INHERITANCE and POLYMORPHISM

Only public and protected members inherited to the derived class
Inheritance is is-A relationship. child is-A Base.

Ex: WageEmployee is is-A Employee

Two types of classes
-Concrete class

- instantiate a concrete class

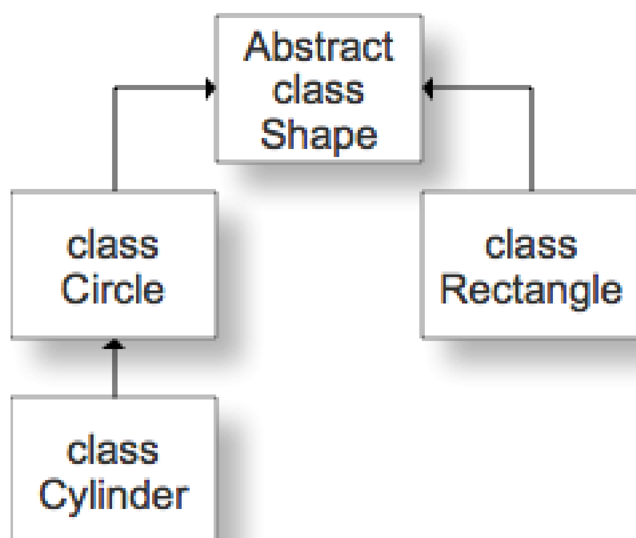
-Abstract class

- As a base class
- cannot instantiate a abstract class
- must have at least one abstract method

Abstract Method

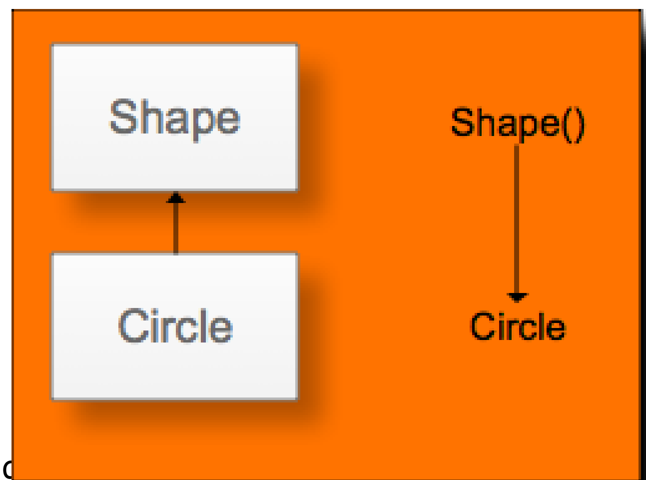
- declare in the abstract class
  - public abstract double computePay();
- Implement (define) the in the child class which is derived from the abstract class.

INHERITANCE EXAMPLE

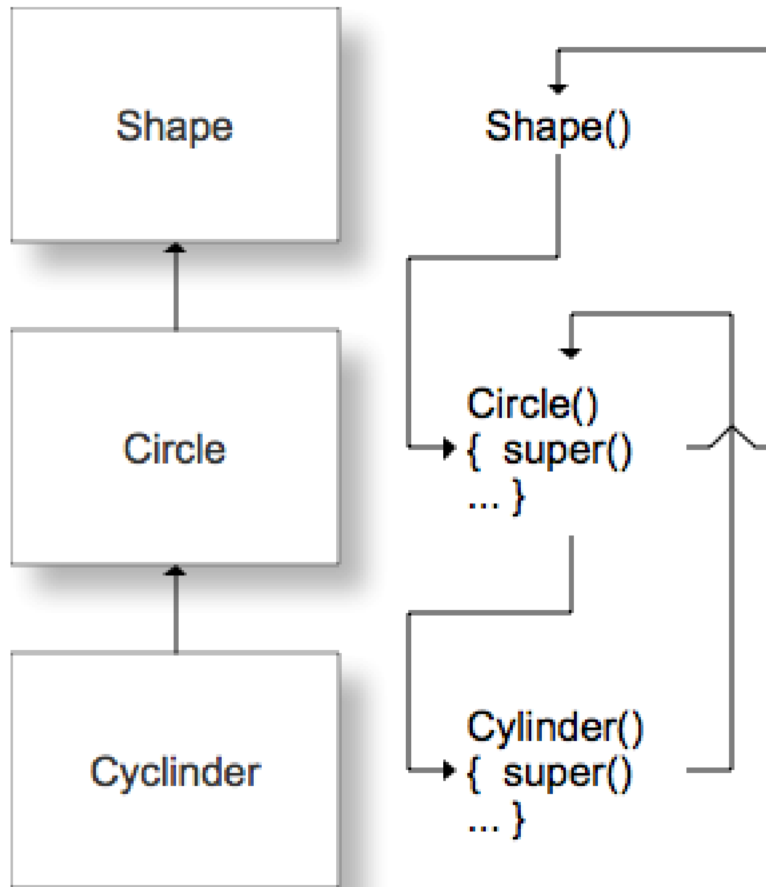


```
/**
        class description
*/
public abstract class Shape
{
        private double xPos;
        private double yPos;


        /** default-argument constructor
        */
        Shape ()
        {
                xPos=0;
                yPos=0;
        }
        /** A constructor to assign values to pos & yPos
                @param  int  x position
                @parma  int y position
        */
        Shape (double x, double y)
        {
                xPos=x;
                yPos=y;
        }
        /** get the xPos
                @return int xPos
        */
        public final double getxPos()
        //final: cannot override the method in the derived class
        {
                return xPos;
        }
```

```
        public final double getyPos()
        {
                return yPos;
        }
        public void moveTo(double x, double y)
        {
                xPos = x;
                yPos = y;
        }
        public String toString()
        {  return "x = " + xPos + " y= " + yPos;
        }
        public abstract double computeArea();
}//end class Shape
public class Circle extends Shape
{
        private double radius;
        Circle()
        {
                super(); //call Shape
                radius=0;
        }
// argument Circle constructor
        public  Circle (double x, double y,
        {
                super(x,y); //call Shape(x,y)
                radius = r;
        }
        public double computeArea()
        {
                return Math.PI * radius * radius;
        }
        public double getRadius()
        {
                return radius;
        }
```



```
        public String toString()
        {
                return super.toString() + “Radius = “+ radius;
        }
}//end class Circle

public Cylinder extends Circle
{
        private double height;
        private double z;
        public Cylinder()
        {
                super();
                height = 0.0;
                z = 0.0
        }
        public Cylinder(double x, double y, double r, double h, double z)
        {
                super(x,y,r); //call Circle 3-arg constructor
                height = h;
                this.z = z
        }
```



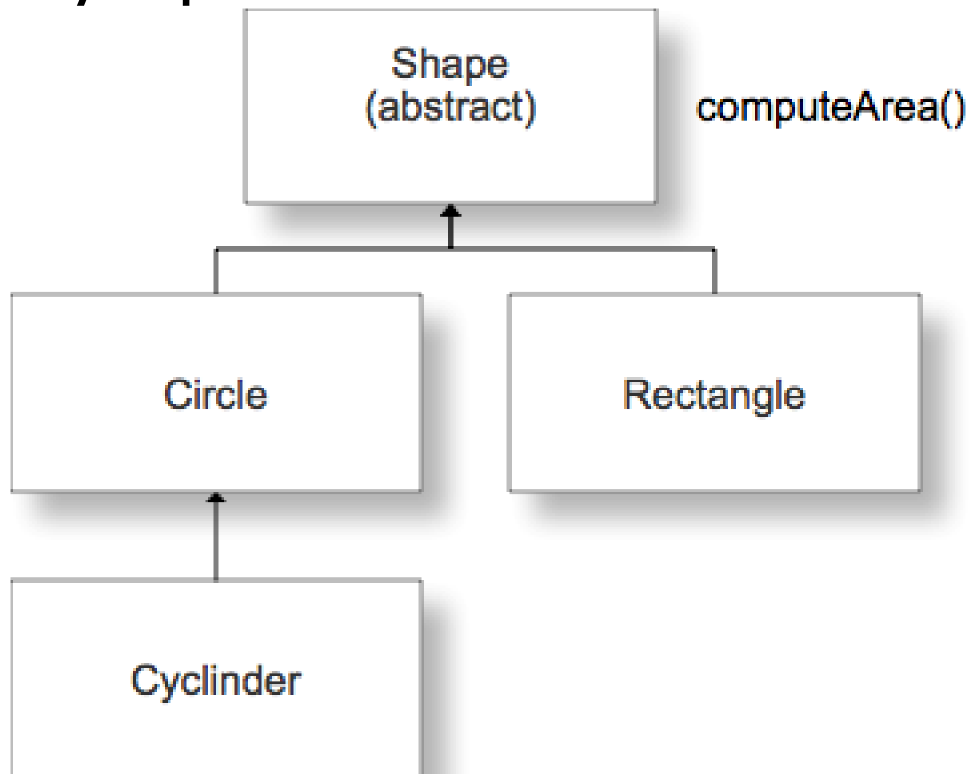

```
public void moveTo(double x, double y, double z)
{
        super.moveTo(x,y);
        this.z = z;
}
public double computeArea()
{
        return 2 * super.computeArea() + 2.Math.PI * getRadius() *
        height;
}
public double computeVolume()
{
        return super.computeArea() * height;
}
```

```
        public double getHeight()
        {
                return height;
        }
}//end class
```

Summary

- Extends
- Is-A relationship describes inheritance
  - Class B extends A
  - B Is-A A
- Use inheritance when Is-A relationship makes sense.
- no multiple inheritance
- multiple interface inheritance
- super()
  - call the parent constructor
- If the method is overridden in the subclass, use super.method to call the method of the base
- Protected is like private except it is inherited to the subclass.

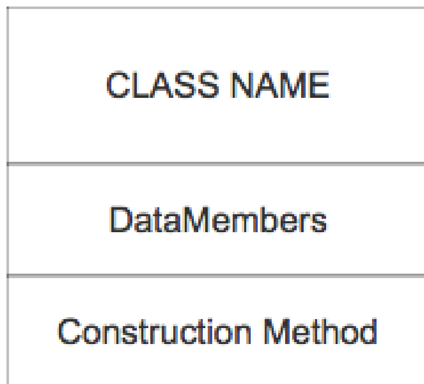
Polymorphism




```
public class TestShape
{ ________    main(_______)
            {
              Shape [] s = new Shape[5];
              s[0] = new Circle(10.0,2.0.5.0);
              Circle c2 = new Circle(5.0,1.0,3.0);
              s[1] = c2;
              s[2] = new Rectangle(_______);
              s[3] = new Cylinder(_________);
              s[4] = new Cylinder(_________);
              //Total area of all object
              double total = 0.0;
              for(int i = 0; i < s.length; i++)
              {
                    total += s[i].computeArea();
              }
```

```
//Total area of all Rectangle objects
double sumArea = 0.0;
for(int i = 0; i < s.length; i++)
{
	if(s[i] instanceof Rectangle)
		sumArea += s[i].computeArea;
}
//Total volume of Cylinders
double totalVolume = 0.0;
for(int i = 0; i < s.length; i++)
{
	if(s[i] instanceof Cyclinder)
	    totalVolume += ((Cylinder) s[i]).computeVolume();
}
```

UML (Unified Modeling Language) Diagram

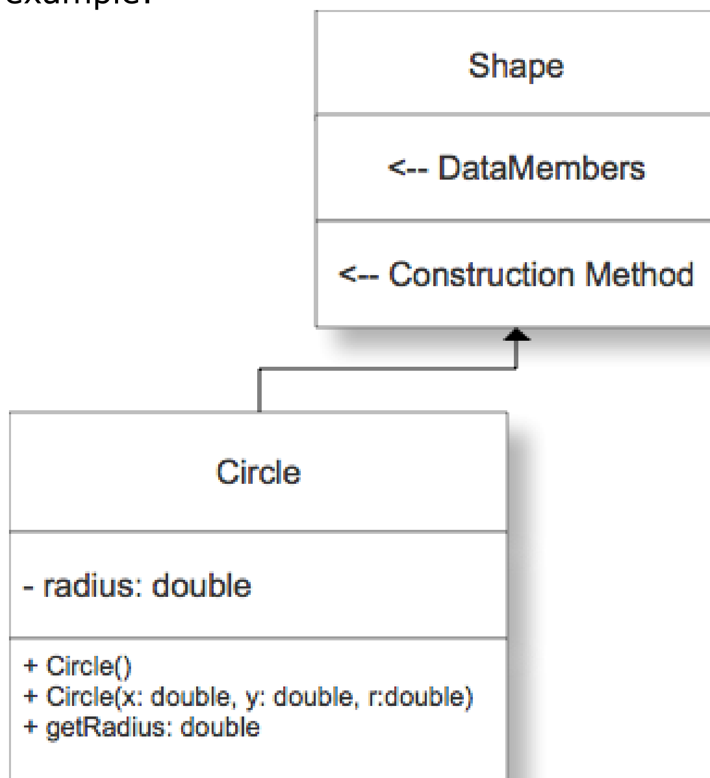


- private members

+ public member

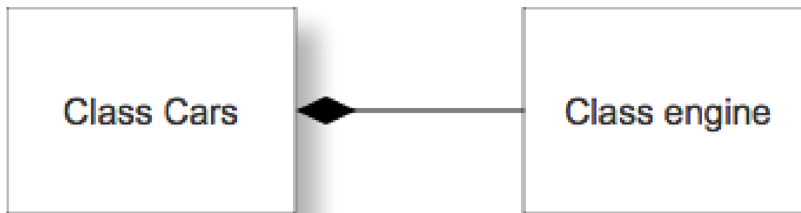
protected member

example:

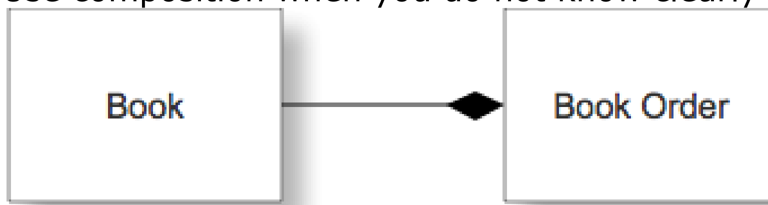


Composition

composition – has a relationship



Use composition when you do not know clearly the is-a relationship




```
public class Book
{
        String code;
        String title;
        double price;
        public Book()
        {
                code = “”;
                title = “”;
                price = 0.0;
        }

        public Book(String c, String t, double p)
        {
                code = c;
                title = t;
                price = p;
        }
        …

        public double getPrice()
        {
        return price;
        }
        public void setBook(Book bb)
        {b = bb;}
}
```

```
public class BookOrder
{
 private Book book;
 private int quantity;
 private double total;
 public BookOrder()
 {
        book = new Book();
        quantity = 0;
        total = 0.0;
 }

 public BookOrder(String c, String t, double p, int q, double tot)
 {
        book = new Book(c,t,p);
        quantity = q;
        total = tot;
 }
 public void setTotal()
 {
        total = quantity * book.getPrice();
 }
 public Book getBook()
 {
 return book;
 }
}
```

INTERFACE

Interface can contains only abstract method (no keyword abstract) and constant variables. The class implementing interfaces must define all the abstract methods.

```
public interface B
{
  constant variables;
  abstract methods; //No keyword public
}
```

In Java, multiple inheritances are not allowed, but multiple interfaces are.

```
public class D extends A implements B, C
{
 …………………………..
}
```

Ex:

```
interface Animal {

   public void eat();
   public void travel();
}

/* File name : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

This would produce the following result:

```
ammal eats
ammal travels
```

Java interface Comparable

Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object).It provide only single sorting sequence i.e. you can sort

the elements on based on single datamember only.
Syntax:

```
public interface Comparable
{
        int CompareTo(Obj o);
}
```

Example:
Student.java

```
class Student implements Comparable
 {
 int rollno;
 String name;
 int age;
 Student(int rollno,String name,int age){
 this.rollno=rollno;
 this.name=name;
 this.age=age;
 }

 public int compareTo(Object obj){
 Student st=(Student)obj;
 if(age==st.age)
 return 0;
 else if(age>st.age)
 return 1;
 else
 return -1;
 }

 }
```

SimpleJava.java

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

 class TestSort3{
 public static void main(String args[]){

 ArrayList al=new ArrayList();
 al.add(new Student(101,"Vijay",23));
```

```
    al.add(new Student(106,"Ajay",27));
    al.add(new Student(105,"Jai",21));

    Collections.sort(al);
  //If you an array named ar, then the sort statement should be
  //Array.sort(ar);
    Iterator itr=al.iterator();
    while(itr.hasNext()){
    Student st=(Student)itr.next();
    System.out.println(st.rollno+""+st.name+""+st.age);
      }
    }
    }
```

Output

105 Jai 21
101 Vijay 23
106 Ajay 2

Java Comparator interface

Comparator interface is used to order the objects of user-defined class. This interface is found in java.util package and contains 2 methods compare (Object obj1,Object obj2) and equals(Object element).
It provides multiple sorting sequence i.e. you can sort the elements based on any data member. For instance it may be on rollno, name, age or anything else.

Syntax

public int compare(Object obj1,Object obj2): compares the first object with second object.

Example of sorting the elements of List that contains user-defined class objects on the basis of age and name

In this example, we have created 4 java classes:

Student.java
AgeComparator.java
NameComparator.java
Simple.java

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
```

AgeComparator.java

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

```
import java.util.*;
class AgeComparator implements Comparator{
public int Compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```
import java.util.*;
class NameComparator implements Comparator{
public int Compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

return s1.name.compareTo(s2.name);
}
}
```

Simple.java

In this class, we are printing the objects values by sorting on the basis of name and age.

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

class Simple{
public static void main(String args[]){

ArrayList al=new ArrayList();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name...");

Collections.sort(al,new NameComparator());
Iterator itr=al.iterator();
while(itr.hasNext()){
Student st=(Student)itr.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}

System.out.println("sorting by age...");
Collections.sort(al,new AgeComparator());
Iterator itr2=al.iterator();
while(itr2.hasNext()){
Student st=(Student)itr2.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Output

Sorting by Name...

106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by age...
105 Jai 21
101 Vijay 23
106 Ajay 27

Differences between Comparable and Comparator

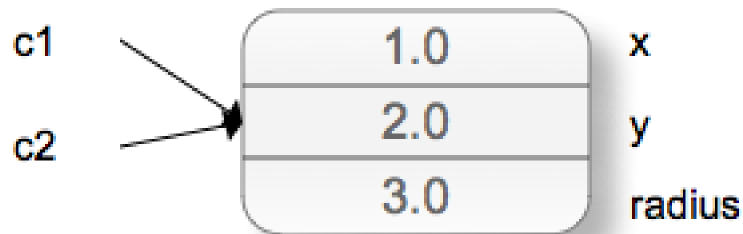
Comparable and Comparator both are interfaces and can be used to sort collection elements.
But there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
2) Comparable **affects the original class** i.e. actual class is modified.	Comparator **doesn't affect the original class** i.e. actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is found in **java.lang** package.	Comparator is found in **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method.

Operator == and the Object method equals

The operator == tests whether two references are the same objects(referential equality)

```
Circle c1 = new Circle(1.0,2.0,3.0);
Circle c2 = c1;
```




```
if (c1 == c2) //returns true based on reference not value

Circle c1 = new Circle(1.0,2.0,3.0);
Circle c2 = new Circle(1.0,2.0,3.0);
if (c1 == c2)// false
```

Overriding the method equals for logical equality.

```
public class Circle()
{
        ...
        public Boolean equals(Object otherobject)
        {
                if ( otherobject instanceof Circle)
                {
                return radius == ((Circle) otherobject).radius;
                }

                else
                {
                  return false;
                }
        }
}

Circle c1 = new Circle(1.0,2.0,3.0);
Circle c2 = new Circle(1.0,2.0,3.0);
if (c1.equals(c2)) // true
```

Clone
Sometimes you want to make a copy of an object.

```
c1 = c2;
```

The statement above does not create a duplicate object.
It simply assigns the reference of c2 to c1. To create an object with separate memory space use the clone() method.

Java interface Cloneable

- The object cloning is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.
- The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement
- Cloneable interface, clone() method generates CloneNotSupportedException.
- The clone() method is defined in the Object class. Syntax of the clone() method is as follows:

```
protected Object clone() throws CloneNotSupportedException
```

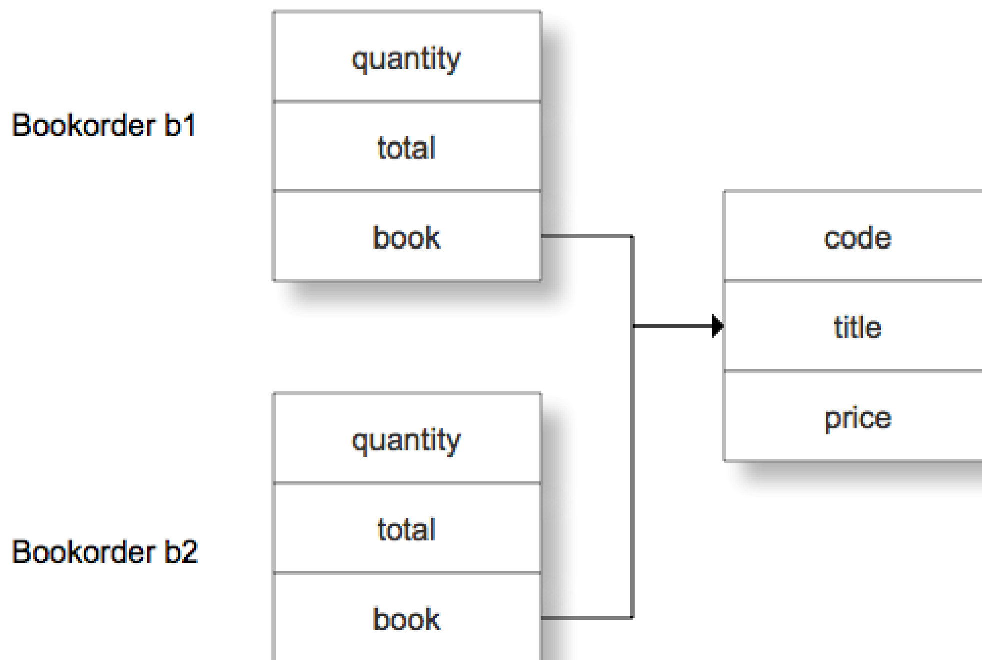
The clone() method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

```
public class Book implements Cloneable
{
        private String title;
        private String code;
        private double price;

        public Object clone() throws CloneNotSupportedException
        {
              return super.clone();
        }
}
```

<u>In the main()</u>

```
Book b1 = new Book(_______);
Book b2 = (Book) b1.clone();
```

Note: If you do not modify clone() and have objects in the object, you will make a shallow copy

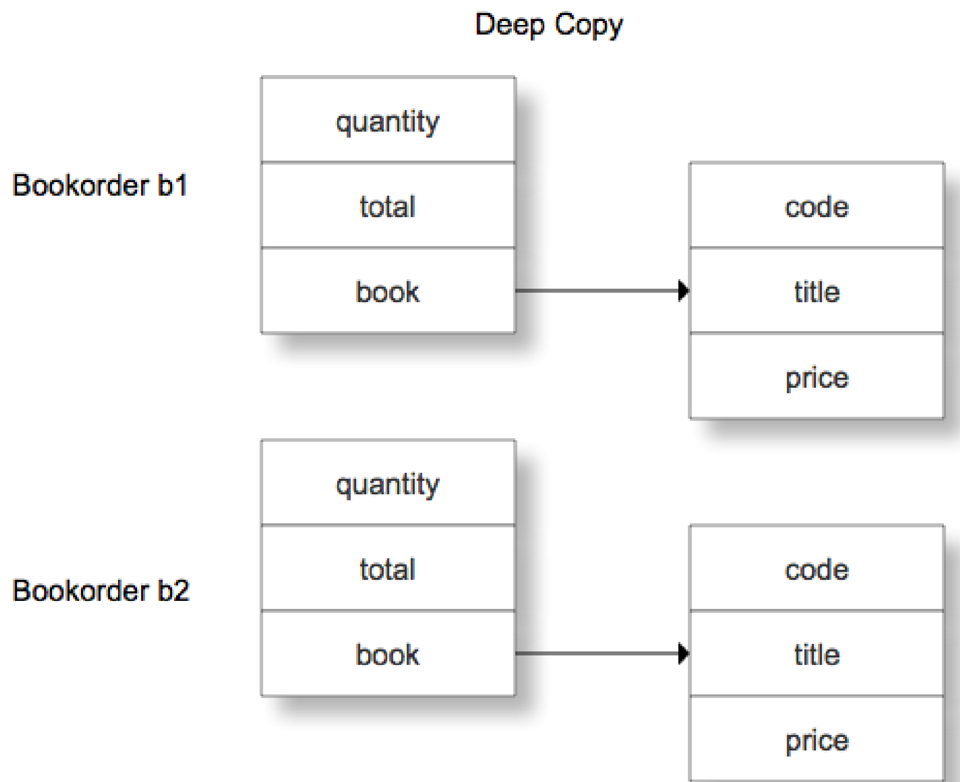


Deep copy

```
public class Bookorder implements Cloneable
{
        private int quantity;
        private double total;
        private Book book;
        public void setBook(Book b)
        { book = b;
        }

        public Object clone() throws CloneNotSupportedException
        {
                Bookorder b = (Bookorder) super.clone;
                book = (Book) book.clone;
                b.setBook(book);
                return b;
        }
}
```

<u>In the main()</u>

```
Bookorder b1 = new Bookorder(_____);
Bookorder b2 = (Bookorder) b1.clone();
```



Inheritance, interface and casting

Recall that inheritance implements the "is-a" relationship. For example, in the Introduction to Inheritance notes, class RaceHorse was defined to be a subclass of Horse (because every RaceHorse is a Horse). Therefore, it makes sense that a RaceHorse can be used in any context that expects a Horse. For example:

```
Horse h;
RaceHorse r = new RaceHorse();
h = r;
```

Variable h is of type Horse, so in an assignment of the form

h = ...

the right-hand side of the assignment should be of type Horse, too. However, since a RaceHorse is-a Horse, it is OK for the right-hand side to be of type RaceHorse, as in the above example.

Note that every Horse is not a RaceHorse, so in general, a Horse cannot be used when a RaceHorse is expected. For example, the following code causes a compile-time error:

```
Horse h = new Horse();
RaceHorse r = h;
```

Here are three more examples of code that sets a Horse variable or parameter to point to a RaceHorse object:

```
(1)   Horse h = new RaceHorse();  // h is of type Horse, but it points to a
                                  // RaceHorse object

(2)   public static void f( Horse h ) { ... }
      ...
      RaceHorse r = new RaceHorse();
      f( r );  // f's formal parameter h is of type Horse, but the actual
               // parameter points to a RaceHorse object

(3)   public static RaceHorse g() { // return a pointer to a RaceHorse object
}
      ...
      Horse h = g();  // h is of type Horse, but it now points to a RaceHorse
                      // object
```

If you know that at a particular point in your code a Horse variable is really pointing to a RaceHorse object, then you can use that variable in a context that expects a RaceHorse, but you must provide a cast. Note that there are two kinds of errors that can arise if you get this wrong:

1. missing cast => compile-time error
2. incorrect cast => runtime error (ClassCastException is thrown)

<u>Examples</u>:
Assume that we have the following declarations of function f and variables h1 and h2:

```
public static void f( RaceHorse r ) { ... }
Horse h1 = new RaceHorse();
Horse h2 = new Horse();
```

Now consider the following three calls to f:

1. f(h1); // compile-time error (missing cast)
2. f((RaceHorse)h1); // fine! h1 really does point to a RaceHorse
3. f((RaceHorse)h2); // runtime error (bad cast) h2 points to a Horse

Note that when you use a cast you must think about what expression you are casting, and perhaps use parentheses (if that expression is part of a larger expression). For example, suppose variable h is of type Horse but actually points to a RaceHorse. You can call h's WinRace method (which is a RaceHorse method, but not a Horse method), but you have to use a cast, like this:

```
((RaceHorse)h).WinRace();
```

The parentheses tell the compiler that you are casting just variable h to be of type RaceHorse. If you omit the parentheses:

```
(RaceHorse)h.WinRace(); // NO! This doesn't work!!
```

the compiler will think you are casting the result of the call h.WinRace() to be of type RaceHorse.