method.

- The version of the method defined by the superclass will be hidden.
- The access specifiers for the overriding method can allow more access than the overridden method, but not less. For example, a protected method in the superclass can be made public but not private. So, it is a syntax error to override a method with a more restricted access modifier.
- For example in the Employee and Manager classes netPay is overridden in the subclass Manager.
- Whenever a subclass method wants to call its superclass's overridden method prefix the keyword super and a dot (.), failure to do so causes infinite recursive call, because the subclass method would then call itself.
- The ability of a subclass to override a method in its superclass allows a class to inherit from a superclass whose behavior is "close enough" and then override methods as needed.

For example, all classes are descendents of the Object class. Object contains the toString method, which returns a String object containing the name of the object's class and its hash code. Most, if not all, classes will want to override this method and print out something meaningful for that class.

Let's resurrect the Stack class example and override the toString method. The output of toString should be a textual representation of the object. For the Stack class, a list of the items in the stack would be appropriate


Method definition
Class
Class
Class
Class
Class
Message sent to object and passed up class hierarchy until a definition is found
Object
Object

Example 1:

```
public class Stack
{
   private Vector items;
   // code for Stack's methods and constructor not shown
   // overrides Object's toString method
   public String toString() {
      int n = items.size();
      StringBuffer result = new StringBuffer();
      result.append("[");
      for (int i = 0; i < n; i++) {
         result.append(items.elementAt(i).toString());
         if (i < n-1) result.append(",");
      }
      result.append("]");
      return result.toString();
   }
}
```

Calling the Overridden Method

- Sometimes, you don't want to completely override a method. Rather, you want to add more functionality to it. To do this, simply call the overridden method using the super keyword. For example,

```
super.overriddenMethodName();
```

Example 2:

```
import java.util.Date;
class  Student{
        String name;
        int id;
        int age;
        Date admDate;
        Student(String nm, int id, int age){
                name = nm;
                this.id = id;
                this.age = age;
                admDate = new Date();
        }
        public String toString(){
                return “Stud name:” + name + “/n”+
                “Stud. Id:” + id + “/n”+
                “Stud age:” + age +”/n”+
                “Admission Date:” + admDate;
        }
}

class ImplementToString{
        public static void main(String args[]){
                Student stud = new Student (“name1”, 1, 23);
                System.out.println(stud);
        }
}
```

Output:

```
Stud name: name1
Stud Id: 1
Stud age: 23
Stud name: Mon Nov 22 03:56:55 PST 2004
```

The **toString** method of class Object is overridden in the Student class, which is a subclass of Object class. The original toStirng method of class Object is generic version, used mainly as a placeholder that can be overridden by a subclass.

Inheritance and Constructor call sequence

- When you create an object of the subclass, it contains a sub-Object of the super class within it.

- It is essential that the super class sub-Object be initialized correctly and there is only one way to guarantee that perform initialization inside the constructor.
- Java automatically inserts calls to the super class constructor in the derived class constructors.
- The first task of any subclass constructor is to call its superclass's constructor, either implicitly or explicitly.
- If super (…) is not used, then the default or parameter less constructor of each superclass will be executed.

Example:

```
class A{
            •       •       •
        A(){
            //implicit call to Object constructor
            System.out.println("Inside A's Constructor");
        }
}
class B extends A{
        •       •       •
        B(){
            //implicit call to A's Constructor
            System.out.println("Inside B's Constructor");
        }
}
class C extends B{
        •       •       •
        C(){
            //implicit call to B's Constructor
            System.out.println("Inside C's Constructor");
        }
}
class DriveConstructor{
        public static void main(String args[]){
            B objB = new B();
```

```
                C objC = new C();
        }
    }
```

```
output: Inside A’s Constructor
        Inside B’s Constructor
        Inside A’s Constructor
        Inside B’s Constructor
        Inside C’s Constructor
```

- Therefore constructor call sequence follows order of derivation.

5.4. Abstract Classes and Methods

Abstract Classes

- Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated.

  Take, for example, food in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate. Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.
- Similarly in object-oriented programming, you may want to model an abstract concept without being able to create an instance of it. For example, the Number class in the java.lang package represents the abstract concept of numbers. It makes sense to model numbers in a program, but it doesn't make sense to create a generic number object. Instead, the Number class makes sense only as a superclass to classes like Integer and Float, both of which implement specific kinds of numbers. A class such as Number,

which represents an abstract concept and should not be instantiated, is called an abstract class. An abstract class is a class that can only be subclassed-- it cannot be instantiated.

- While working with OOPs one may encounter classes showing similarities in behavior but that do not belong the inheritance hierarchy. One of the reasons for their similarities can be because they belong to the same base class. If the system does not have entities of the base class type, we define these classes as abstract.
- Abstract classes are just used to capture the communalities between similar subclasses (or are introduced for the purpose of inheritance).
- To declare that your class is an abstract class, use the keyword abstract before the class keyword in your class declaration:

Syntax:

```
abstract class <class idn.>
 {
        . . .
}
```

Example:

```
abstract class Number {
        . . .
}
```

If you attempt to instantiate an abstract class, the compiler displays an error similar to the following and refuses to compile your program:

```
AbstractTest.java:6: class AbstractTest is an abstract class.
It can't be instantiated.
    new AbstractTest();
    ^
1 error
```

- Since abstract classes represent an entity too general to be real, one can not create instances (objects) of the abstract type.

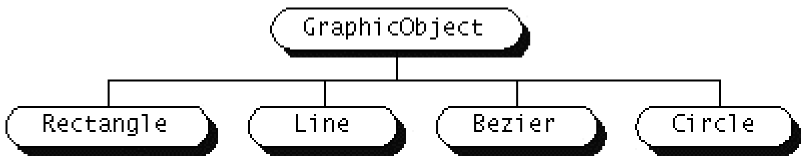
Example:

```
abstract class Person{
. . .
}
```

```
class Student extends Person{
        . . .
}
class Employee extends Person{
        . . .
}
class DriveAbstract{
        public static void main(String args[]){
                Employee emp = new Employee();
                Student stud = new Student();
                Person p = new Person();        //error Person can't be
                                                instantiated
        }
}
```

Abstract Methods

- An abstract class may contain abstract methods, that is, methods with no implementation. In this way, an abstract class can define a complete programming interface, thereby providing its subclasses with the method declarations for all of the methods necessary to implement that programming interface. However, the abstract class can leave some or all of the implementation details of those methods up to its subclasses.
- Let's look at an example of when you might want to create an abstract class with an abstract method in it. In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and so on. Each of these graphic objects shares certain states (position, bounding box) and behavior (move, resize, draw). You can take advantage of these similarities and declare them all to inherit from the same parent object--GraphicObject.



- However, the graphic objects are also substantially different in many ways: drawing a circle is quite different from drawing a rectangle. The graphics objects cannot share these types of states or behavior. On the other hand, all GraphicObjects must know how to draw themselves; they just differ in how they are drawn. This is a perfect situation for an abstract superclass.
- First you would declare an abstract class, GraphicObject, to provide member variables and methods that were wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw, that need to be implemented by all subclasses, but are implemented in entirely different ways (no default implementation in the superclass makes sense). The GraphicObject class would look something like this:

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

- Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, would have to provide an implementation for the draw method.

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

```
        }
```

- An abstract class is not required to have an abstract method in it. But any class that has an abstract method in it or that does not provide an implementation for any abstract methods declared in its superclass must be declared as an abstract class.
- An abstract method is a method with no implementation. An abstract method must be overridden in a subclass, other wise the subclass itself should be declared as abstract.

Example: The figure class with two subclasses Rectangle and Triangle

```
//abstract Figure class: Figure.java
    public abstract class Figure{
        protected float x,y;
        public vid setDim(float a, float b){
                x = a;
                y = b;
        }
        public abstract vid showArea()
    }
    //Rectangle subclass: Rectangle.java
    public Rectangle extends Figure{
        public void showArea(){
            System.out.println("Area=" + x*Y);
        }
        public boolean isSquare(){
            if(x == y)return true;
            else
                return false;
        }
    }
    //Triangle subclass: Triangle.java
    class Triangle extends Figure{
        public viod showArea(){
                System.out.println("Area=" + 0.5*x*y);
        }
        public Boolean isRightAngle(){
            .
            .
            .
```

```
          }
    }
```

5.5. Final classes and methods

<u>**final method**</u>

- Making a method final ensures that the functionality defined in this method will never be altered in any way (i.e. final methods can not be overridden).

<u>**Syntax**</u>:

```
    final <ret. type> <methodName>(…){…}
```

<u>**final class**</u>

- A class that can not be subclasses is called a final class (they are not inherited by other classes).

Syntax:

```
    final class CName
    {
          •     •     •
    }
```

Example:

```
    class A{
              final void method(){…}
    }
    class B extends A{
              void method(){…}     //Wrong
    }
```

Example:

```
    final class A{…}
    class B extends A{…}
```

Comparison between abstract and final classes

- final class must not be subclassed, but an abstract class must be subclassed.
- A final method must not be overridden, but an abstract method must be overridden.

- final keyword is used to prevent inheritance, but abstract keyword is used to prevent instantiation.

Methods a Subclass Cannot Override

- A subclass cannot override methods that are declared final in the superclass (by definition, final methods cannot be overridden). If you attempt to override a final method, the compiler displays an error message similar to the following and refuses to compile the program:

```
FinalTest.java:7: Final methods can't be overridden.
Method void iamfinal() is final in class ClassWithFinalMethod.
void iamfinal() {
        ^
1 error
```

- Also, a subclass cannot override methods that are declared static in the superclass. In other words, a subclass cannot override a class method.
- A subclass can hide a static method in the superclass by declaring a static method in the subclass with the same signature as the static method in the superclass.
- A subclass must override methods that are declared abstract in the superclass, or the subclass itself must be abstract.

5.6. Interface

- An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference.
- The difference is that interfaces define only abstract methods and final fields.

Creating an interface

- The syntax for creating an interface is similar to that for creating a class.
- The most significant difference is that none of the methods in your interface may have a body.

Syntax:

```
interface InterfaceName{
        Variable declaration;
        Method declaration;
}
```

variable declaration

[static] [final] type va_Name = <value>;

method declaration

[public]ret_Type methodName(arg_List);

Example:

```
//shape.java
interface Shape{
        final double PI = 3.14;
        public abstract double Volume();
        public abstract double getName();
}
```

Note:

- ☞ All the methods are defined as abstract (method with no implementation).
- ☞ Interface shall also be saved in Java file using .java extension.
- ☞ If there are attributes in an interface they should be static final.

Example 2:

```
//class Point implements the Shape interface
public class Point extends Object implements Shape{
        protected int x, y;
        public Point(){
            setPoint(0, 0);
        }
        public Point (int xCoord, int yCoord){
            setPoint (xCoord, yCoord);
        }
        public setPoint (int xCoord, int yCoord){
            x = xCoord;
            y = yCoord;
        }
        public void getX(){return x;}
        public double area(){return 0.0;}
        public double volume(){return 0.0;}
```

```
                    public String getName(){return "Point";}

        }
```

5.7. Polymorphism

- Polymorphism, in general means one thing having different forms.
- It is a feature in OOP, that allows objects to take more than one shape (form or appearance) based on the environment they are working in (Object polymorphism).
- It is the ability that different objects respond distinctively to the same message. For example, when you send the same message, "speak" to a Cat object, a Dog object, and a Cow object, each one respond appropriately. The Cat purrs, the Dog barks, and the Cow moos (Feature polymorphism).

Up casting

- The most important aspect of inheritance is not only that it provides methods for the new class but it is the relationship expressed between the new classes and the base class.
- This relationship can be summarized by saying "The new class is a type of the existing class".
- It is supported by the language by its feature of up casting. In up casting one can use references of the base type to refer to instances of derived class type.

Example:

```
class A{
                    int x;
                    . . .

        }
        class B extends A{
                    int y;
                    . . .
        }
```

```
B refb;
refb = new A();
A refa = new B();
refa = new B();
refa.x = 10;
refa.y = 20;      //wrong
refb = (B)refa;
refb.x = 10;
refb.y = 20
```

Tip: It is possible to downcast objects when a base reference refers to a derived object and you want to give the object to a derived reference.

```
Base Ref. = new Derived();
Der. Ref = (Derived)BaseRef;
Base Ref. = new Base();           ⎫
                                  ⎬ Error
Der. Ref = (Derived) Base Ref.;   ⎭
```