Chapter 8 : Polymorphism

➔ We have seen in last chapter that in inheritance allows many types(inherited from the same base type) to be treated as if they were one type. The polymorphism allows one type to express its distinction from another, similar type, as long as they derived from the same base class. This distinction is expressed through differences in behaviour of the methods that you can call through the base class

➔ We know that in cases where upcasting has been used, java works fine because of late binding. That is compiler doesn't know which piece of code will be first executed during compile time it is only got decided during run time through late binding. This late binding is also called as dynamic binding. Polymorphism is possible because of this.

➔ Following is an example of polymorphism and also shows how late binding works.

```
package ch8Polymorphism;

class Instrument
{
	void play(){System.out.println("instrument play()");}
	String what(){return "instrument" ;}
}
class Wind extends Instrument
{
	void play(){System.out.println("wind play()");}
	String what(){return "wind" ;}
}
class Percussion extends Instrument
{
	void play(){System.out.println("percussion play()");}
	String what(){return "percussion" ;}
}
class Woodwind extends Wind
{
	void play(){System.out.println("woodwind play()");}
	String what(){return "woodwind" ;}
}
class Brass extends Wind
{
	void play(){System.out.println("brass play()");}
	String what(){return "brass" ;}
}

public class MusicPolymorphism
{
	public static void tune(Instrument i)
	{
		i.play();
	}
public static void tuneAll(Instrument[] I)
{
	for(Instrument i : I)
		tune(i);
}
public static void main(String [] args)
{
	Instrument [] inst = {
			new Wind(),
			new Percussion(),
			new Woodwind(),
			new Brass(),
	};
	tuneAll(inst);
}
}
Output :
wind play()
percussion play()
woodwind play()
brass play()
```

➔ **Pitfall : Overriding private method** :

See the programme given below

```
package ch8Polymorphism;

public class PrivateOverride
{
	private void f(){ System.out.println("private f()"); }

	public static void main(String[] args)
	{
		PrivateOverride pv = new Derived();
		pv.f();
	}
}

class Derived extends PrivateOverride
{
	public void f(){System.out.println("public f()");}
}
Output:
private f()
```

In the above example one may expect output as “public f()” . But a private method is automatically final and hidden from derived class. In the above case public method f() is a brand new method, its not even overloaded since base class version of f() is not visible in derived class. **The point is where there is no overriding polymorphism doesn’t work there**. i.e. Virtual Method Invocation(VMI) only works when there is overridding. Example which shows that polymorphism doesnot work with overloaded methods.

```
package ch8Polymorphism;

class NoOverrideNoPoly1
{
	void meth1(String str)
	{
		System.out.println(str);
	}
}
```

```
class NoOverrideNoPoly2 extends NoOverrideNoPoly1
{
	void meth1(String str1, String str2)
	{
		System.out.println(str1);
		System.out.println(str2);
	}
}

class NoOverrideNoPoly3 extends NoOverrideNoPoly1
{
	void meth1(String str1, String str2, String str3)
	{
		System.out.println(str1);
		System.out.println(str2);
		System.out.println(str3);
	}
}

public class NoOverrideNoPoly
{
	public static void main(String[] args)
	{
		NoOverrideNoPoly1 nonp1 = new NoOverrideNoPoly1();
		NoOverrideNoPoly2 nonp2 = new NoOverrideNoPoly2();
		NoOverrideNoPoly3 nonp3 = new NoOverrideNoPoly3();

		nonp1 = nonp2;
		nonp1.meth1("one and two");
		//nonp1.meth1("one","two");
		////nonp2's meth1() is overloaded not overridden hence hidden for nonp1
object.

		nonp1 = nonp3;
		nonp1.meth1("one and three");
		//nonp1.meth1("one","two","three");
		//nonp3's meth1() is overloaded not overridden hence hidden for nonp1
object.
	}
}
Output:
one and two
one and three
```

➔ **<u>Pitfall : Fields and static methods</u> :**

Also polymorphism is applied for only methods and not for fields and static methods. As shown in following example.

```
package ch8Polymorphism;

class FieldBase
{
	public int field = 0;
	public int getField()
	{
		return field;
	}
}

class FieldDerive extends FieldBase
{
	public int field = 1;
	public int getField()
	{
		return field;
	}
	public int getSuperField()
	{
		return super.field;
	}
	}

public class FieldAccessPoly {
	public static void main(String[] args)
	{
		FieldBase fb = new FieldDerive();
		System.out.println("sub.field = "+fb.field+" sub.getfield() = "+
				fb.getField());
		FieldDerive fd = new FieldDerive();
		System.out.println("sub.field = "+fd.field+" sub.getfield() = "+
				fd.getField()+" super.field = "+fd.getSuperField());
	}

}
Output:
sub.field = 0 sub.getfield() = 1
sub.field = 1 sub.getfield() = 1 super.field = 0
```

When a **Sub** object is upcast to a **Super** reference, any field accesses are resolved by the compiler, and are thus not polymorphic. I.e. **things which can be resolved during compile time polymorphism doesn't work there**.

```
package ch8Polymorphism;

class StaticBase
{
	public static String staticGet()
	{
		return "Base staticGet()";
	}
	public String dynamicGet()
	{
		return "Base dynmaicGet()";
	}
}

class StaticDerive extends StaticBase
{
	public static String staticGet()
	{
		return "Derive staticGet()";
	}
	public String dynamicGet()
	{
		return "Derive dynamicGet()";
	}
}

public class StaticMethodPolymorphism
{
	public static void main(String[] args)
	{
		StaticBase sb = new StaticDerive();
		System.out.println(sb.staticGet());
		System.out.println(sb.dynamicGet());
	}
}
Output:
Base staticGet()
Derive dynamicGet()
```

Static methods are associated with the class and not with the individual objects.

➔ **From above we can note that Polymorphism doesn't work with Fields, private methods and static methods**.

➔ How construction takes place with polymorphism. Example given below.

```
package ch8Polymorphism;

class Meal
{
	Meal() { System.out.println("Meal()"); }
}
class Bread
{
	Bread() { System.out.println("Bread()"); }
}
class Cheese
{
	Cheese() { System.out.println("Cheese()"); }
}
class Lettuce
{
	Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal
{
	Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch
{
	PortableLunch() { System.out.println("PortableLunch()");}
}

public class ConstructorPolymorph extends PortableLunch
{
	private Bread b = new Bread();
	private Cheese c = new Cheese();
	private Lettuce l = new Lettuce();
	private Meal ml = new Lunch();
	public ConstructorPolymorph()
	{
		System.out.println("ConstructorPolymorph()");
	}
	public static void main(String[] args)
	{
	new ConstructorPolymorph();
	}
}
Output:
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Meal()
Lunch()
ConstructorPolymorph()
```

➔ **<u>Inheritance and cleanup</u> :**

When one have clean up issue that is if one want to handle clean up thing apart from Garbage collector then one must create a method which will dispose things. Lets say the method name is dispose() and following is a example of how it should work.

```
package ch8Polymorphism;

class Characteristic {
private String s;
Characteristic(String s) {
this.s = s;
System.out.println("Creating Characteristic " + s);
}
protected void dispose() {
System.out.println("disposing Characteristic " + s);
}
}
class Description {
private String s;
Description(String s) {
this.s = s;
System.out.println("Creating Description " + s);
}
protected void dispose() {
System.out.println("disposing Description " + s);
}
}
class LivingCreature {
private Characteristic p =
new Characteristic("is alive");
private Description t =
new Description("Basic Living Creature");
LivingCreature() {
System.out.println("LivingCreature()");
}
protected void dispose() {
```

```java
System.out.println("LivingCreature dispose");
t.dispose();
p.dispose();
}
}
class Animal extends LivingCreature {
private Characteristic p =
new Characteristic("has heart");
private Description t =
new Description("Animal not Vegetable");
Animal() { System.out.println("Animal()"); }
protected void dispose() {
System.out.println("Animal dispose");
t.dispose();
p.dispose();
super.dispose();
}
}
class Amphibian extends Animal {
private Characteristic p =
new Characteristic("can live in water");
private Description t =
new Description("Both water and land");
Amphibian() {
System.out.println("Amphibian()");
}
protected void dispose() {
System.out.println("Amphibian dispose");
t.dispose();
p.dispose();
super.dispose();
}
}
public class FrogDispose extends Amphibian {
private Characteristic p = new Characteristic("Croaks");
private Description t = new Description("Eats Bugs");
public FrogDispose() { System.out.println("Frog()"); }
protected void dispose() {
System.out.println("Frog dispose");
t.dispose();
p.dispose();
super.dispose();
}
public static void main(String[] args) {
FrogDispose frog = new FrogDispose();
System.out.println("Bye!");
frog.dispose();
}
}
```

```
Output:
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristic has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
```

```
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
disposing Description Eats Bugs
disposing Characteristic Croaks
Amphibian dispose
disposing Description Both water and land
disposing Characteristic can live in water
Animal dispose
disposing Description Animal not Vegetable
disposing Characteristic has heart
LivingCreature dispose
disposing Description Basic Living Creature
disposing Characteristic is alive
```

In the above example we can see that in each class in the hierarchy contains a member object of type Characteristics and Description. The order of disposal should be reverse of the order of initialization. For base classes one should perform the derived class clean up first, then base class clean up. From output one can see that all parts of the Frog object are disposed in reverse order of creation.

➔ In some of the cases we need to call a method from a constructor. In case of overridden methods the output may be completely different as shown in the example below.

```java
package ch8Polymorphism;

class ConstructorMethod1
{
	void draw()
	{
		System.out.println("ConstructorMethod1.draw()");
	}
	ConstructorMethod1()
	{
		System.out.println("ConstructorMethod1 const starts");
		draw();
		System.out.println("ConstructorMethod1 const ends");
	}
}

class ConstructorMethod2 extends ConstructorMethod1
{
	int radius = 1;
	ConstructorMethod2(int r)
	{
		radius = r;
		System.out.println("ConstructorMethod2 radius "+radius);
	}
	void draw()
	{
		System.out.println("ConstructorMethod2.draw.radius "+radius);
	}
}
```

```
public class ConstructorMethod
{
      public static void main(String[] args)
      {
            ConstructorMethod2 cs2 = new ConstructorMethod2(5);
      }
}
Output: ConstructorMethod1 const starts
ConstructorMethod2.draw.radius 0
ConstructorMethod1 const ends
ConstructorMethod2 radius 5
```

In the above example we can see that draw() is overridden in the derived class. But when draw() is called in ConstructorMethod1s constructor the derived class draw() method has been called. This is the bug very tough to find. Also We can see the above output because of following order of initialization.

a. The storage allocated for the object is initialized to binary zero before anything else happens.

b. The base-class constructors are called as described previously. At this point, the overridden **draw( )** method is called (yes, *before* the **ConstructorMethod2** constructor is called), which discovers a **radius** value of zero, due to Step 1.

c. Member initializers are called in the order of declaration

d. The body of the derived-class constructor is called.

Because of this its always safe to call only final and private(which are implicitly final) methods of base class in constructors because they cant be overridden.

➔ <u>**Covariant return types**</u> **:**

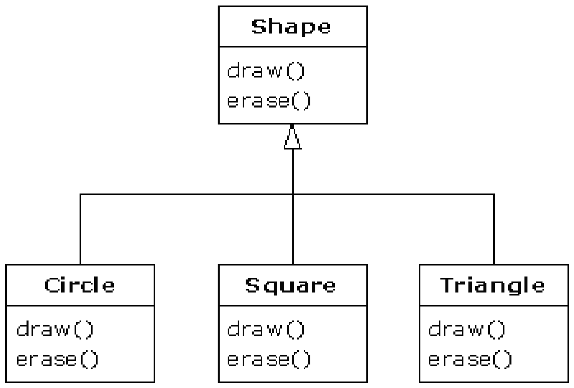
Java SE5 adds the covariant return types, which means that an overridden method in a derived class can return a type which is derived from the type returned by the base class. Example is given below.

```
class Grain {

public String toString() { return "Grain"; }
}
class Wheat extends Grain {
public String toString() { return "Wheat"; }
}
class Mill {
Grain process() { return new Grain(); }
}
class WheatMill extends Mill {
Wheat process() { return new Wheat(); }
}
public class CovariantReturn {
public static void main(String[] args) {
Mill m = new Mill();
Grain g = m.process();
System.out.println(g);
m = new WheatMill();
g = m.process();
System.out.println(g);
}
} /* Output:
Grain
Wheat
*///:~
```

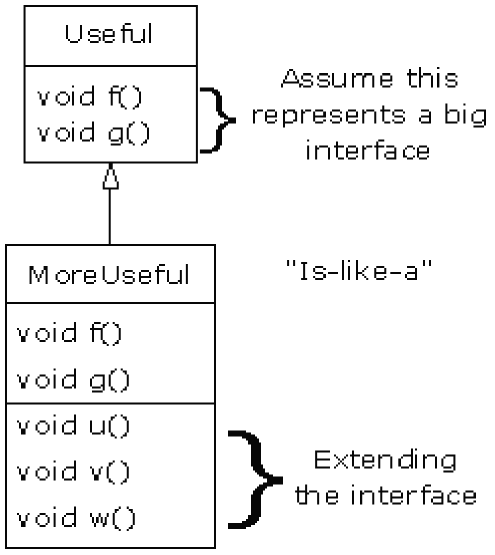
Earlier java versions would have forced overridden method process() to return 'Grain' and not wheat.

➔ When inheritance is used and if its pure inheritance(is-a relationship) that is both Base and Derived class have same interfaces then its good. Pure inheritance diagram is shown below.



With pure inheritance base class can receive any message that one can send to derive class(through upcasting, polymorphism) because both have the same interfaces.

But when tries to add more interfaces to the derived class as show below.



Then the extended part of the interface in derived class is not available from the base class, so once you upcast you cant call the add interfaces. A program is shown below.

```java
package ch8Polymorphism;

class ExtendedMethods1
{
	int a;
	void method1()
	{
		System.out.println("Base method1");
	}
	void method2()
	{
		System.out.println("Base method2");
	}
}

class ExtendedMethods2 extends ExtendedMethods1
{
	void method1()
	{
		System.out.println("Derived method1");
	}
	void method2()
	{
		System.out.println("Derived method2");
	}
	void method3()
	{
		System.out.println("Derived method3");
	}
}

public class ExtendedMethods {

	public static void main(String[] args)
	{
		ExtendedMethods1 em1 = new ExtendedMethods1();
		ExtendedMethods2 em2 = new ExtendedMethods2();
		em1.method1();
		em1.method2();

		em1=em2;

		em1.method1();
		em1.method2();
		//em1.method3();//cant call method3 cos its undefined in base class
	}
}
```

```
Output:
Base method1
Base method2
Derived method1
Derived method2
```

Above we can see em1.method3() has been commented out because method3() is not present in the base class hence compiler throws error.

➔ To solve the above problem we have a option of 'Downcasting'. As we know upcasting is always safe as we are sure that there cant be bigger interfaces compare to base class but downcasting is not that safe as you don't know that a shape (for example) is actually a circle. It can be triangle or square or some other type.

To be ensure that downcast is safe java performs checks on every cast. During runtime downcast will be checked and if there is a type mismatch then java throws an **ClassCastException.** This act of checking type at run type is called 'runtime type identification' (**RTTI**). Follwing program demonstrate when java does that.

➔ Example : 1

```
package ch8Polymorphism;

class ExtndMethDowncast1
{
	void emdMethod1()
	{
		System.out.println("base method 1");
	}
	void emdMethod2()
	{
		System.out.println("base method 2");
	}
}

class ExtndMethDowncast2 extends ExtndMethDowncast1
{
	void emdMethod1()
	{
		System.out.println("derive method 1");
	}
	void emdMethod2()
	{
		System.out.println("derive method 2");
	}
	void emdMethod3()
	{
		System.out.println("derive method 3");
	}
}

public class ExtndMethDowncast {

	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		ExtndMethDowncast1 emd1 = new ExtndMethDowncast1();
		ExtndMethDowncast2 emd2 = new ExtndMethDowncast2();
		emd1 = emd2;
		emd1.emdMethod1();
		emd1.emdMethod2();
		((ExtndMethDowncast2)emd1).emdMethod3();
```

```
        }
}
Output :
derive method 1
derive method 2
derive method 3
```

➔ Example : 2

```
package ch8Polymorphism;

class ExtndMethodDowncasting1
{
        int a;
        void method1()
        {
                System.out.println("Base method1");
        }
        void method2()
        {
                System.out.println("Base method2");
        }
}

class ExtndMethodDowncasting2 extends ExtndMethodDowncasting1
{
        void method1()
        {
                System.out.println("Derived method1");
        }
        void method2()
        {
                System.out.println("Derived method2");
        }
        void method3()
        {
                System.out.println("Derived method3");
        }
}

public class ExtndMethodDowncasting {

        public static void main(String[] args)
        {
                ExtndMethodDowncasting1 [] em = { new ExtndMethodDowncasting1(), new
                            ExtndMethodDowncasting2() };
                em[0].method1();
                em[1].method2();
                ((ExtndMethodDowncasting2)em[1]).method3();//RTTI
                //((ExtndMethodDowncasting2)em[0]).method3();//Exception
                }
}
Output:
Base method1
Derived method2
Derived method3
```

Note that class cast has been used here i.e. ExtndMethodDowncasting2.

➔ Example 3

```
package ch5InitializationNCleanUp;

class UniqueMethBaseCls
{
	void method1()
	{
		System.out.println("BaseMethod1()");
	}
	void method2()
	{
		System.out.println("BaseMethod2()");
	}
	void method3()
	{
		System.out.println("BaseMethod3()");
	}
}

class UniqueMethDeriveCls extends UniqueMethBaseCls
{
	void method1()
	{
		System.out.println("DeriveMethod1()");
	}
	void method2()
	{
		System.out.println("DeriveMethod2()");
	}
	void method4()
	{
		System.out.println("DeriveMethod4()");
	}
}

public class UniqueMethOfBaseNDeriveCls {

	public static void main(String[] args)
	{
		UniqueMethBaseCls umb = new UniqueMethBaseCls();
		UniqueMethDeriveCls umd = new UniqueMethDeriveCls();

		umb=umd;
		umb.method1();
		umb.method2();
		umb.method3();
		((UniqueMethDeriveCls)umd).method4();
	}
}
Output:
DeriveMethod1()
DeriveMethod2()
BaseMethod3()
DeriveMethod4()
```

Note : After assigning derive class reference to base class we can see only overridden methods of derive class are called. Non-overridden methods from base class will be called and unique methods of derive class only called through class cast.