



Chapter 12: Inheritance

Objectives

Students should

- Understand the concept and role of inheritance.
- Be able to design appropriate class inheritance hierarchies.
- Be able to make use of inheritance to create new Java classes.
- Understand the mechanism involved in instance creation of a class inherited from another class.
- Understand the mechanism involved in method invocation from a class inherited from another class.

Inheritance: Creating Subclasses from Superclasses

Inheritance is an ability to derive a new class from an existing class. That new class is said to a *subclass*, or *derived class*, of the class it is derived from, which is called *superclass*, or *base class*. A subclass can be thought of as an extension of its superclass. It inherits all attributes and behaviors from its superclass. However, more attributes and behaviors can be added to existing ones of its superclass.

Let's look at a simple example of how we create subclasses. Although it is not going to be very useful in any real programs, this should serve as the first example of class inheritance that gives you a first look at how corresponding class definition can be written. Suppose that we have a class called *L12A* whose definition is:

```
public class L12A
{
    public int x;
    public double d;
    public double f(){
        return x*d;
    }
}
```

Now, let's say that we would like to create a new class *L12B* that inherits all attributes and behaviors from *L12A* with no extra attributes or behaviors added. Writing the class definition of *L12B* does not involve repeating the code in *L12A*. We use the keyword "*extends*" to let Java know that *L12B* inherits from *L12A*. With no additional attributes or behaviors, the class definition of *L12B* can be simply written as:

```
public class L12B extends L12A
{
}
```

This very short code segment is already a valid class definition of *L12B* that we want. We could now write a program that makes use of *L12B*. Consider the following program and its output.

```
public class InheritanceDemo0
{
    public static void main(String[] args)
    {
        L12B b = new L12B();
        b.x = 2;
        b.d = 1.5;
        System.out.println("b.f() = "+b.f());
    }
}
```

1
2
3
4
5
6
7
8
9

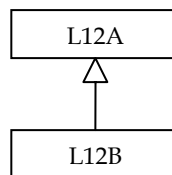


```
C:\WINDOWS\system32\cmd.exe

C:\>javac InheritanceDemo0.java
C:\>java InheritanceDemo0
b.f() = 3.0
C:\>
```

We can see from the above program that an object of the class *L12B* contains the instance variables *x* and *d*, as well as *f()*, without having to explicitly write these attributes and behaviors in the class definition of *L12B*. *L12B* is called a subclass of *L12A*. On the other hand, *L12A* is called the superclass of *L12B*.

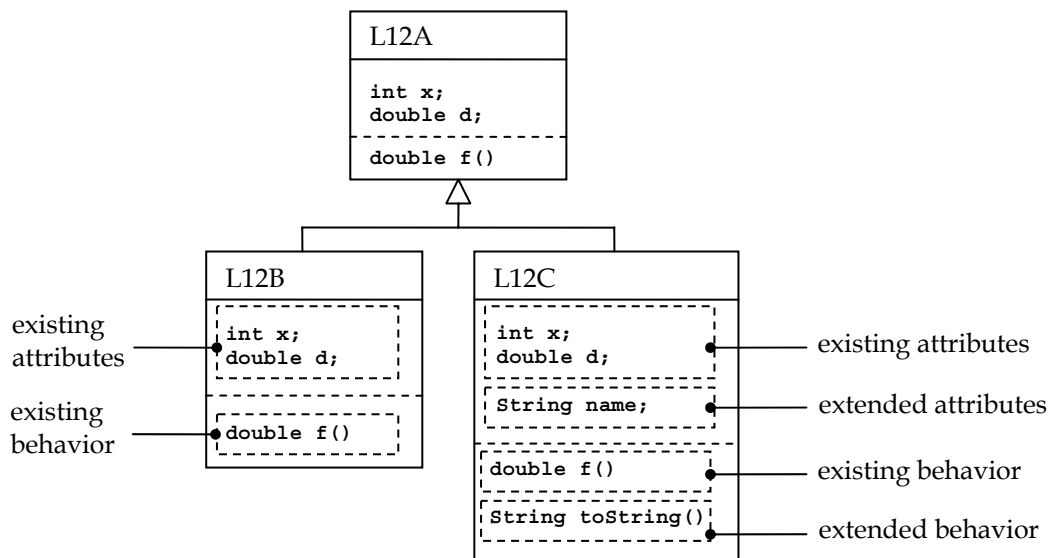
A class inheritance diagram can be used to show the relationship among classes. To show that *L12B* is extended or inherited from *L12A*, a diagram in the figure below can be used.



However, the real benefit of inheritance is not creating a new class that behaves exactly the same way as its superclass but creating a new class contains all attributes and behaviors of its superclass together with some additional attributes and behaviors. For example, if we would like to create a new class called *L12C* which has all the attributes and behaviors of *L12A* but with a name for each instance of *L12C*, we could extends *L12A* with an additional instance variable of type *String* as well as some appropriate methods. Such a class could be written as:

```
public class L12C extends L12A
{
    public String name;
    public String toString(){
        return name+" "+x+" "+d;
    }
}
```

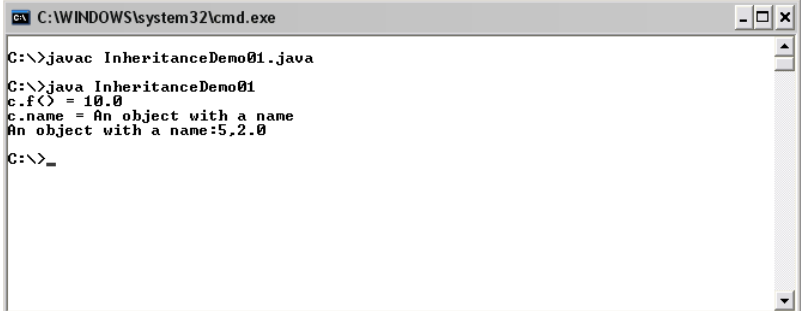
From the class definitions of the three classes: *L12A*, *L12B*, and *L12C*, their structures can be depicted as the following figure.





Now consider the following program and observe its output.

```
public class InheritanceDemo01
{
    public static void main(String[] args)
    {
        L12C c = new L12C();
        c.x = 5;
        c.d = 2.0;
        c.name = "An object with a name";
        System.out.println("c.f() = "+c.f());
        System.out.println("c.name = "+c.name);
        System.out.println(c);
    }
}
```



On line 5, a variable `c` is created and is referred to a new object of `L12C`. The statements line 6 and line 7 assign values to instance variables which are defined in the class definition of `L12A`, the superclass of `L12C`. On line 8, the instance variable `name` which is a part extended in `L12C` from `L12A` is assigned with a `String`. On line 9, the existing method `f()` in the superclass is called, while on line 10, the instance variable `name` is used, and on line 11, Java automatically calls `toString()` defined in `L12C` in response to `System.out.println(c);`.

Note that multiple-inheritance is not allowed. A subclass can only extend from a single superclass, while a superclass can have more than one subclass inherited from the class.

A More Realistic Example

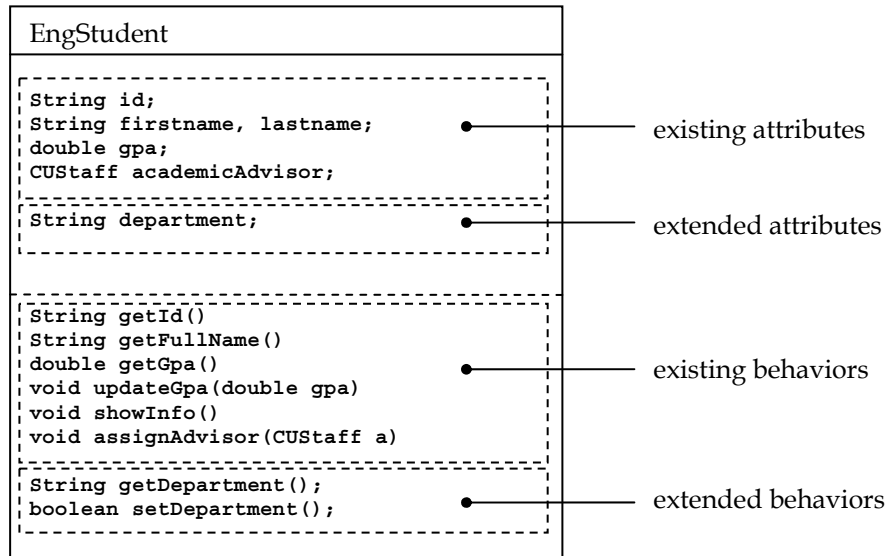
Suppose we have a class called `CUStudent` representing a student in Chulalongkorn University. The class might have the structure as shown here.

CUStudent
<pre>String id; String firstname, lastname; double gpa; CUStaff academicAdvisor;</pre>
<pre>String getId() String getFullName() double getGpa() void updateGpa(double gpa) void showInfo() void assignAdvisor(CUStaff a)</pre>

Observing the structure, we can see that the class is designed so that it contains attributes and behaviors that can be applied to all students in the university regardless of which departments or faculties they enroll to.

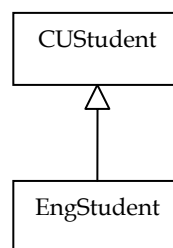


Now, let's consider creating a new class, called *EngStudent*, for representing students in Chulalongkorn University who are in the faculty of Engineering. An engineering student is still a student in Chulalongkorn University but probably with additional attributes and behaviors. Here, we suppose that every student in the faculty of Engineering must have an attribute showing which departments they are in. Therefore, this class should have all attributes and behaviors that *CUStudent* has, together with additional attributes and behaviors specific to *EngStudent*. We can create the class *EngStudent* by deriving or extending from *CUStudent*. The desired structure might be like:



By its nature, an object of the class *EngStudent* can be thought of as a special case of the class *CUStudent*. To define *EngStudent*, we need to create its class definition just like what we have to when we would like to create a new class. However, we do not have to rewrite everything in the class definition of *EngStudent*. We extend the class definition of *CUStudent* to obtain the class definition of *EngStudent*.

The class inheritance diagram for *CUStudent* and *EngStudent* can be shown in the following figure.



Suppose that *CUStudent* is defined as:

```
public class CUStudent
{
    private String id;
    private String firstname, lastname;
    private double gpa;
    private CUStaff academicAdvisor;

    public CUStudent(String id, String firstname, String lastname)
    {
        this.id = id;
        this.firstname = firstname;
        this.lastname = lastname;
        gpa = 0.0;
        academicAdvisor = null;
    }
}
```



```

    }
    public String getId(){
        return id;
    }
    public String getFullName(){
        return firstname+" "+lastname;
    }
    public double getGpa(){
        return gpa;
    }
    public void updateGpa(double gpa){
        this.gpa = gpa;
    }
    public void showInfo(){
        System.out.println();
        System.out.println("ID:"+getId());
        System.out.println(getFullName());
        System.out.println("Advisor:"+getAdvisor());
    }
    public void assignAdvisor(CUStaff a){
        academicAdvisor = a;
    }
    public CUStaff getAdvisor(){
        return academicAdvisor;
    }
    public String toString(){
        return "CUStudent:"+getFullName();
    }
}

```

Note that private modifiers with instance variables on line 3 to line 6 are used so that the access to these variables is limited to this class only. It means that these variables can be used directly inside the class definition of *CUStudent* only.

To obtain the desired *EngStudent*, we can create its class definition as the following.

```

public class EngStudent extends CUStudent
{
    private department;

    public EngStudent(String id,String firstname,String lastname){
        super(id,firstname,lastname);
        department = null;
    }

    public String getDepartment(){
        return department;
    }
    public boolean setDepartment(String dept){
        if(validDepartment(dept)){
            department = dept;
            return true;
        }else{
            return false;
        }
    }
    private boolean validDepartment(String dept){
        // This method returns true if the input String
        // is a valid department in Engineering school.
        ...
        // Implementation is omitted.
    }
}

```

Observing the definition of *EngStudent*, we can see that only extended attributes and extended behaviors, including the *EngStudent* constructor, are needed to be defined. We use the keyword “*extends*” followed by the name of the desired base class, which in this case is *CUStudent*, to let Java know that this class is derived from the specified class. Therefore, *EngStudent* contains every attributes and behaviors that its base class has.



Let's look at the following example to see *EngStudent* in action.

```
public class InheritanceDemo1
{
    public static void main(String[] args)
    {
        EngStudent a
        = new EngStudent("4971234521", "Alan", "Smith");
        CUStaff b
        = new CUStaff("3600", "Alex", "Ferguson");
        a.assignAdvisor(b);
        a.showInfo();
        if(a.setDepartment("ICE"))
            System.out.println(a.getDepartment());
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac InheritanceDemo1.java
C:\>java InheritanceDemo1
ID:4971234521
Alan Smith
Advisor:CUStaff:Alex Ferguson
ICE
C:\>
```

On line 5, *a* is a variable referring to a new *EngStudent* object which is created by the new statement written on line 6. This statement causes the invocation of the constructor of *EngStudent*, which is defined on line 5 of the definition of *EngStudent*. The purpose of the constructor is to initialize appropriate values to all of the attributes which, in this case, consists of attributes existing in the superclass and extended attributes additionally defined in the class definition of the subclass. To initialize attributes in the superclass, we usually call a constructor of the superclass. The *super()* is used for referring to the corresponding constructor of the superclass. Therefore, *super(id,firstname,lastname);* on line 6 of the *EngStudent* class definition invokes the constructor with three *String* parameters defined on line 8 in *CUStudent* class definition. Initialization of extended attributes for the subclass is done via typical assignment statements, such as *department = null;* in this example.

Next in the program, an object of class *CUStaff*, whose definition is omitted here, is created and referred to by a variable *b*. Then, on line 9, *assignAdvisor()* is called from an object of class *EngStudent*. We can see that *assignAdvisor()* is not defined in *EngStudent*. However, this method is defined in its superclass. Therefore, it is a behavior existing in *CUStudent* and statements listed in the corresponding method (on line 34 in *CUStudent*) are performed.

The same applies to *showInfo()*. It is defined in *CUStudent* not in *EngStudent*. Always keep in mind that an object of the class *EngStudent* has attributes and behaviors that its superclass has even though they are not listed explicitly in the definition of the subclass. Thus, methods such as *assignAdvisor()* and *showInfo()* can be called from an object of the class *EngStudent* without any problems.

On line 11 and line 12 in the program, *setDepartment()* and *getDepartment()* are called from *a*. These two methods are the ones extended from the superclass. They only exist for the subclass only. Although we can access methods belonging to superclasses from objects of the subclasses, we cannot access the method belonging to subclasses through objects of superclasses. For example, if *x* is a variable referring to a *CUStudent* object, the program



cannot be compiled successfully when `x.setDepartment(...)` or `x.getDepartment()` is used. See the following example and observed the compilation result.

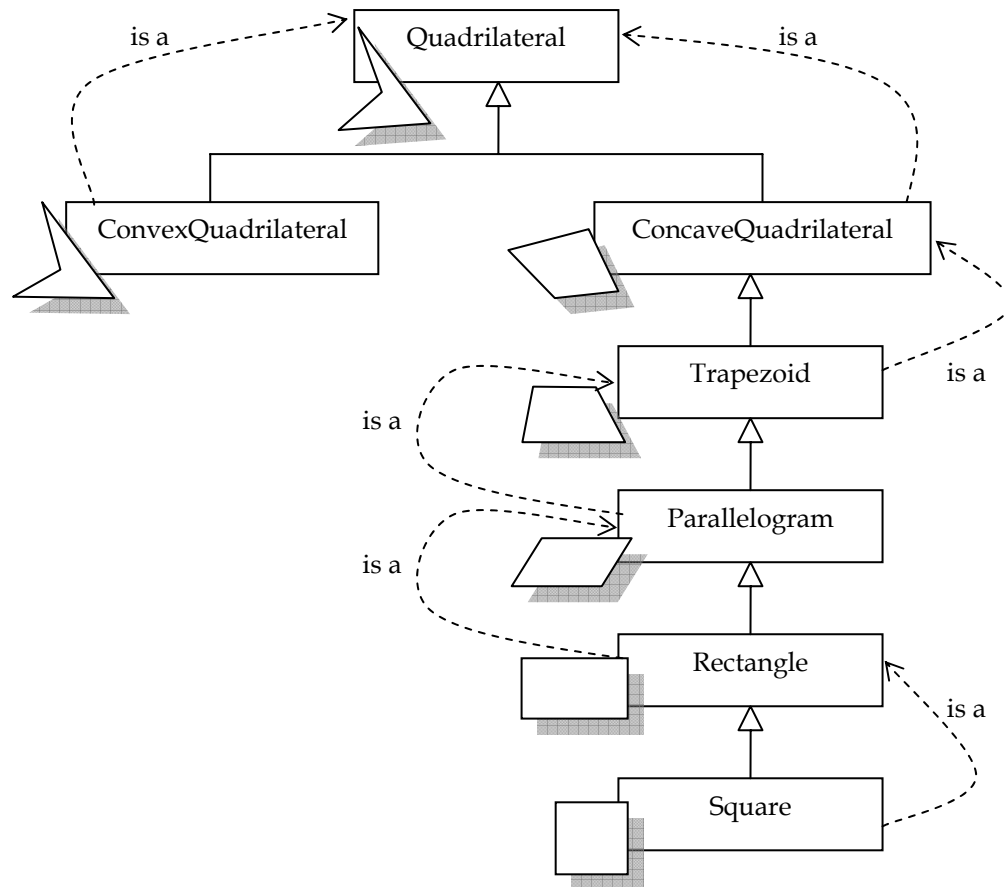
```
public class InheritanceDemo2
{
    public static void main(String[] args)
    {
        CUStudent x
        = new CUStudent("4971234521", "Alan", "Smith");
        CUStaff b
        = new CUStaff("3600", "Alex", "Ferguson");
        x.assignAdvisor(b);
        x.showInfo();
        if(x.setDepartment("ICE"))
            System.out.println(x.getDepartment());
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac InheritanceDemo2.java
InheritanceDemo2.java:11: cannot find symbol
symbol : method setDepartment<java.lang.String>
location: class CUStudent
    if(x.setDepartment("ICE"))
        ^
InheritanceDemo2.java:12: cannot find symbol
symbol : method getDepartment<>
location: class CUStudent
        System.out.println(x.getDepartment());
                             ^
2 errors
C:\>
```

Designing Class Inheritance Hierarchy

Using inheritance effectively in creating new classes does not involve only the Java syntax used in defining the new class but also organizing classes in a hierarchical manner. A good class hierarchy helps us understand the relationship among classes. Superclasses are always more general than subclasses since a subclass possesses everything that its superclass has while it can also possess additional attributes and behaviors. There is an *is-a relationship* between a subclass and its superclass. We can say that an object of a subclass is also naturally an object of its superclass. In the previous example, *EngStudent* is a subclass of *CUStudent*. That means an engineering student “*is a*” student in Chulalongkorn University.

A subclass can be inherited further by other classes. This makes a hierarchy of classes. The figure below shows an example of a hierarchy among some quadrilaterals.



Here are the definitions of the shape shown above.

- Quadrilateral: A polygon with four sides and four vertices.
- Convex quadrilateral: A quadrilateral whose every internal angle is at most 180 degrees and Every line segment between two vertices of the quadrilateral remains inside or on the boundary of the quadrilateral.
- Concave quadrilateral: A quadrilateral that is not convex.
- Trapezoid: A convex quadrilateral in which one pair of opposite sides is parallel.
- Parallelogram: A trapezoid whose opposite sides have equal length, opposite angles are equal and the diagonals bisect each other.
- Rectangle: A parallelogram where each angle is a right angle.
- Square: A rectangle where four sides have equal length.

A good class hierarchy must conform to that each single subclass 'is a' superclass. From the above example, a square is a rectangle, a rectangle is a parallelogram, and so on. According to the hierarchy, a square is also a quadrilateral, or we can say that a square is a more specific case of a quadrilateral. A quadrilateral is the most general case of all the shapes in the hierarchy. Therefore, each shape is a quadrilateral with additional attributes and behaviors.

Access Control

When data members and methods are declared or defined in a class, creator of the class can give access right to each of them. We have seen the keywords `public` and `private` used in front of the declaration of data members and methods. These two keywords determine which classes can access those data members and methods. Class access rights regarding to three



access levels used in Java: **public**, **private**, and **protected**, are listed in the table below. When no explicit access levels are used, the access rights are the ones listed in the row labeled “default”.

Access Level	Accessing Class		
	current class	subclass	other
public	☑	☑	☑
protected	☑	☑	☒
default	☑	☒	☒
private	☑	☒	☒

A table cell marked with ☑ means that the corresponding accessing class can access resources with the corresponding access level. In other words, the dot operator can be used in the code listed inside the accessing class to access resources with the corresponding access level directly. A table cell marked with ☒ means that we cannot use the dot operator inside the accessing class in order to access resources with that access level directly.

Observe the following example.

```
public class MyDot2D
{
    private double x=0;
    private double y=0;

    public double getX(){
        return x;
    }
    public double getY(){
        return y;
    }
}
```

```
public class AccessLevelDemo1
{
    public static void main(String[] args)
    {
        MyDot2D p = new MyDot2D();
        System.out.println(p.x);
        System.out.println(p.y);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac AccessLevelDemo1.java
AccessLevelDemo1.java:6: x has private access in MyDot2D
    System.out.println(p.x);
                        ^
AccessLevelDemo1.java:7: y has private access in MyDot2D
    System.out.println(p.y);
                        ^
2 errors
C:\>_
```

The above program yields compilation errors since **x** and **y** have private access level and attempts to access them directly using the dot operator are made inside a class (*AccessLevelDemo1*) other than *MyDot2D*.

Keyword ‘super’

Data members or methods in a superclass having either *public* or *protected* access levels can be accessed directly from inside the class definitions of its subclasses by referring to the name of the data members or methods to be accessed. However, if referring by names alone causes



any confusion, for example, when names already used in the superclass are reused in the subclass, we can use the keyword '*super*' to identify explicitly that data members or methods of the superclass are the ones desired. This keyword acts as a reference to the superclass instance associated with the current instance of a subclass, in the same fashion as '*this*' is used as the reference to the current instance of the class it is used in.

Consider the following example and observe its output.

```
public class MySuperclass
{
    public int a = 1;
    public int b = 2;
    public void f(){
        System.out.println("\tf() of MySuperclass is called.");
    }
    public void g(){
        System.out.println("\tg() of MySuperclass is called.");
    }
}
```

```
public class MySubclass extends MySuperclass
{
    public int a = 9;
    public void f(){
        System.out.println("\tf() of MySubclass is called.");
    }
    public void test(){
        System.out.println("a = "+a);
        System.out.println("b = "+b);
        System.out.println("super.a = "+super.a);
        System.out.println("super.b = "+super.b);
        System.out.println("f() ");
        f();
        System.out.println("g() ");
        g();
        System.out.println("super.f() ");
        super.f();
        System.out.println("super.g() ");
        super.g();
    }
}
```

```
public class SuperDemo
{
    public static void main(String[] args)
    {
        MySubclass y = new MySubclass();
        y.test();
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac SuperDemo.java
C:\>java SuperDemo
a = 9
b = 2
super.a = 1
super.b = 2
f()
    f() of MySubclass is called.
g()
    g() of MySuperclass is called.
super.f()
    f() of MySuperclass is called.
super.g()
    g() of MySuperclass is called.
```



Object Variables

A variable whose type is a class can refer to an object of that class as well as an object of any one of its subclasses. However, the opposite is not true. A variable of a class cannot be used to refer to an object of its superclass, just like when a variable cannot be used to refer an object whose class is different from the variable type.

Recall the class hierarchy of *L12A*, *L12B*, and *L12C*. Each of the following code segments are valid and compiled without errors.

```
L12A a = new L12A();  
L12B b = new L12B();  
L12C c = new L12C();  
a = b;  
a = c;
```

```
L12A a1 = new L12B();  
L12A a2 = new L12C();
```

```
L12A [] a = new L12A[3];  
a[0] = new L12A();  
a[1] = new L12B();  
a[3] = new L12C();
```

However, the following program cannot be compiled successfully.

```
public class ObjectVariableInvalidDemo  
{  
    public static void main(String[] args)  
    {  
        L12A a = new L12A();  
        L12B b = new L12B();  
        L12C c = new L12C();  
        b = a;  
        c = a;  
        b = c;  
    }  
}
```

```
C:\WINDOWS\system32\cmd.exe  
C:\>javac ObjectVariableInvalidDemo.java  
ObjectVariableInvalidDemo.java:8: incompatible types  
found   : L12A  
required: L12B  
    b = a;  
ObjectVariableInvalidDemo.java:9: incompatible types  
found   : L12A  
required: L12C  
    c = a;  
ObjectVariableInvalidDemo.java:10: incompatible types  
found   : L12C  
required: L12B  
    b = c;  
3 errors  
C:\>
```

Method Overriding and Polymorphism

When a method that has already been defined in a class is redefined in its subclass using the same identifier and the same list of input arguments, it is called that the method defined in the subclass *overrides* the one in its superclass. When this happens, which method to be invoked, i.e. the one in the superclass or the one in the subclass, depends on the type of the object from which the method is invoked.

Consider the following program and its output.



```
public class MethodOverridingDemo1
{
    public static void main(String[] args)
    {
        L12A a = new L12A();
        a.x = 2;
        a.d = 1.0;

        L12D d = new L12D();
        d.x = 2;
        d.d = 1.0;
        d.y = 2.5;

        System.out.println("a.f()="+a.f());
        System.out.println("d.f()="+d.f());

        a = d;
        System.out.println("a.f()="+a.f());
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac MethodOverridingDemo1.java
C:\>java MethodOverridingDemo1
a.f()=2.0
d.f()=4.5
a.f()=4.5
C:\>_
```

The important points in this example is the methods invoked when `a.f()` and `d.f()` are called. On line 5, `a` is made to refer to an `L12A` object. On line 9, `a` is made to refer to an `L12D` object. Therefore, `a.f()` on line 14 invokes `f()` from `L12A`, and `a.f()` on line 15 invokes `f()` from `L12D`. However, after that, `a` is made to refer to an `L12D` object (the same object that `a` refers to), which is a valid operation since `L12D` is a subclass of `L12A`. Consequently, `a.f()` on line 18 invokes `f()` from `L12D` instead of `f()` from `L12A`.

In conclusion, you have to keep in mind that it is not the variable type that determines the method invocation, but the type of object to which the variable refers.

Polymorphism is the ability of objects belonging to different types to respond to method calls of methods with the same name, each one according to an appropriate type-specific behavior. Java does not have to know the exact type of the object in advance, so this behavior can be implemented at run time. This is called *late binding* or *dynamic binding*.

Consider the following class definitions.

```
public class BookItem
{
    protected String name;
    protected double listedPrice;

    public BookItem(String name,double price){
        this.name = name;
        listedPrice = price;
    }
    public double getSellingPrice(){
        return listedPrice;
    }
    public double getListedPrice(){
        return listedPrice;
    }
    public String toString(){
        return "BookItem:"+name;
    }
}
```



```
public class UsedBook extends BookItem
{
    protected double discountFactor;

    public UsedBook(String name,double price,double discountFactor){
        super(name,price);
        this.discountFactor = discountFactor;
    }
    public double getSellingPrice(){
        return (1-discountFactor)*listedPrice;
    }
    public String toString(){
        return "UsedBook:"+name;
    }
}
```

```
public class RareBook extends BookItem
{
    protected double premiumFactor;

    public RareBook(String name,double price,double premiumFactor){
        super(name,price);
        this.premiumFactor = premiumFactor;
    }
    public double getSellingPrice(){
        return (1+premiumFactor)*listedPrice;
    }
    public String toString(){
        return "RareBook:"+name;
    }
}
```

The three class definitions listed above show that the class *BookItem* is the superclass of the other two classes, *UsedBook* and *RareBook*. Each of the three class has its own implementation of *getSellingPrice()* and *toString()*. As mentioned, which implementation of these two methods is invoked depends on the type of the object from which the method is called. The following program uses the three classes we have just defined.

```
import java.io.*;
public class BookShop
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Number of books:");
        int nBooks = Integer.parseInt(stdin.readLine());
        BookItem [] bookInventory = new BookItem[nBooks];
        int i=1;
        while(i<=nBooks){
            bookInventory[i-1] = getBook(i);
            i++;
        }
        showBookInfo(bookInventory);
    }
    public static BookItem getBook(int i) throws IOException{
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        BookItem b;
        while(true){
            System.out.println("-----\nBook #"+i);
            System.out.print("Type (1=Regular, 2=Used, 3=Rare):");
            int type = Integer.parseInt(stdin.readLine());
            if(type<1 || type>3){
                System.out.println("Invalid type.");
                continue;
            }
            System.out.print("Book name:");
            String name = stdin.readLine();
            System.out.print("Listed price:");

```



```

double price = Double.parseDouble(stdin.readLine());      34
double factor;                                           35
switch(type){                                           36
    case 1:                                             37
        b = new BookItem(name,price);                 38
        break;                                         39
    case 2:                                             40
        System.out.print("Discount factor:");         41
        factor = Double.parseDouble(stdin.readLine()); 42
        b = new UsedBook(name,price,factor);          43
        break;                                         44
    case 3:                                             45
        System.out.print("Premium factor:");          46
        factor = Double.parseDouble(stdin.readLine()); 47
        b = new RareBook(name,price,factor);          48
        break;                                         49
    default:                                           50
        b = null;                                     51
}                                                       52
break;                                                 53
}                                                       54
return b;                                              55
}                                                       56
public static void showBookInfo(BookItem [] bookInventory){ 57
    System.out.println("#####");                     58
    for(int i=0;i<bookInventory.length;i++){           59
        System.out.println("Item #"+i+":\t"+bookInventory[i]); 60
        System.out.print("Listed Price:\t");           61
        System.out.println(bookInventory[i].getListedPrice()); 62
        System.out.print("Selling Price:\t");          63
        System.out.println(bookInventory[i].getSellingPrice()); 64
        System.out.println();                          65
    }                                                    66
    System.out.println("#####");                     67
}                                                       68
}                                                       69

```

Let's look at *main()* first. The program asks the user to input the number of books to be stored in an array of *BookItem*. Then, the user is asked to input the information of each book. Finally, the program prints out information associated with each element in the array. We can see that at the compilation of the code, the program has no way to know the type of each element in the array *bookInventory* except that it has to be of the type *BookItem* or one of its subclasses.

On line 12 to line 15, a *while* loop is used to gather information of each array element. Inside the method *getBook()*, an object of type either *BookItem*, *UsedBook*, or *RareBook* is created and returned from the method. The returned object is referred to as an element of *bookInventory*. Notice that the return type of *getBook()* is *BookItem*, with which it is also perfectly correct for the method to return an object of its subclass.

Polymorphism is used in *showBookInfo()* since the program does not know in advance about the exact type of each object in *bookInventory*. For each element in the array, *toString()* is called inexplicitly on line 60, *getListedPrice()* is called on line 62, and *getSellingPrice()* is called on line 64. Late binding through polymorphism in Java makes the program know at run-time which implementation of *toString()* and *getSellingPrice()* (i.e. the implementations in the superclass or the subclass) to be invoked based on the type of the object from which the methods are called. For *getListedPrice()*, it is only implemented in the superclass. Therefore, there is no confusion on which method to be invoked.

Observe an output of *BookShop.java* shown below. Pay attention to the value of the selling price for each element printed on screen.



```
C:\WINDOWS\system32\cmd.exe

C:\>javac BookShop.java
C:\>java BookShop
Number of books:3
-----
Book #1
Type<1=Regular,2=Used,3=Rare>:1
Book name:The Alchemist
Listed price:500.00
-----
Book #2
Type<1=Regular,2=Used,3=Rare>:3
Book name:Behind ISE, limited edition
Listed price:1999.99
Premium factor:0.5
-----
Book #3
Type<1=Regular,2=Used,3=Rare>:2
Book name:Calculus I
Listed price:350.00
Discount factor:0.3
#####
Item #0:      BookItem:The Alchemist
Listed Price: 500.0
Selling Price: 500.0

Item #1:      RareBook:Behind ISE, limited edition
Listed Price: 1999.99
Selling Price: 2999.985

Item #2:      UsedBook:Calculus I
Listed Price: 350.0
Selling Price: 244.99999999999997

#####
C:\>_
```

Note that we will not cover *abstract classes* and *interface* in this course. These concepts are usually used with polymorphism in order to obtain more benefits from late binding.

Instance Creation Mechanism

When an instance or object of a class is created, if there is no explicit call of any constructors of its superclass (like when *super()* is used in the constructor of *EngStudent* mentioned earlier), the no-argument constructor of the superclass is called automatically before the execution of any statements in the subclass's constructor (if there are any).

To demonstrate this mechanism, let's look at the following class definitions.

```
public class L12E
{
    public L12E(){
        System.out.println("\tL12E() is called.");
    }
}
```

```
public class L12F extends L12E
{
}
```

```
public class L12G extends L12E
{
    public L12G(){
        System.out.println("\tL12G is called.");
    }
    public L12G(String s){
        System.out.println("\tL12G(String s) is called.");
    }
    public L12G(int i){
        super();
        System.out.println("\tL12G(int i) is called.");
    }
}
```

L12F and *L12G* are subclasses of *L12E*. Let's consider the following program. Pay attention to messages printed on screen when each object is created.



```
public class CreationDemo1
{
    public static void main(String[] args)
    {
        System.out.println("1)-----");
        L12F f = new L12F();
        System.out.println("2)-----");
        L12G g1 = new L12G();
        System.out.println("3)-----");
        L12G g2 = new L12G("Hello");
        System.out.println("4)-----");
        L12G g3 = new L12G(8);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac CreationDemo1.java
C:\>java CreationDemo1
1)-----
   L12E() is called.
2)-----
   L12E() is called.
   L12G is called.
3)-----
   L12E() is called.
   L12G(String s) is called.
4)-----
   L12E() is called.
   L12G(int i) is called.
C:\>
```

When an object of *L12F* is created on line 5, since there is no implementation of any constructor in *L12F*, the no-argument constructor of its superclass, *L12E()*, is invoked automatically.

When an object of *L12G* is created, if either *L12G()* or *L12G(String s)* is called, the no-argument constructor of its superclass, *L12E()*, is again invoked automatically since there is no explicit call to the constructor of *L12E*. If *L12G(int i)* is called, the constructor of *L12E* is not called automatically since it is explicitly called by the first statement of *L12G(int i)*.

Keep in mind that, if you want to explicitly call any constructor of the superclass, the *super()* statement has to be used in the first statement only. Otherwise, the class will not be compiled successfully. The following class is an example of such a case that cannot be compiled successfully.

```
public class L12H extends L12E
{
    public L12H(){
        System.out.println("\tL12H is called.");
        super();
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac L12H.java
L12H.java:5: call to super must be first statement in constructor
    super();
    ^
1 error
C:\>_
```