Object-Oriented Programming

Chapter Four: Polymorphism and Packages

Outline

- polymorphism Overview
- Relationships among Objects in an inheritance hierarchy
- Implementation of Polymorphism
- Method Binding
- Upcasting and Downcasting
- Polymorphism using Abstract Class and Methods
- Packages

Polymorphism Overview

- Polymorphism is one of the principles in Object Oriented Programming paradigm
- The term polymorphism means "a method the same as another in spelling but with different behavior."
- Polymorphism is the ability of objects to act depending on the run time type
- Polymorphism allows programmers to send the same message to objects from different classes. i.e., in its simplest from, polymorphism allows a single variable to refer to objects from different classes.
- Polymorphism enables us to "program in the general" rather than "program in the specific"

Relationships among Objects in an inheritance hierarchy

- An object of a subclass can be treated as object of its superclass.
- This enables various interesting manipulations.
- For example, a program can create an array of superclass reference that refers to objects of many subclass types.
- This is allowed despite the fact that the subclass objects are of different types, because each subclass object is an object of its subclasses.

Implementation of Polymorphism

- Method Overloading (Early Binding)
  - Methods with same name but whose signature (the number and type of formal parameters) are used to handle
- Method Overriding (Late Binding)
  - Methods of a subclass override the methods of a super class in a given inheritance hierarchy
  - Methods of a subclass implement the abstract methods of an abstract class
  - Methods of a concrete class implement the methods of an interface

Method Binding

- In general, connecting a method call to a method body is called binding
- When binding is performed before the program is run, it's called early binding
- When the binding occurs at run-time based on the type of object, it's called late binding. It is also called dynamic binding or run-time binding
- When late binding is implemented, there must be some mechanism to determine the type of the object at run-time and to call the appropriate method
  - That is, the compiler still doesn't know the object type, but the method-call mechanism finds out and calls the correct method body

Method Binding Cont.

- All method binding in Java uses late binding unless a method has been declared **final**

- This means that ordinarily you don't need to make any decisions about whether late binding will occur- it happens automatically.

- But if a method is declared as **final**, it prevents anyone from overriding that method. In other words, it effectively "turns off" dynamic binding, or rather it tells the compiler that dynamic binding isn't necessary

Method Binding Cont.

- **Example1: Polymorphism using Overloaded Methods**

```
1   package overloaded;
2    class DemoClass
3   {
4   public int add(int x, int y)
5   {
6   return x + y;
7   }// end add(int, int)
8   public int add(int x, int y, int z)
9   {
10  return x + y + z;
11  }// end add(int, int, int)
12  public int add(double pi, int x)
13  {
14  return (int)pi + x;
15  }// end add(double, int)
16  }// end DemoClass
17  public class OverLoaded {
18      public static void main(String[] args) {
19    DemoClass obj = new DemoClass();
20  System.out.println(obj.add(2,5)); // int, int
21  System.out.println(obj.add(2, 5, 9)); // int, int, int
22  System.out.println(obj.add(3.14159, 10)); //double, int
23      }
24  }
```

```
Output - OverLoaded (run)
run:
7
16
13
```

- This form of method binding is called early binding
- Here the method add() has more than one form

Method Binding Cont.

Example 2: Polymorphism using overriding methods in inheritance hierarchy

```
1   package animalpolymorphism;
2   public class AnimalPolymorphism {
3       public static void main(String[] args) {
4           AnimalPolymorphism  animal = new AnimalPolymorphism ();
5           Dog dog = new Dog();
6                   Cat cat = new Cat();
7                   animal.print();
8                   dog.print();
9                   cat.print();
10      }
11      void print() {
12          System.out.println("Superclass Animal");
13      }
14  }
15   class Dog extends AnimalPolymorphism {
16       @Override
17      void print() {
18          System.out.println("Subclass Dog");
19      }
20  }
21  class Cat extends AnimalPolymorphism{
22      @Override
23          void print(){
24          System.out.println("Subclass Cat");
25          }
26  }
```

```
Output - AnimalPolymorphism (run)
run:
Superclass Animal
Subclass Dog
Subclass Cat
```

- Method print in Dog and Method print in Cat override method print in Animal

Upcasting and Downcasting

- A process of converting one data type to another is known as Typecasting .
- Just like the data types, the objects can also be typecasted.
- In objects, there are only two types of objects, i.e. parent object and child object.
- typecasting of objects basically means that one type of object (i.e.) child or parent to another.
- There are two types of typecasting.
  - Upcasting and
  - Downcasting

Upcasting and Downcasting Cont.

1. Upcasting: Upcasting is the typecasting of a child object to a parent object.

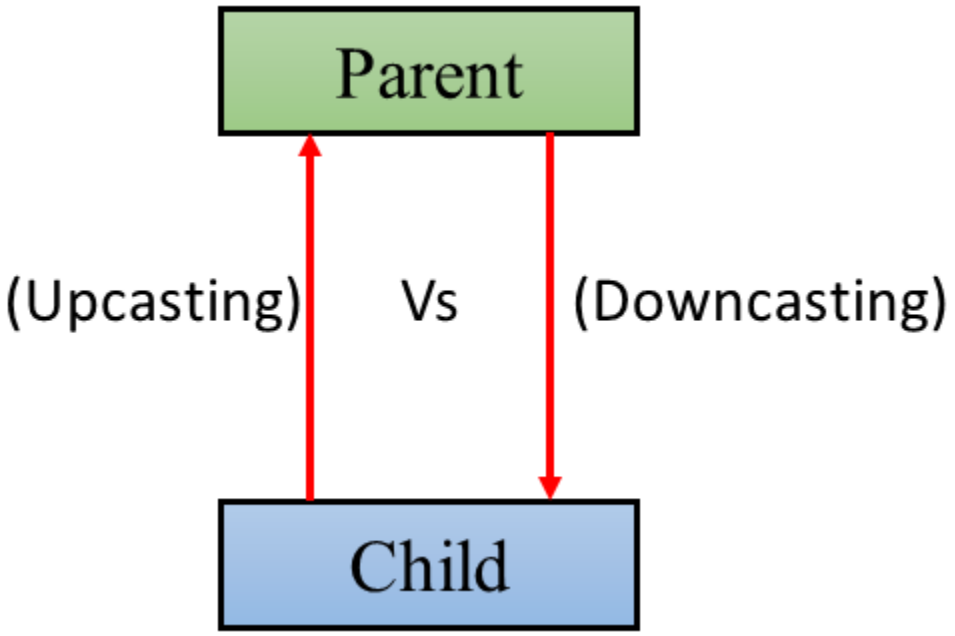
- Upcasting can be done implicitly.
- Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature.
- Instead of all the members, we can access some specified members of the child class.
- For instance, we can access the overridden methods.

2. Downcasting: means the typecasting of a parent object to a child object.

- Downcasting cannot be implicit.

Upcasting and Downcasting Cont.

- The following image illustrates the concept of upcasting and downcasting:



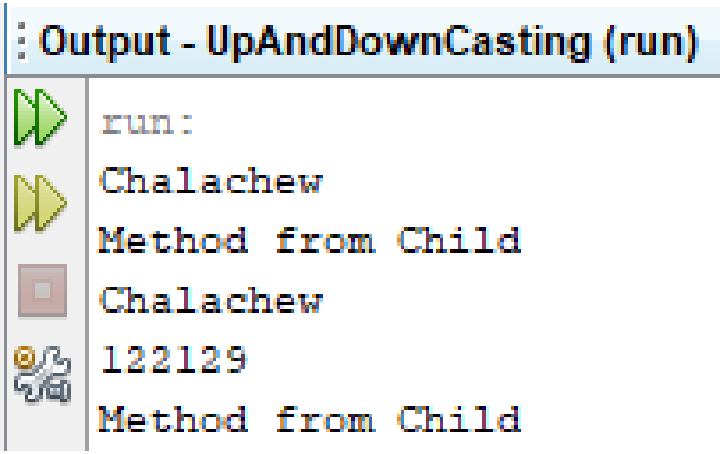
Upcasting and Downcasting Cont.

- Upcasting Example

```
1   package upanddowncasting;
2   // Parent class
    class Parent {
4       String name;
        void method()
6       {
7           System.out.println("Method from Parent");
8       }
9   }
10  // Child class
11  class Child extends Parent {
12      int id;
        @Override void method()
14      {
15          System.out.println("Method from Child");
16      }
17  }
```

Upcasting and Downcasting Cont.

```
18  public class UpAndDownCasting {
19      public static void main(String[] args) {
20          // Upcasting
21          Parent p = new Child();
22          p.name = "Chalachew";
23          //Printing the parentclass name
24          System.out.println(p.name);
25          //parent class method is overridden method hence this will be executed
26          p.method();
27          // Trying to Downcasting Implicitly
28          // Child c = new Parent(); - > compile time error
29          // Downcasting Explicitly
30          Child c = (Child)p;
31          c.id = 122129;
32          System.out.println(c.name);
33          System.out.println(c.id);
34          c.method();
35      }
36  }
```

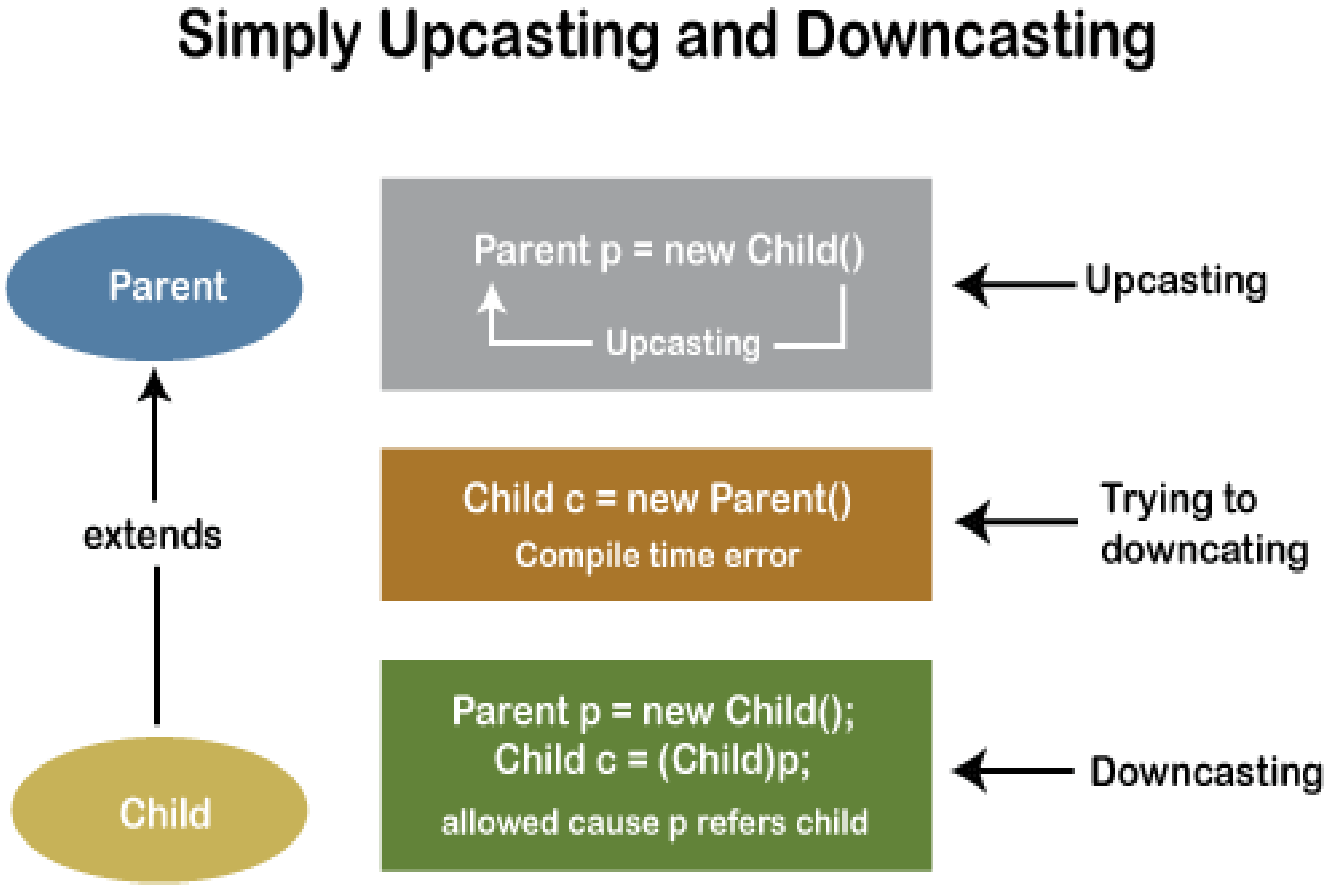


Upcasting and Downcasting Cont.

- Using a superclass reference variable, you can call methods of the subclass that are inherited from a superclass or methods that are overridden in the subclass
- Using a superclass reference you can not call methods that are found only in the subclass. Example:

```
Parent p=new Child();
    p.name = "Chalachew"; //correct
System.out.println(p.name);//correct
      p.method();
Child c = new Parent(); - > compile time error
   Child c = (Child)p; // Downcasting Explicitly
c.id = 122129;//ok
System.out.println(c.name);
System.out.println(c.id);//correct
c.method();
```

Upcasting and Downcasting Cont.



Difference between Upcasting and Downcasting

- These are the following differences between Upcasting and Downcasting:

S.No	Upcasting	Downcasting
1.	A child object is typecasted to a parent object.	The reference of the parent class object is passed to the child class.
2.	We can perform Upcasting implicitly or explicitly.	Implicitly Downcasting is not possible.
3.	In the child class, we can access the methods and variables of the parent class.	The methods and variables of both the classes(parent and child) can be accessed.
4.	We can access some specified methods of the child class.	All the methods and variables of both classes can be accessed by performing downcasting.
5.	Parent p = new Child()	Parent p = new Child() Child c = (Child)p;

Example: Polymorphism using Abstract Class and Methods

```
1   package testshape;
    abstract class Shape {
    public  abstract double circumference();
    public abstract double area();
5   }
6   public class TestShape {
7       public static void main(String[] args) {
8   Shape[] shapes = new Shape[3];
9   shapes[0] = new Circle(2.0);
10  shapes[1] = new Rectangle(1.0, 3.0);
11  shapes[2]=new Rectangle(4.0, 2.0);
12  double total_area = 0;
13  for(int i = 0; i < shapes.length; i++){
14      total_area += shapes[i].area();
15  System.out.println(total_area); }
16      }
17  }
18  class Rectangle extends Shape {
19  protected double w, h;
20  public Rectangle(double w, double h) {
21  this.w = w;   this.h = h;
22  }
23  public double getWidth() {
24      return w;
25  }
```

```
26  public double getHeight() {
27      return h;
28  }
        @Override
30      public double area() {
31          return w*h;
32      }
        @Override
34      public double circumference() {
35   return 2*(w + h);
36      }
37  }
38  class Circle extends Shape {
39  public static final double PI = Math.PI;
40  protected double r;
41  public Circle(double r) {
42      this.r = r;
43  }
44  public double getRadius() {
45      return r;
46  }
        @Override
48      public double area() {
49          return PI*r*r;
50      }
```

Example: Polymorphism using Abstract Class and Methods Cont..

```
52      @Override
        public double circumference()
53      {
54          return 2*PI*r;
55      }
56  }
```

```
Output - TestShape (run)
run:
12.566370614359172
15.566370614359172
23.566370614359172
```

- What is the circumference of both rectangle and circle?

Example polymorphism using overriding method

```
1   package pointtest;
    class Point{
3   private int x;
4   private int y;
5   public Point (int xValue,int yValue)
6       {     x=xValue;
7             y=yValue;
8       }
9   public int getX() {  return x;  }
10  public int getY(){ return y;    }
        @Override
12      public String   toString(){
13          return "+getX()+"+"+getY()+";
14  }
15  public String display(Point p){
16      return    p.toString();   }
17  }//end class
18  public class PointTest {
19      public static void main(String[] args) {
20  Point  dot = new Point(30,50);
21  Circle circle = new Circle(120,89,2.7);
22  System.out.println(dot.toString());
23  System.out.println (circle.toString());
24  Point ptRef = circle;
25  System.out.println(ptRef.toString());
26      }
27  }
```

```
28  class Circle extends Point{
29  private double radius;
30  public Circle(int xValue, int yValue, double rValue)     {
31        super(xValue,yValue);
          setRad(rValue);
33  }
34  public void setRad(double rValue){
35   radius = (rValue < 0.0?0.0:rValue);
36  }
37  public double getRadius()
38     {     return radius;
39     }
        @Override
41   public String toString(){
42   return "center = "+super.toString()+ "; Radius="+getRadius();
43  }
44  }//end class
```

```
Output - PointTest (run)
run:
+getX()++getY()+
center = +getX()++getY()+; Radius=2.7
center = +getX()++getY()+; Radius=2.7
```

Example polymorphism using overriding method

- In the previous program, the statement Point ptRef = circle; assigns the subclass object circle to superclass-type reference variable- ptRef.
- A superclass-type variable that contains a reference to a subclass object calls the subclass method; hence, ptRef.toString() actually calls class Circle's toString method
- The Java compiler allows this "crossover" because an object of a subclass is an object of its superclass
- Note:- a superclass object is not an object of any of its subclasses. The "is-a" relationship applies only from a subclass to its direct and indirect superclasses

Benefits of Using Polymorphism

- **Simplicity**

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

- **Extensibility**

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

Packages

- Packages are containers for classes that are used to keep the class name space compartmentalized
- A package contains a set of related classes.
- To create a package, simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored.
- Syntax:

  package packageName;
- A package name consists of one or more identifiers separated by periods.
- There is a special package called the **default package**, which has no name. If you **did not include any package statement** at the top of your source file, **its classes** are placed in the default package.

Packages Cont.

- If you want to use a class from a package, you can refer to it by its full name(package name plus class name). For example, java.util.Scanner refers to the Scanner class in the java.util package:
- The 'import' directive lets you refer to a class of a package by its class name, without the package prefix.
- You cannot rename a package without renaming the directory in which the classes are stored.
- A package name corresponds to the directory name where the file is located.
- Remember that case is significant, and the directory name must match the package name exactly.

Packages Cont.

- Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.
- You can import all classes of a package with an import statement that ends in .*. For example, you can use the statement
- import java.util.*;
- Default package:-import java.lang.*;
- Package Names
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.
- The syntax for a multileveled package statement is:
- package pkg1[.pkg2[.pkg3]];

Packages Cont.

Importing Classes and Packages

- Specify the class you want to import:

```
import java.util.Date;
```

- You can import multiple classes:

```
import java.util.ArrayList;
```

- You can also import whole packages:

```
import java.util.*;
```

- Default imported package:

```
import java.lang.*;
```

Example: Package

```
package examples;

class Person {
    String name;
  // add a field to store a birthday
  // write a method to set the
    birthday
  // write a method to get the
    birthday
}
```

```
//Using java.util.Date

package examples;
class Person {
String name;
java.util.Date birthday;
void setBirthday(java.util.Date d) {
 birthday = d;
  }
java.util.Date getBirthday() {
 return birthday;
  }
}
```

Example: Package Cont.

```
//Importing java.util.Date
package examples;
import java.util.Date;
class Person {
      String name;
      Date birthday;
      void setBirthday(Date d) {
      birthday = d;}
      Date getBirthday() {
       return birthday;
   }
}
```

```
//Importing java.util.ArrayList
package examples;
import java.util.Date;
import java.util.ArrayList;
class Person {
      String name;
      Date birthday;
      ArrayList friends;
      void setBirthday(Date d) { birthday = d; }
      // . . .
}
```

```
//Importing java.util.*
  package examples;
  import java.util.*;
  class Person {
        String name;
        Date birthday;
        ArrayList friends;
    void setBirthday(Date d) { birthday = d;}
        // . . .
  }
```

Packages and Visibility

- The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.
- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.
- This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.