CS2

Theme:

Object Oriented Programming in Java

Handout : Week9

prepared by N S kumar

Preface

Dear friends,

In this lecture notes, we will discuss the concept of inheritance – which is normally not well understood either in the industry or in the academics. Please go through these notes carefully and try to appreciate the need for inheritance.

This topic is not included for the test T2. The format of the question paper shall be similar to the one we had for T1. You will be required to read and understand the programs or methods, you may have to find the output of small program snippets, you may have to write your own code and you may have to answer questions about finer points of Object Oriented Programming in Java. Do not forget to put your thinking hat that day!

Believe me, it is very difficult to get zero in these tests.

All the best. May all of you score above 90%.

N S Kumar
Visiting Professor
Dept. Of Computer Science and Engineering
PES Institutions.
Email : ask.kumaradhara@gmail.com
Date : 15th March 2015

Making the client life easier:

The idea of development of software is to make the life of the user – who is another developer – easy. Can we support different shapes and make the user of shape play with different shapes uniformly. He should be able to select Rectangle or Circle. Once he selects the shape, he should be able to display the attributes of that shape, find its area, find its perimeter and so on. Here is our first attempt to support the client – but as it turns out – this is a very bad way of development of software!

```
//L32/Rect.java
import java.util.*;
public class Rect
{
	private double length;
	private double breadth;

	public void read()
	{
		Scanner in = new Scanner(System.in);
		length = in.nextDouble();
		breadth = in.nextDouble();
	}

	public void disp()
	{
		System.out.println(length + ":" + breadth);
	}

	public double area()
	{
		return length * breadth;
	}
```

```
	public double peri()
	{
		return 2 * (length + breadth);
	}
}

//L32/Circle.java
import java.util.*;
public class Circle
{
	private double radius;
	public void read()
	{
		Scanner in = new Scanner(System.in);
		radius = in.nextDouble();
	}
	public void disp()
	{
		System.out.println(radius);
	}
	public double area()
	{
		return 3.1416 * radius * radius;
	}
	public double peri()
	{
		return 2 * 3.1416 * radius;
	}
}
```

At this point, we have two different shapes for the client to select – Rect and Circle. The class Shape should have one of them depending on the choice of the client. But unfortunately this concept called union in 'C' is not supported in Java. As we have decided to go with Java for OO programming in this course, we have to come out with an alternate solution.

How about making the Shape class have both Rect and Circle. I have two cars and on a given day, I use one of them. These shall be stored as references to Rect and Circle – these references occupy very small amount of space. We will create the Shape of user's choice when he asks for it. Alas, this idea does not apply to my cars!

Our mother (anybody who shows the motherly affection – could be father or spouse or may be our child!) will pack our lunch in the same box each day. Its content will be different each day. How do we make out what it contains? May be You may decide, based on the content, whether to feed yourself or the very friendly dog in the campus!

It will be so nice if there is a small transparent portion through which we can make out what the box contains. Such a concept in programming is called discriminator. We do require one for the Shape class. Otherwise we will not know whether we have a Rect or a Circle. We may implement this discriminator with an enum class.

```
//L32/ShapeType.java
public enum ShapeType
{
	RECT, CIRCLE;
}
```

```
//L32/Example1.java

// This program shows how not to write programs
// shows the necessity of inheritance.
// Liskov's property of substitution:
//       a) we can replace object of a super class by an object of the subclass
//  b) interface of the base class is the interface of the derived.
public class Example1
{
        public static void main(String[] args)
        {
                Shape s = new Shape(ShapeType.RECT);
                s.read();
                s.disp();
                System.out.println("peri " + s.peri());
                System.out.println("area " + s.area());

                s = new Shape(ShapeType.CIRCLE);
                s.read();
                s.disp();
                System.out.println("peri " + s.peri());
                System.out.println("area " + s.area());
        }
}
```

This program appears super simple. Is it not? The client specifies which type of Shape he wants while invoking the constructor. The remaining code is exactly same for both Rect and Circle. The client life has become easy – but at what cost? Let us inspect the Shape class.

```
//L32/Shape.java
// behaviour based on value of a type makes the code brittle.
// when a new type is added, code breaks down
// we may have to add a few additional clauses in selection
// this is a bad design
public class Shape
{
	private Rect rect;
	private Circle circle;
	private ShapeType st;
	public Shape(ShapeType st)
	{
		this.st = st;
		if(st == ShapeType.RECT)
		{
			rect = new Rect();
		}
		else if(st == ShapeType.CIRCLE)
		{
			circle = new Circle();
		}
		else
		{
			rect = null;
			circle = null;
		}
	}
```

```
public void read()
{
	if(st == ShapeType.RECT)

	{
		rect.read();
	}
	else if(st == ShapeType.CIRCLE)
	{
		circle.read();
	}
	else
	{
		System.out.println("read error");
	}
}

public void disp()
{
	if(st == ShapeType.RECT)
	{
		rect.disp();
	}
	else if(st == ShapeType.CIRCLE)
	{
		circle.disp();
	}
	else
	{
		System.out.println("disp error");
	}
}
```

```
	public double peri()
	{

		if(st == ShapeType.RECT)
		{
			return rect.peri();
		}
		else if(st == ShapeType.CIRCLE)
		{
			return circle.peri();
		}
		else
		{
			return 0.0;
		}
	}
	public double area()
	{
		if(st == ShapeType.RECT)
		{
			return rect.area();
		}
		else if(st == ShapeType.CIRCLE)
		{
			return circle.area();
		}
		else
		{
			return 0.0;
		}
	}
}
```

We observe that every method is littered with selection. We should check each time whether the Shape is a Rect or a Circle. Based on that, we should invoke the method of Rect or Circle. This appears fine so far.

What if we want to support another shape – say Square. Or one more – say Ellipse. We can easily implement these classes - class Square and class Ellipse. We can add a couple of objects to the enum class ShapeType.

What about the Shape class itself? You will observe every method has to be changed. This is definitely a nightmare for software maintenance.
This is a very bad design.

Rule of object oriented programming :
Behaviour of a class should not depend on the value of an attribute.
Use inheritance to avoid behaviour based on the value of an attribute.

Inheritance:
Inheritance is a relationship between classes.
Let us have a class A. We want to make a class B which has every characteristic of A(attributes and behaviour) – but could have something more. Then we use inheritance. We say that class A is a superclass. We say that class B is a subclass. This process of creating B using A is called extending.

Let us have two classes Y and Z. We find that some attributes and behaviour are common to both Y and Z. We remove the commonality and put them into a super class say X and then make Y and Z extend X.

Inheritance will result in a hierarchical relationship between classes - like taxonomy. Let us consider a few examples of relationship between types in the real world.

A 4 Wheeler is a vehicle. A 2 Wheeler is a vehicle. A car is a 4 Wheeler and

therefore is also a vehicle. A jeep is a 4 Wheeler and therefore is also a vehicle. All vehicles support steering, accelerating and braking. Each vehicle will support steering in its own way.

We can use a car when we require a vehicle. This property is called “property of substitution”. We can always replace an object of superclass by an object of subclass. This in turn implies that whatever we can do with the superclass, we can also do with subclass – may be differently.

If we cannot support even a single interface of a superclass in a subclass, we should not use inheritance.

Let us consider a simple example – a class representing birds. One possible abstraction of bird could be that it flies and it plucks worm or a grain of wheat with its beaks. Can we inherit Ostrich from bird?

The answer is NO. We can replace bird with ostrich and then ask that bird to fly. Unfortunately, Ostrich cannot fly!

The basic idea of inheritance is not reuse. A friend of mine is a senior architect in Philips. He says that if has very little time to check somebody on Object Oriented skills, he would ask only one question. Why inheritance? If the candidate says “reuse”, he will reject the candidate.

The idea of reuse is to create relationship between classes - create a class hierarchy. We can replace an object of superclass by an object of subclass. When we invoke methods using superclass object, the runtime will decide to invoke the methods of the subclass to which the object belongs. So, the same method call can result in call to a method of subclass which may not be even known at the point the superclass is implemented.

This behaviour of a method call resulting in call to different methods based on the

object at runtime is called polymorphism.

Inheritance is also a reuse mechanism at the class level. We can not have partial inheritance. We cannot say we want to reuse only 3 methods out of 5 of the superclass. We cannot say we would use pluck of bird in Ostrich without using fly of bird.

Inheritance follows two rules stated by Liskov.

1. Should be able to replace an object of super class by object of sub class
2. interface of the super class should also be interface of the sub class

Inheritance allows us to avoid if statements based on the value of attributes.

Mechanism of inheritance in Java:

Object Layout and Constructor calls:

In the example below, we have created a class called P2D to represent a point in 2 dimensions – this has attributes x and y. We then want to create a class called P3D to represent a point in 3D. We do not want reinvent every wheel. We want to use the class P2D to make the class P3D – which has an additional attribute z.
We use inheritance here. We will extend P3D from P2D.

How will the layout of an object of P3D be? An object of P3D will have a superclass subobject having x and y and additional fields of its class z. Please note that even the private members of the superclass will be allocated as part of the subclass object -even though they may not be accessible.

What happens when an object of subclass is created? Constructor of that class will be called. This has the additional responsibility of initializing the superclass subobject. An object of any class can be initialized by its constructor only. Constructor has to be called on the subobject of superclass which is part of subclass object. The compiler will automatically insert the code to call the constructor of the superclass.

Constructor of subclass will always invoke the constructor of superclass.

```
//L33/Inher2.java
public class Inher2
{
	public static void main(String[] args)
	{
		P3D p3d = new P3D();
	}
}
class P2D
{
	private int x;
	private int y;
	public P2D()
	{
		System.out.println("ctor of superclass");
	}
}



// every member of P2D is also in P3D irrespective of access
// a) show the layout
// b) observe that the call to subclass ctor will in turn call the superclass
//		ctor
//	an object of a class can be initialized by calling its ctor
//	an object of a subclass has an unnamed object of superclass within it
```

```
class P3D extends P2D
{
	private int z;
	public P3D()
	{
		System.out.println("ctor of subclass");
	}
}
```

What if the constructor of the superclass is overloaded and has parameters? The compiler cannot figure out which constructor of superclass to invoke from within the constructor of the subclass.

So, the constructor of the subclass will have to explicitly call the constructor of the superclass. This is make using a keyword super. This statement super(...) will cause a call to the constructor of the superclass. Also, note that this should be the first statement in the subclass constructor. We cannot build the first floor without constructing the ground floor.

The next few examples illustrate the way of invoking the superclass constructor.

//L33/Inher3.java

```
public class Inher3
{
	public static void main(String[] args)
	{
		P3D p3d = new P3D(3, 4 ,5);
	}
}
```

```
class P2D
{
	private int x;
	private int y;
	public P2D(int x, int y) // no default constructor.
	{
		System.out.println("ctor of superclass");
		this.x = x;
		this.y = y;
	}
}

class P3D extends P2D
{
	private int z;
	// compile time error:
	// tries to call the default ctor of P2D; which does not exist.
	public P3D(int p, int q, int r)
	{
		System.out.println("ctor of subclass");
	}
}
```

//L33/Inher4.java

```
public class Inher4
{
	public static void main(String[] args)
	{
		P3D p3d = new P3D(3, 4 ,5);
	}
}
```

```
class P2D
{
	private int x;
	private int y;
	public P2D(int x, int y)
	{
		System.out.println("ctor of superclass");
		this.x = x;
		this.y = y;
	}
}
class P3D extends P2D
{
	private int z;
	// call ctor of superclass by using the keyword super
	public P3D(int p, int q, int r)
	{
		System.out.println("ctor of subclass");
		super(p, q); // Error: should be the first stmt
		// build the foundation before the super structure!
		z = r;
	}
}
```

Default implementation in the superclass:

In this example, we are invoking the method disp on an object of class P3D which extends P2D. What will happen if there is no method called disp in the subclass P3D?

This will result in a call to the superclass disp. Superclass disp provides a default implementation which the subclass can modify. This concept of modifying the superclass method is called overriding. We will discuss this in detail in the next

paragraphs.

//L33/Inher5.java

```
public class Inher5
{
	public static void main(String[] args)
	{
		P3D p3d = new P3D(3, 4 ,5);
		p3d.disp();
	}
}

class P2D
{
	private int x;
	private int y;
	public P2D(int x, int y)
	{
		System.out.println("ctor of superclass");
		this.x = x;
		this.y = y;
	}

	public void disp()
	{
		System.out.println("x : " + x);
		System.out.println("y : " + y);
	}
}
```

```
class P3D extends P2D
{
	private int z;
	public P3D(int p, int q, int r)
	{
		super(p, q); // calls to base class ctor
		z = r;
		System.out.println("ctor of subclass");
	}
	// check what happens if this method does not exist
	public void disp()
	{
		// this.disp() // will be infinite recursion
		// P2D.disp(); // cannot call method of base class in this fashion
		// compiler thinks disp is a static method
		// this is a way to call the method of superclass if the names clash
		super.disp();
		System.out.println("z : " + z);
	}
}
```

Observe in the above example how the disp of the class P3D is implemented. It has the responsibility of displaying its attributes as well as those of its superclass P2D. This method should display its attributes and delegate the responsibility of displaying the superclass attributes to the disp of the superclass. Here again we use the keyword super to refer to the method of the superclass.

Implication of the property of substitution:

Following are valid operations.

- Assign a reference to a subclass object to a reference to superclass.
- Create an object of subclass using new and assign the resulting reference to a superclass reference
- Parameter of a method expects a superclass reference. Pass a reference to a subclass object.

Following is not a valid operation.

- Assigning a superclass reference to a subclass reference.

What can we access using this superclass reference which refers to a subclass object?

We can access only the interface of the superclass. Compiler would not know at compile time whether this reference would refer to an object of superclass or an object of subclass.

//L34/Example1.java

```
// property of substitution
// subclass object when converted to superclass object can access interface
//       of the superclass only
public class Example1
{
        public static void main(String[] args)
        {
                // examine property of substitution
                // we can always replace an object of superclass by an object of subclass
                // upcasting is implicit
                A x = new B();
                //B y = new A(); // error
                // called downcasting; requires an explicit cast; we shall discuss later
```

```
		// another variation
		B y1 = new B();
		A x1 = y1;
		//  yet another variation
		foo(y1);

		// We can access only the superclass interface.
		// subclass object can also access the interface of the superclass
		y1.methodA();
		y1.methodB();
		x1.methodA();
		// x1.methodB(); // Error
	}
	public static void foo(A a)
	{
		System.out.println("foo called ");
	}
}

class A
{
	public void methodA() { System.out.println("I am method A"); }
}

class B extends A
{
	public void methodB() { System.out.println("I am method B"); }
}
```

Polymorphic behaviour:

We call a method using a base class reference. This base class reference at runtime can refer to any subclass object. If the resolution of the method call is based on the object we have at runtime, then we say that this call is polymorphic – many forms. The compiler cannot make out which method to invoke.

We have said earlier also that the superclass provides a default implementation. The subclass can modify this. This is called **overriding.**

Let us list the **differences between overloading and overriding**.

- Overloading requires that the methods have different signatures whereas overriding requires that the methods have the same signatures.
- Overloading does not require multiple classes whereas overriding requires that the methods be in classes with the relation of inheritance.
- Overloading is resolved at compile time whereas overriding is resolved at runtime.

In the example below, you will observe that method foo of class B overrides the method foo of class A. All method calls using the superclass reference x2 which refers to an object of subclass B will result in calls to methods of class B.

//L34/Example2.java

```
// polymorphism
//      concept of virtual method
public class Example2
{
        public static void main(String[] args)
        {
                // test 1:
                A x1 = new A();
                A x2 = new B();
                x1.foo();
                x2.foo();
```

```
            test(x1);
            test(x2);
        }
        public static void test(A a)
        {
            System.out.println("calling foo ");
            a.foo();
        }
}

class A
{
        public void foo() { System.out.println("I am foo of A"); }
}

class B extends A
{
        public void foo() { System.out.println("I am foo of B"); }
}
```

Case for Inheritance:

Let us try this experiment. We will have a class called Tester which has a method called test. This method receives a reference to superclass A as parameter. On this parameter, it invokes a method called foo. This call will behave polymorphically based on which object the superclass reference a refers to.

We will create the classes Tester, A and B along with the driver Example3. We compile all these. We can check the class files in the directory. We will run the program and convince ourselves that the calls to foo using superclass reference a in the test method of the Tester class are polymorphic.

What happens if we now create a new class C which extends A and we pass the

object of C as an argument to the method test of Tester from our driver Example3? Should we recompile the class Tester? Will this class C which Tester did not know earlier cause a runtime error? Will the method foo called in test call the one of the superclass A or the new class C?

You will be amazed to find that we need not recompile Tester. The call from test will behave polymorphically. So, we can keep adding new classes (types) and all the code we have had so far will continue to work with no change. Is this not marvelous? You may want to compare with what would happen in the Shape class if we introduced a new class(type) like Square.

Please note:
Inheritance is not a reuse mechanism. It is a mechanism for maintenance. Adding a new class will not break the existing code. It allows to avoid if statement based on the value of an attribute.

```
//L34/Example3.java
// polymorphism
//       concept of virtual fn
// we shall make a case for inheritance
// We will have a class called Tester which supports a method called test
// which takes a reference to an object of class A and so of any class which extends
```

```
// A.
// foo is an interface of class A
public class Example3
{
	public static void main(String[] args)
	{

// experiment 1
//	we create object of A and B; call test of Tester
//	which in turn calls foo polymorphically
		A x = new A();
		Tester.test(x);
		B y = new B();
		Tester.test(y);
// experiment 2:
//	we will add another class C which extends A.
		C z = new C();
		Tester.test(z);
//	should we recompile Tester as it did not know about C earlier?
//	would this give a compile time error? runtime error?

//  would test of Tester call foo of A? Or foo of C?
//	do we require an if statement to check the object type in test of Tester?
//	This is the beauty of inheritance. Adding a new type or class does not
//	require that we modify the existing code. No if statements required!
	}
}
```

```
//L34/Tester.java
// do not recompile this!
public class Tester
{
	public static void test(A a)
	{
		a.foo();
	}

}

//L34/A.java
public class A
{
	public void foo()
	{
		System.out.println("foo A");
	}
}

//L34/B.java
public class B extends A
{
	public void foo()
	{
		System.out.println("foo B");
	}
}
```

```
//L34/C.java
// this is a new class
public class C extends A
{
	public void foo()
	{
		System.out.println("foo C");
	}
}
```

Back to shape:

We will make a few changes to our earlier implementation of Shape. We will keep the classes Rect and Circle as they were. We observe that the methods read, disp, peri and area have exactly same signatures. If two classes have something in common, we should raise them to the level of a superclass. So, we will make the Shape the superclass and Rect and Circle its subclasses.

The Shape class that we had earlier had embedded Rect and Circle as part of the class. This required selection at each stage to check which of them was actually instantiated. It is not a good design, So we will remove everything of the Shape class but for the method signatures.

We find ourselves in an unusual position in the Shape class. How do we define these methods read, disp, area and peri unless we know which Shape we are talking about?

We cannot provide any meaningful default implementation for these methods. So we make these methods abstract indicating there is no implementation. If any method in a class is abstract, then we should not be allowed to make an instance of the class. If we created one and call the method, the program would crash. So, we are not even allowed to make an instance of this class. The class itself becomes abstract. The language Java expects that we also declare the class abstract.

In this example the client is trying to make an array of Shapes – each referring to one of the subclass objects. Then he wants to populate them by calling read and then he wants to find the total area of all the shapes.

Observe the method area closely. This method uses an enhanced for loop to traverse the Shape array and on each element of the Shape array, invokes the method area. It does not have to check whether the area is a Circle or a Rect. There are no if statements. If another class is added the Shape class hierarchy, this code does not change. That is the beauty of inheritance.

```
public static double area(Shape[] s)
	{
		double total = 0.0;
		for(Shape e : s)
		{
			total += e.area();
		}
		return total;
	}
```

```
//L35/Example1.java
// we will revisit the Shape class we discussed earlier.
// avoid selection based on values of type
// adding a new type will break the existing code
// instead use inheritance
// capture the commonality in superclass
// What is common bet Rect and Circle?
//	method signatures are common.

// let the user make an array of shapes and find the total area
```

```
public class Example1
{
	public static void main(String[] args)
	{
		Shape[] s = new Shape[4];
		read(s);
		System.out.println("total area : " +  area(s));
	}

	// discuss area first; show why we do not require if statement
	public static double area(Shape[] s)
	{
		double total = 0.0;
		for(Shape e : s)
		{
			total += e.area();
		}
		return total;
	}

	public static void read(Shape[] s)
	{
		double total = 0.0;
		for(int i = 0; i < s.length; i++)
		{
			// discuss why ctor cannot work polymorphically
			s[i] = Shape.make(); // why this?
			s[i].read();
		}
	}
}
```

Creation of Objects cannot be polymorphic:

But the read method is not so simple. We can read into the Shape object once we create it. How do we create the Shape object – Rect or Circle? This definitely requires selection. We will have to somehow say whether we want Rect or Circle. So this code cannot behave polymorphically. We therefore will isolate the creation of object which requires selection into a static method. Please check the static method make of the Shape class.

```
//L35/Circle.java
import java.util.*;
// observe the change
public class Circle extends Shape
{
	private double radius;
	public void read()
	{
		Scanner in = new Scanner(System.in);
		radius = in.nextDouble();
	}
	public void disp()
	{
		System.out.println(radius);
	}
	public double area()
	{
		return 3.1416 * radius * radius;
	}
	public double peri()
	{
		return 2 * 3.1416 * radius;
	}
}
```

```
//L35/Rect.java
import java.util.*;

// Observe the change
public class Rect extends Shape
{
	private double length;
	private double breadth;
	public void read()
	{
		Scanner in = new Scanner(System.in);
		length = in.nextDouble();
		breadth = in.nextDouble();
		//in.close();
	}
	public void disp()
	{
		System.out.println(length + ":" + breadth);
	}
	public double area()
	{
		return length * breadth;
	}
	public double peri()
	{
		return 2 * (length + breadth);
	}
}
```

Inheritance, Polymorphism and some finer points:

- Methods should have the exact same signatures for overriding. Otherwise the method of the subclass will not override the method of the superclass. Then call using superclass reference will always call the superclass method.
- You may use an annotation called @Override to check for these sort of errors.
- A class provides three different abstraction to the users.
  - public: These are accessible(interface) to the class developer, derived class developer and the normal client
  - protected : These are accessible(interface) to the class developer and derived class developer. These are not accessible to the normal client.
  - private: These are accessible(interface) to the class developer only.
- If public members are modified, the whole world is affected. If protected members are modified, it affects the class developer and the derived class developer. If the private members are modified, only the class developer is affected.
- Private methods of the superclass are inherited – they do not behave polymorphically.
- We can stop overriding of a method of the superclass by specifying that the method is final.
- Constructors and static methods of a class are not polymorphic.

These experiments are left to you to try out.

//L35/Example2.java

```
// overriding:
//      annotation : @Override
//      stop overriding : final method
//      stop extending : final class
//

// access control:
```

```
//      multiple interface for a class:
//      a) one for the normal client
//      b) one for the derived class developer
//      c) one for the class developer
public class Example2
{
        public static void main(String[] args)
        {
                // introduce calls to check each of these methods
        }
}
class A
{
        public void f1()
        {
                System.out.println("f1 A");
        }

        public void f2(double x)
        {
                System.out.println("f2 A");
        }

        public A f3()
        {
                System.out.println("f3 A"); return this;
        }

        private void f4()
        {
                System.out.println("f4 A");
        }
```

```
	public final void f5()
	{
		System.out.println("f5 A");
	}

	protected void f6()
	{
		System.out.println("f6 A");
	}
}
```

```
class B extends A
{
	// does this override the method of class A
	@Override
	public void f1()
	{
		System.out.println("f1 B");
	}
	// check with or without
	// show what happens if we call the method using a super class reference
	//	referring to the subclass object
	//@Override
	public void f2(int x)
	{
```

```
        System.out.println("f2 B");
    }

    // covariant return type
    @Override
    public B f3()
    {
        System.out.println("f3 B"); return this;
    }
    // does not override the method of the super class
    // is not an interface
    // @Override
    private void f4()
    {
        System.out.println("f4 B");
    }
    // gives an error even without the @Override annotation
    /*
    public final void f5()
    {
        System.out.println("f5 B");
    }
    */
    @Override
    protected void f6()
    {
        System.out.println("f6 B");
    }
}
```

Make your own observations about these 6 methods – f1 to f6.

Some questions for you to think:

// can constructor be polymorphic?

// can static method be polymorphic?

// are calls within a static method polymorphic?

// are calls within a final method polymorphic?

Protected access : an interesting example:

We have a job which has a number of steps to be executed in a particular order. That order cannot be changed. Can we change any of these steps - do that step differently.

In this example, the client makes an object of class A. It then calls the method called caller. The caller in turn calls the methods step1, step2 and step3 in that order.

What happens if the class B extends A and overrides the method step2 and the client calls the caller with an instance of B?

The call in the method caller behaves polymorphically – calls step1 of A, step2 of B and step3 of A. We have changed one of the steps in the job without altering the order of execution.

//L35/Example3.java

```
// Example of protected:
// requirement:
// execute a few steps in order
// user cannot change the order
// any of the steps can be fine tuned
```

```
public class Example3
{
	public static void main(String[] args)
	{
		A x = new A();
		x.caller();
		x = new B();
		x.caller();
	}
}
class A
{
	public void caller()
	{
		step1();
		step2();
		step3();
	}
	protected void step1() { System.out.println("step1"); }
	protected void step2() { System.out.println("step2"); }
	protected void step3() { System.out.println("step3"); }
}

class B extends A
{
	protected void step2() { System.out.println("new step2"); }
}
```

Upcasting and Downcasting:

We have seen a number of examples where we convert an object of subclass to an object of superclass. This is a trivial conversion – it is implicit – is also called upcasting.

Can we convert an object of superclass to an object of subclass?

Can we call a monkey a mammal and then again refer it as a monkey?

That is fine.

Can we call a monkey a mammal and then refer to it later as a donkey? That is not fine.

When we convert from subclass to superclass, we lose the type information. We can gain the information back by using explicit casting – called downcasting. We can also check whether the object we have at runtime belongs to a particular class hierarchy by using an operator instanceof.

```
//L35/Example4.java
public class Example4
{
	public static void main(String[] args)
	{
		A x = new A(); B y = new B();
		check(x); check(y);
		convert(x); convert(y);
	}

	// check the runtime type
	public static void check(A a)
	{
		System.out.println(a instanceof A);
		System.out.println(a instanceof B);
	}
```

```
	// regain the type info
	public static void convert(A a)
	{
		//B b = a; // error
		// downcasting requires an explicit cast
		// B b = (B) a; converts if possible; otherwise a runtime exception
		if(a instanceof B)
		{
			B b = (B) a;
			b.bar();
		}
	}
}

class A
{
	protected void foo() { System.out.println("foo A"); }
}

class B extends A
{
	protected void bar() { System.out.println("bar B"); }
}
```

Some questions for your AATMA TRISHA!

//L36/points.txt

You may want to try:
a) what happens if we call a method in the ctor?
b) what happens if we call a method in a static method of the class?
c) what happens if we call a private method in the ctor?
d) what does final associated with the class mean?
e) can we have multiple inheritance?
f) can an abstract class have method implementations?
g) can an abstract class have fields?
h) can an abstract class have ctors?

i) how does instanceof work if we have a number of classes in linear inheritance?
x) can we implement a class called MyArray which has diffent lowerbound?
   can I extend the Arrays class?
y) can we implement a class called MyString which always stores the strings

   in case insensitive manner? Can I extend the String class?
z) how do we make classes which are
   not instantiable?
   Not inheritable?

Wish you a very happy new year.
All the best for your T2. We will catch up again after T2.