Comparator Interface

- *Allows two objects to compare explicitly.*
- *Syntax For Unparametrized:*

```
public interface Comparator
{
 int compare(Object O1, Object O2);
}
```

- *Syntax For Parametrized:*

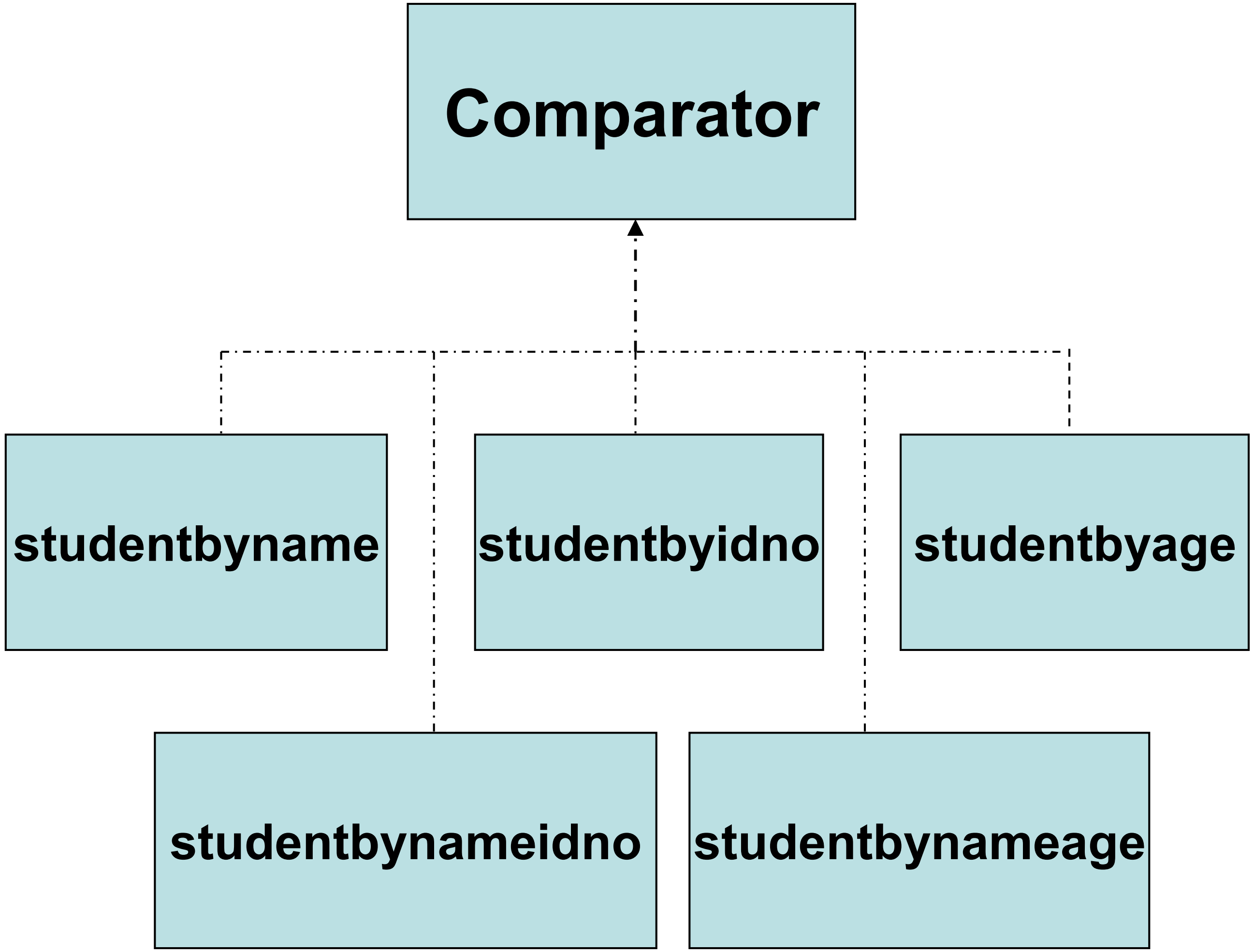
```
public interface Comparator<T>
{
 int compare(T O1, T O2);
}
```

<<T>> type of object reference

- *Does not require change in the base class.*
- *We can define as many comparator classes for the base class.*
- *Each Comparator class implements Comparator interface and provides different logic for comparisons of objects.*
- *But as we are passing both parameters explicitly, we have to type cast both Object types to their base type before implementing the logic.*

Student

```
class Student
{
private String name;
private String idno;
private int age;
private String city;
……………………..
……………………..
```



Comparator

studentbyname

studentbyidno

studentbyage

studentbynameidno

studentbynameage

```
class studentbyname implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
return s1.getName().compareTo(s2.getName());
}
}
class studentbyidno implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
return s1.getIdNo().compareTo(s2.getIdNo());
}
}
```

```
class studentbyage implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
if( s1.getAge() > s2.getAge() ) return 1;
if( s1.getAge() < s2.getAge() ) return -1;
return 0;
}
}
```

```
class studentbynameidno implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
if( s1.getName().compareTo(s2.getName()) == 0)
return s1.getIdNo().compareTo(s2.getIdNo());
else
return s1.getName().compareTo(s2.getName());
} }
```

```
class studentbynameage implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
if( s1.getName().compareTo(s2.getName()) == 0)
return s1.getAge() – s2.getAge();
else
return s1.getName().compareTo(s2.getName());
}
}
```

```
Import java.util.*;
class comparatorTest
{
public static void main(String args[])
{
Student[] students = new Student[5];
Student[0] = new Student(“John”,”2000A1Ps234”,23,”Pilani”);
Student[1] = new Student(“Meera”,”2001A1Ps234”,23,”Pilani”);
Student[2] = new Student(“Kamal”,”2001A1Ps344”,23,”Pilani”);
Student[3] = new Student(“Ram”,”2000A2Ps644”,23,”Pilani”);
Student[4] = new Student(“Sham”,”2000A7Ps543”,23,”Pilani”);

// Sort By Name
Comparator c1 = new studentbyname();
Arrays.sort(students,c1);
for(int i=0;i<students.length;i++)
System.out.println(students[i]);
```

```
// Sort By Idno
c1 = new studentbyidno();
Arrays.sort(students,c1);
for(int i=0;i<students.length;i++)
System.out.println(students[i]);

// Sort By Age
c1 = new studentbyage();
Arrays.sort(students,c1);
for(int i=0;i<students.length;i++)
System.out.println(students[i]);

// Sort by Name & Idno
c1 = new studentbynameidno();
Arrays.sort(students,c1);
for(int i=0;i<students.length;i++)
System.out.println(students[i]);
```

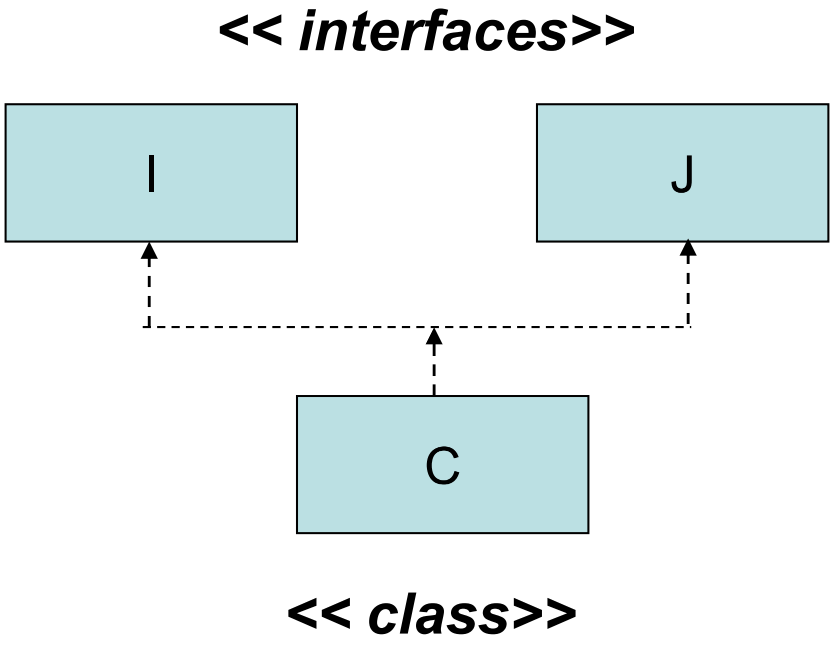
```
// Sort by Name & Age
c1 = new studentbynameage();
Arrays.sort(students,c1);
for(int i=0;i<students.length;i++)
System.out.println(students[i]);
} // End of Main
} // End of test class.
```

Exercise 1

- *Suppose C is a class that implements interfaces I and J. Which of the following Requires a type cast?*

```
C    c =       ……?
I    i =       …..?
J    j =       …..?

1.   c = i
2.   j = c
3.   i = j
```



First c = (C) i

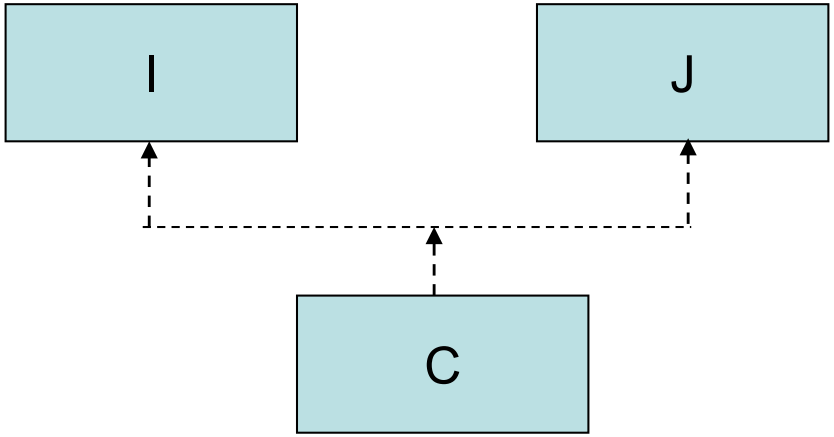
Exercise 2

- *Suppose C is a class that implements interfaces I and J. Which of the following will throw an Exception?*

```
C     c = new C()

1.    I i       = c;
2.    J j       = (J) i;
3.    C d       = (C) i;
```

<< interfaces>>



<< class>>

Second

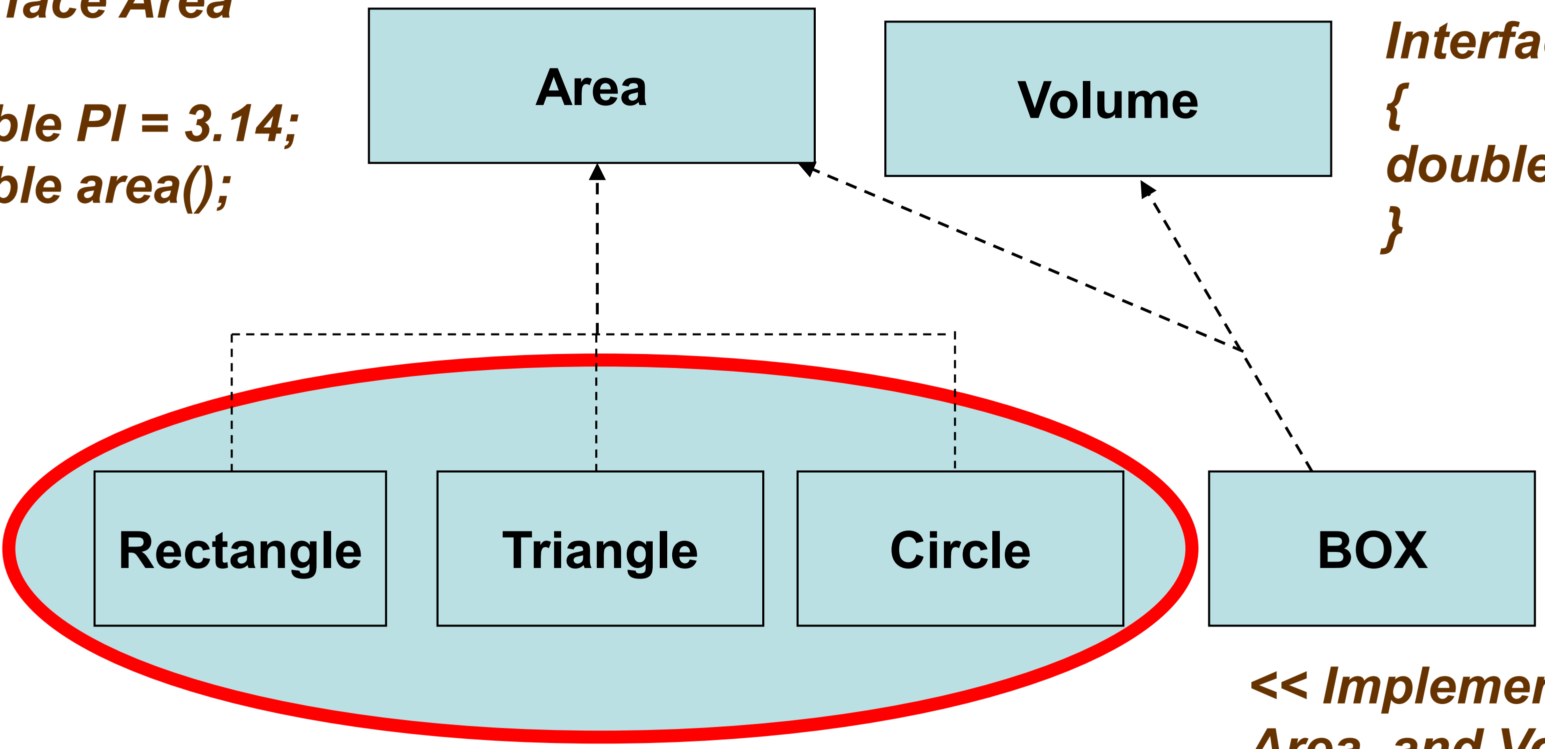
Exercise 3

- *Suppose the class Sandwich implements Editable interface. Which Of the following statements are legal?*

1. *Sandwich sub = new Sandwich();* **OK**
2. *Editable e = sub;* **OK**
3. *sub = e* ***Illegal***
4. *sub = (Sandwich) e;* **OK**

Write classes Implementing the Area and Volume Interface

```
Interface Area
{
double PI = 3.14;
double area();
}
```



```
Interface Volume
{
double volume();
}
```

<< Implements only Area interface>>

<< Implements both Area and Volume interface>>

Cont…

```
class circle implements Area
{
........................
........................
public double area()
{
.....
}
.........
........
}
```

```
class Rectangle implements Area
{
........................
........................
public double area()
{
.....
}
.........
........
}
```

```
class BOX implements Area , Volume
{
........................
........................
public double area()
{.....}
public double volume()
{......}
}
```

<< Implement the methods from interface with public scope>>

```
import java.util.*;
class A
{ int a;
}
class ctest
{
public static void main(String args[])
{
String[] names = {"OOP",“PES",“BANGALORE"};
Arrays.sort(names);
int[] data = { 10,-45,87,0,20,21 };
Arrays.sort(data);
A[] arr = new A[5];
arr[0] = new A();
arr[1] = new A();
arr[2] = new A();
arr[3] = new A();
arr[4] = new A();
Arrays.sort(arr);
} }
```

```
Exception in thread "main"
java.lang.ClassCastException: A
        at
java.util.Arrays.mergeSort(Arrays.java:1156)
        at java.util.Arrays.sort(Arrays.java:1080)
        at ctest.main(ctest.java:21)
```

Ok As String class implements Comparable

Ok As Integer class implements Comparable

NOT Ok as A class does not implements Comparable.

Unparametrized Comparator

```
import java.util.*;
~~class A implements Comparable~~
{
int a;
public int compareTo(Object other)
{
A a1 = (A) other;
if(this.a == a1.a ) return 0;
if(this.a < a1.a )  return -1;
return 1;
}
}
```

Type cast Object type to Base Type Before use

Unparametrized Comparable

```
class ctest
~~{~~
public static void main(String args[])
{
String[] names =
{"OOP",“SPECIAL",“TOPIC"};
Arrays.sort(names);
```

Will Work

```
int[] data = { 10,-45,87,0,20,21 };
Arrays.sort(data);
```

Will Work

```
A[] arr = new A[5];
arr[0] = new A();
arr[1] = new A();
arr[2] = new A();
arr[3] = new A();
arr[4] = new A();
Arrays.sort(arr);
```

Will Work

```
}
}
```

Parametrized Comparator

```
import java.util.*;
class A implements Comparable<A>
{
int a;
public int compareTo(A other)
{
// A a1 = (A) other; //No need of cast
if(this.a == other.a ) return 0;
if(this.a < other.a )  return -1;
return 1;
}
}
```

Parametrized Comparable

```
class ctest
{
public static void main(String args[])
{
String[] names =
{"OOP",“SPECIAL",“TOPIC"};
Arrays.sort(names);       Will Work
int[] data = { 10,-45,87,0,20,21 };
Arrays.sort(data);        Will Work

A[] arr = new A[5];
arr[0] = new A();
arr[1] = new A();
arr[2] = new A();
arr[3] = new A();
arr[4] = new A();
Arrays.sort(arr);         Will Work
}
}
```

```
import java.util.*;
class BOX implements Comparable<BOX>
{
private double l,b,h;
// Overloaded Constructors
BOX(double a)
{ l=b=h=a;
}
BOX(double l,double b,double h)
{ this.l=l; this.b=b; this.h=h;
}
// Acessor Methods
public double getL()
{  return l;
}
public double getB()
{ return b;
}
public double getH()
{ return h;
}
```

Parametrized Comparable of type BOX

Cont….

```
// area() Volume()  Methods
double area()
{
return 2*(l*b+b*h+h*l);
}
double volume()
{
return l*b*h;
}
// isEquals() method
boolean isEquals(BOX other)
{
if(this.area() == other.area()) return true;
return false;
/* OR
if(area() == other.area()) return true
return false;
*/
}
static boolean isEquals(BOX b1, BOX b2)
{
if(b1.area() == b2.area()) return true;
return false;
}
```

```
// compareTo method
public int compareTo(BOX other)
{
if(area() > other.area()) return 1;
if(area() < other.area()) return -1;
return 0;
}
public String toString()
{
String s1="length:"+l;
String s2="width:"+b;
String s3="area:"+h;
String s4="Area:"+area();
String s5="Volume:"+volume();
return s1+s2+s3+s4+s5;
}
} // End of class BOX
```

```
class comparableTest10
{
public static void main(String args[])
{
ArrayList<BOX> boxes = new ArrayList<BOX>();
boxes.add(new BOX(10));
boxes.add(new BOX(20));
boxes.add(new BOX(10,6,8));
boxes.add(new BOX(4,6,10));
boxes.add(new BOX(10,12,14));

Iterator itr = boxes.iterator();
while(itr.hasNext())
System.out.println((BOX)itr.next());

Collections.sort(boxes);

Iterator itr1 = boxes.iterator();
while(itr1.hasNext())
System.out.println((BOX)itr1.next());
}
}
```

Converting a Class To an Interface Type

1. Interface acts as a super class for the implementation classes.

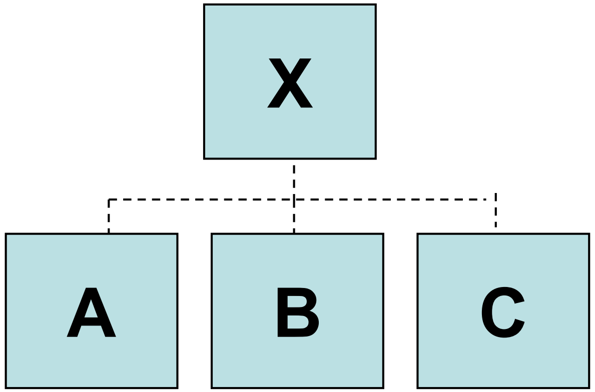
2. A reference variable belonging to type interface can point to any of the object of the classes implementing the interface.

```
A a1 = new A();

X  x1 = a1;
```

Class to interface type Conversion

<< interface >>



<< classes >>

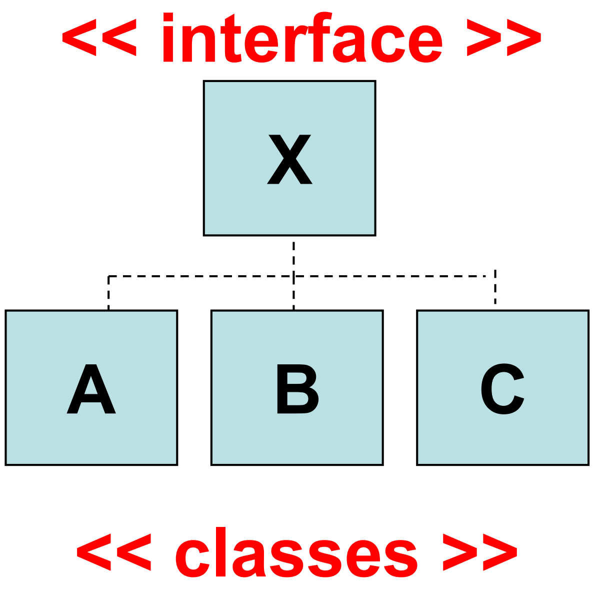
Converting an Interface to a class Type

```
X x1 = new A();

A a1 = (A) x1;


X x1 = new B();
B b1 = (B) x1;


X x1 = new C();
C c1 = (C) x1;
```



Interface to Class type Conversion

Comparator Example

- ***Supply comparators for BOX class so that BOX[] OR ArrayList<BOX> can be sorted by any of the following orders:***

1. ***Sort By Length Either in Ascending or descending order***
2. ***Sort By Width Either in Ascending or descending order***
3. ***Sort By Height Either in Ascending or descending order***
4. ***Sort By Area Either in Ascending or descending order***
5. ***Sort By Volume Either in Ascending or descending order***

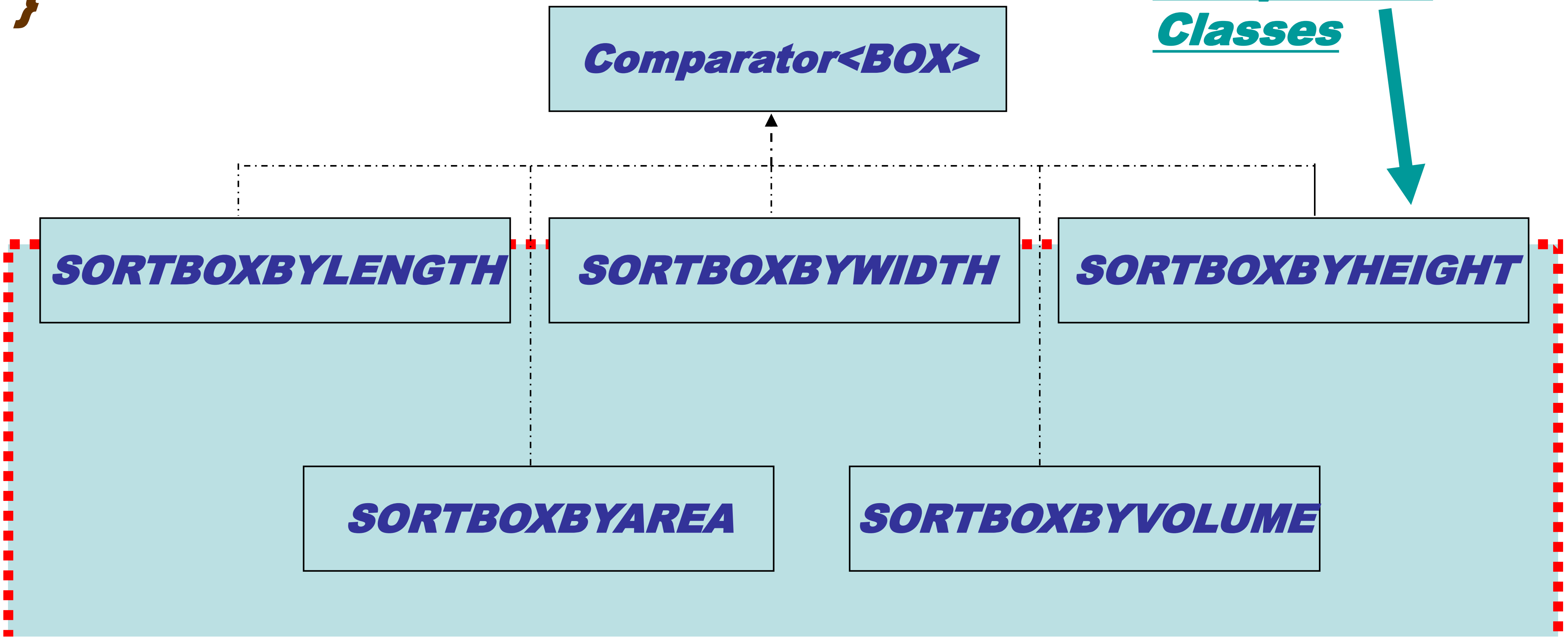
BOX is base class whose references stored either in Arrays or in Any Collection class such as ArrayList, Vector or LinkedList Needs to be sorted

```
class BOX
{
....................instance fields
...................instance methods

.........................
}
```

BOX class does not implement any comparable or comparatorinterface

Comparator Classes



```
import java.util.*;
class BOX
{
private double l,b,h;
// Overloaded Constructors
BOX(double a)
{ l=b=h=a;
}
BOX(double l,double b,double h)
{
this.l=l;
this.b=b;
this.h=h;
}
// Acessor Methods
public double getL()
{ return l;
}
public double getB()
{ return b;
}
public double getH()
{ return h;
}
```

```
// area() Volume()  Methods
double area()
{
return 2*(l*b+b*h+h*l);
}
double volume()
{
return l*b*h;
}
// isEquals() method
boolean isEqual(BOX other)
{
if(this.area() == other.area()) return true;
return false;
/* OR
if(area() == other.area()) return true
return false;
*/
}
```

Cont …..

```
static boolean isEquals(BOX b1, BOX b2)
{
if(b1.area() == b2.area()) return true;
return false;
}
public String toString()
{
String s1="length:"+l;
String s2="width:"+b;
String s3="area:"+h;
String s4="Area:"+area();
String s5="Volume:"+volume();
return s1+s2+s3+s4+s5;
}
} // End of class BOX
```

NOTE :

BOX class is base class whose references needs to be sorted. It does not implement either comparable or comparator class

Cont .....

// Comparator class for Sorting by BOX references By length

```
class SORTBOXBYLENGTH implements Comparator<BOX>
{
private int order; // Defines Order of sorting 1 for Ascending -1 for Descending
SORTBOXBYLENGTH(boolean isAscending)
{
if(isAscending)
order =1;
else
order =-1;
}
public int compare(BOX b1,BOX b2)
{
if(b1.getL() > b2.getL()) return 1*order;
if(b1.getL() < b2.getL()) return -1*order;
return 0;
}
}// End of class
```

// Comparator class for Sorting by BOX references By Width

```
class SORTBOXBYWIDTH implements Comparator<BOX>
{
private int order;
SORTBOXBYWIDTH(boolean isAscending)
{
if(isAscending)
order =1;
else
order =-1;
}
public int compare(BOX b1,BOX b2)
{
if(b1.getB() > b2.getB()) return 1*order;
if(b1.getB() < b2.getB()) return -1*order;
return 0;
}
} // End of class
```

Comparator class for Sorting by BOX references By Height

```
class SORTBOXBYHEIGHT implements Comparator<BOX>
{
private int order;
SORTBOXBYHEIGHT(boolean isAscending)
{
if(isAscending)
order =1;
else
order =-1;
}
public int compare(BOX b1,BOX b2)
{
if(b1.getH() > b2.getH()) return 1*order;
if(b1.getH() < b2.getH()) return -1*order;
return 0;
}
} // End of class
```

Comparator class for Sorting by BOX references By Area

```
class SORTBOXBYAREA implements Comparator<BOX>
{
private int order;
SORTBOXBYAREA(boolean isAscending)
{
if(isAscending)
order =1;
else
order =-1;
}
public int compare(BOX b1,BOX b2)
{
if(b1.area() > b2.area()) return 1*order;
if(b1.area() < b2.area()) return -1*order;
return 0;
}
} // End of class
```

Comparator class for Sorting by BOX references By Volume

```
class SORTBOXBYVOLUME implements Comparator<BOX>
{
private int order;
SORTBOXBYVOLUME(boolean isAscending)
{
if(isAscending)
order =1;
else
order =-1;
}
public int compare(BOX b1,BOX b2)
{
if(b1.volume() > b2.volume()) return 1*order;
if(b1.volume() < b2.volume()) return -1*order;
return 0;
}
} // End of class
```

```
class comparatorTest
{
public static void main(String args[]) {
ArrayList<BOX> boxes = new ArrayList<BOX>();
boxes.add(new BOX(10));
boxes.add(new BOX(20));
boxes.add(new BOX(10,6,8));
boxes.add(new BOX(4,6,10));
boxes.add(new BOX(10,12,14));

// SORT BY LENTH ORDER:Ascending
Comparator<BOX> c1 = new SORTBOXBYLENGTH(true);
Collections.sort(boxes,c1);
for(int i=0;i<boxes.size();i++)
System.out.println(boxes.get(i));
System.out.println("");

// SORT BY LENTH ORDER:Descending
c1 = new SORTBOXBYLENGTH(false);
Collections.sort(boxes,c1);
for(int i=0;i<boxes.size();i++)
System.out.println(boxes.get(i));
System.out.println("");
// SORT BY Volume ORDER:Ascending
c1 = new SORTBOXBYVOLUME(true);
Collections.sort(boxes,c1);
for(int i=0;i<boxes.size();i++)
System.out.println(boxes.get(i));
System.out.println("");
// SORT BY Volume ORDER:Descending
c1 = new SORTBOXBYVOLUME(false);
Collections.sort(boxes,c1);
for(int i=0;i<boxes.size();i++)
System.out.println(boxes.get(i));
System.out.println("");
}
} // End of Main class
```

OUTPUT

```
length:4.0width:6.0area:10.0Area:248.0Volume:240.0
length:10.0width:10.0area:10.0Area:600.0Volume:1000.0
length:10.0width:6.0area:8.0Area:376.0Volume:480.0
length:10.0width:12.0area:14.0Area:856.0Volume:1680.0
length:20.0width:20.0area:20.0Area:2400.0Volume:8000.0
```

```
length:20.0width:20.0area:20.0Area:2400.0Volume:8000.0
length:10.0width:10.0area:10.0Area:600.0Volume:1000.0
length:10.0width:6.0area:8.0Area:376.0Volume:480.0
length:10.0width:12.0area:14.0Area:856.0Volume:1680.0
length:4.0width:6.0area:10.0Area:248.0Volume:240.0
```

```
length:4.0width:6.0area:10.0Area:248.0Volume:240.0
length:10.0width:6.0area:8.0Area:376.0Volume:480.0
length:10.0width:10.0area:10.0Area:600.0Volume:1000.0
length:10.0width:12.0area:14.0Area:856.0Volume:1680.0
length:20.0width:20.0area:20.0Area:2400.0Volume:8000.0
```

```
length:20.0width:20.0area:20.0Area:2400.0Volume:8000.0
length:10.0width:12.0area:14.0Area:856.0Volume:1680.0
length:10.0width:10.0area:10.0Area:600.0Volume:1000.0
length:10.0width:6.0area:8.0Area:376.0Volume:480.0
length:4.0width:6.0area:10.0Area:248.0Volume:240.0
```

THANK YOU

Dr. N MEHALA

Department of Computer Science and Engineering

mehala@pes.edu

Abstract Classes

1. An abstract class is a class that has at least one *abstract method* (i.e a method with only heading with no body of executable statements)
2. We can not create an object of abstract classes i.e abstract class objects can not be instantiated
3. An abstract class needs to be extended by sub classes to provide the implementation for the abstract methods.
4. Abstract classes may contain static methods
5. abstract and static keyword combination is wrong

   abstract static void print(); wrong
6. Abstract classes may extend either another abstract class or concrete class
7. Abstract classes may include constructors, nested classes and interfaces
8. Abstract classes has either public, protected, private or package accessibility

Abstract Classes

- Syntax :

```
 abstract class <classname>
{

  ……………………………

  abstract <return type> methodname(<parameter List>);
  abstract <return type> methodname(<parameter List>);
}
```

Note:

1. Abstract Class should have atleast one abstract method
2. Abstract classes may extend another class , implements another interface , may have concrete methods

Example

```
abstract class A
{
private int a;
void display()
{
System.out.println("Concrete Method of class A");
}
abstract void show();
}
```

Abstract method without body

Abstract declaration is must for both class as well as method

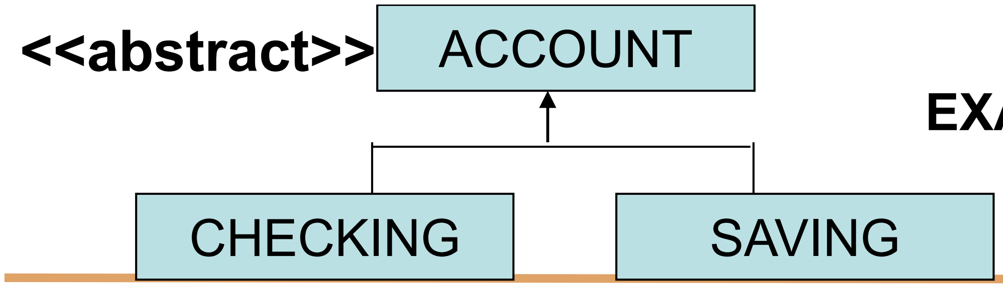
Example 2

```
class A
{
}
abstract class B extends A
{
private int a;
void display()
{
System.out.println("Concrete Method of class A");
}
abstract void show();
}
```

A is Complete Class

B is abstract class extending a complete class

Abstract class either extends a complete class or an abstract class



<<abstract>> ACCOUNT

CHECKING

SAVING

EXAMPLES ABSTRACT CLASS

```
abstract class Account
{
private String name;
private String actno;
private double balance;
private Address addr;

// Overloaded Constructors
Account(String n,String a)
{
name = n;
actno= a;
balance = 0.0;
}
Account(String n,String a,double b)
{
name = n;
actno= a;
balance = b;
}
```

```
// Accessor Methods
String getName()  { return name;}
String getactno() { return actno;}
double getbalance() { return balance;}

// Mutator Method only for balance
void setbalance(double amount)
{  this.balance = amount;}
void showAccountDetails()
{
System.out.println("Name :"+this.getName());
System.out.println("Account No
:"+this.getactno());
System.out.println("Balance
:"+this.getbalance());
}
// provide abstract methods
abstract double withdraw(double amount);
abstract void deposit(double amount);
} // END OF Account CLASS
```

```
class Saving extends Account
{
Saving(String n,String a)
{
super(n,a);
System.out.println("Saving Account Created");
System.out.println("Name :"+this.getName());
System.out.println("Account No :"+this.getactno());
System.out.println("Balance :"+this.getbalance());
showAccountDetails();
}
Saving(String n,String a,double b)
{
super(n,a,b);
System.out.println("Saving Account Created");
System.out.println("Name :"+this.getName());
System.out.println("Account No :"+this.getactno());
System.out.println("Balance :"+this.getbalance());
showAccountDetails();
}
double withdraw(double amount)
{
 /*
 if( balance == 0) return 0.0;
 if( balance < amount ) return 0.0;
 balance = balance - amount;
*/

 if(this.getbalance() == 0) return 0.0;
 if(this.getbalance() < amount ) return 0.0;
 setbalance(getbalance() - amount);
 return amount;
}
void deposit(double amount)
{
 setbalance(getbalance() + amount);
 return ;
}
}//end of Saving class
```

```
class Checking extends Account
{
Checking(String n,String a,double b)
{
super(n,a,b);
System.out.println("Checking Account
Created");
showAccountDetails();
}
double withdraw(double amount)
{
/*
 if( balance - 100 == 0) return 0.0;
 if( balance -100 < amount ) return 0.0;
 balance = balance - amount;
*/

 if(this.getbalance() - 100 == 0) return 0.0;
 if(this.getbalance() - 100 < amount ) return 0.0;
 setbalance(this.getbalance() - amount);
 return amount;
}
```

```
void deposit(double amount)
{
 setbalance(this.getbalance() + 0.9 *
amount)  ;
 return ;
}
}//end of Checking class
```

```
class AccountTest
{
public static void main(String args[])
{
Checking c1 =  new Checking("Rahul Sharma","C106726",100000);
Checking c2 =  new Checking("Raman Kumar","C106727",100000);

Saving s1 =  new Saving("Kumar Sharma","S106726",100000);
Saving s2 =  new Saving("Mohan Lal","S106727");

c1.withdraw(2000);
c1.showAccountDetails();
c2.deposit(10000);
c2.showAccountDetails();
s1.deposit(900);
s1.showAccountDetails();
s2.withdraw(400);
s2.showAccountDetails();
}
}
```

1) Abstract class must have only abstract methods. True or false?

False. Abstract methods can also have concrete methods.

2) Is it compulsory for a class which is declared as abstract to have at least one abstract method?

Not necessarily. Abstract class may or may not have abstract methods.

3) Can we use “abstract” keyword with constructor, Instance Initialization Block and Static Initialization Block?

No. Constructor, Static Initialization Block, Instance Initialization Block and variables can not be abstract.

4) Why final and abstract can not be used at a time?

Because, final and abstract are totally opposite in nature. A final class or method can not be modified further where as abstract class or method must be modified further. “final” keyword is used to denote that a class or method does not need further improvements. “abstract” keyword is used to denote that a class or method needs further improvements.

5) Can we instantiate a class which does not have even a single abstract methods but declared as abstract?

No, We can’t instantiate a class once it is declared as abstract even though it does not have abstract methods.

6) Can we declare abstract methods as private? Justify your answer?

No. Abstract methods can not be private. If abstract methods are allowed to be private, then they will not be inherited to sub class and will not get enhanced.

7) We can’t instantiate an abstract class. Then why constructors are allowed in abstract class?

It is because, we can’t create objects to abstract classes but we can create objects to their sub classes. From sub class constructor, there will be an implicit call to super class constructor. That’s why abstract classes should have constructors. Even if you don’t write constructor for your abstract class, compiler will keep default constructor.

8) Can we declare abstract methods as static?
No, abstract methods can not be static.

9) Can a class contain an abstract class as a member?
Yes, a class can have abstract class as it’s member.

10) Can abstract method declaration include throws clause?
Yes. Abstract methods can be declared with throws clause.

PES UNIVERSITY ONLINE

THANK YOU

Dr. N MEHALA

Department of Computer Science and Engineering

mehala@pes.edu

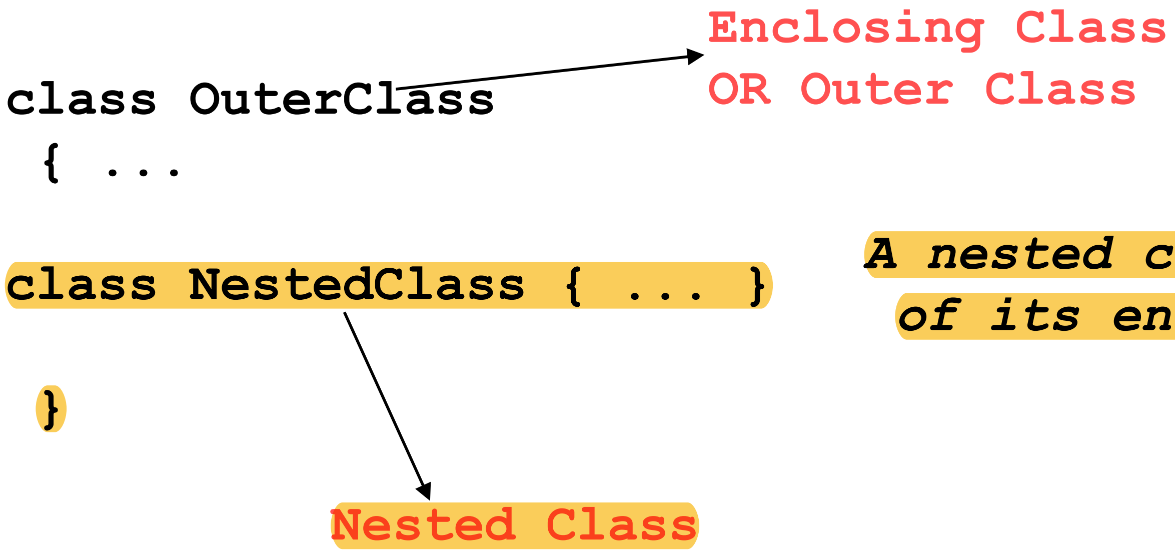
OBJECT ORIENTED PROGRAMMING WITH JAVA

Dr. N MEHALA

Department of Computer Science and Engineering

Nested Classes

Java programming language allows you to define a class within another class



```
class OuterClass
 { ...

class NestedClass { ... }

 }
```

Enclosing Class
OR Outer Class

Nested Class

A nested class is a member of its enclosing class

1. Nested has access to other members of the enclosing class, even if they are declared private

2. Can be private, public, protected or friendly access

Nested Class Types

- Static nested classes

1. Static keyword applied for class declaration
2. Static nested class can use the instance fields/methods of the outer class only through object reference.
3. Static nested class can be accessed

   OuterClass.StaticNestedClass

4. To create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

Nested Class Types cont..

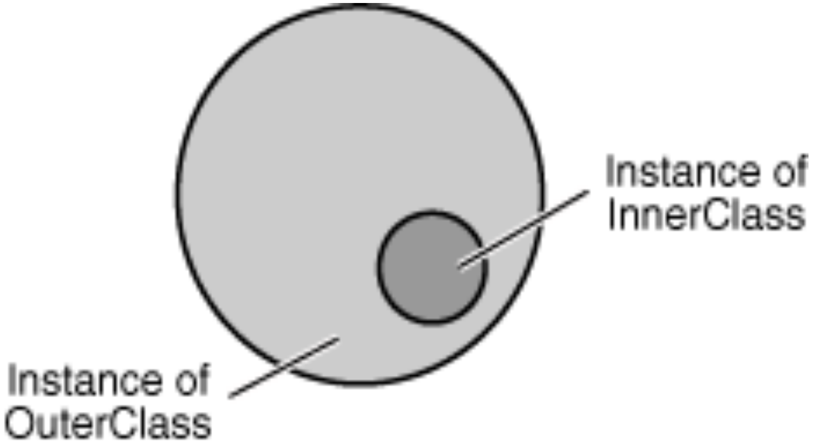
- Non-Static nested classes

1. These nested classes do not have ***static*** keyword applied
2. Non-Static nested class can use the instance fields/methods of the outer class directly.

3. To create an object for the non-static nested class, use this syntax:

OuterClass.NestedClass nestedObject = Outerobjectreference. new innerclass();

Inner class instance can only exists inside Outer class instance.



Example 1 [Non-static Nested Class]

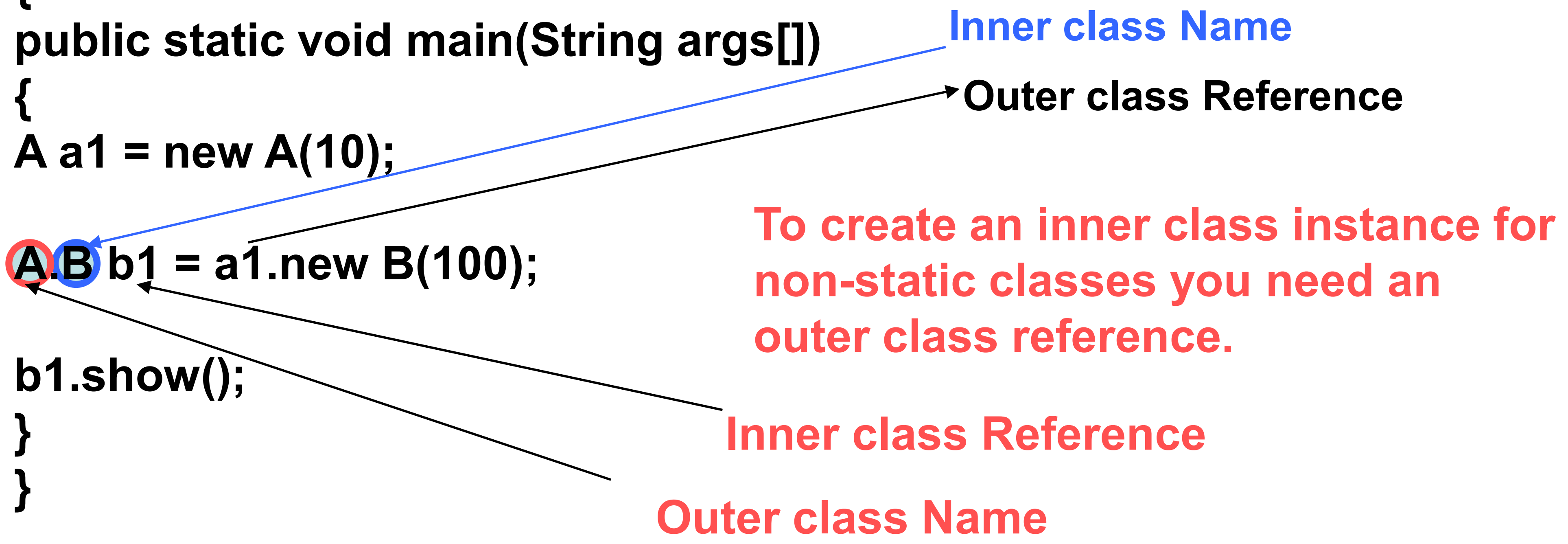
```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void print()
{
System.out.println("a="+a);
}
class B
    {
    int b;
    B(int b)
    {
    int c = b+10;
    this.b = c;
    }
    void show()
    {
    print();
    System.out.println("b="+b);
    }
    } // End of class B
} // End of class A
```

Outer Class

Nested class with friendly access

Call to print() of outer class

Example 1 [Non-static Nested Class] cont….



```
class innertest1
{
public static void main(String args[])
{
A a1 = new A(10);

A.B b1 = a1.new B(100);

b1.show();
}
}
```

To create an inner class instance for non-static classes you need an outer class reference.

If class B is Private then it is not visible in main().

A.B b1 = a1.new B(100); is WRONG/INVALID

Example 2

```
class A
{
private int a;
private int b=10;
```

Outer class

```
A(int a)
{
this.a=a;
}
```

Nested Inner class [Non static Type]

```
class B
{
private int b;
B(int b)
{
this.b =b;
}
void show()
{
int b=20;
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("this.b="+this.b);
System.out.println("Outer b="+A.this.b);
}
} // End of B inner class
```

Instance Field of B

Outer Class A's a

Local b

B's instance Field b

A's instance Field b

```
void show()
{
B b1 = new B(30);
b1.show();
}
} // End of Outer class A
```

```
class innerTest
{
public static void main(String args[])
{
// Create an inner class B's instance
// Call show() method

// STEP 1
// Create an Outer Instance first

A a1 = new A(20);
A.B b1 = a1.new B(-30);
b1.show();
```

```
a=20
b=20
this.b=-30
Outer b=10
```

```
// inner class object instantiation thru anonymous outer
// reference
A.B b2 = new A(30).new B(-40);
b2.show();
}
}
```

```
a=30
b=20
this.b=-40
Outer b=10
```

Static Inner class / Static Nested class Example

```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void print()
{
System.out.println("a="+a);
}
static class B
{
int b;
B(int b)
{
int c = b+10;
this.b = c;
}
void show()
{
// print();  INVALID
A a1 = new A(10);
a1.print();
System.out.println("b="+b);
}
} // End of class B
} // End of class A
```

Static inner class

Static nested class can refere to outer members only through outer reference

Example cont....

```
class innertest10
{
public static void main(String args[])
{
A.B b1 = new A.B(100);
b1.show();
}
}
```

Instance of static Inner class

Static Nested class Example 2

```
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
public void show()
{
System.out.println("a="+a);
display();
}
public static void display()
{
System.out.println("b="+b);
}
```

Example 2 cont....

```
static class B
{
private int a;
~~protected static int b=100;~~
B(int a)
{
this.a=a;
}
void show()
{
// A.this.show(); // Won't work show() is non-static in outer
display(); // Will work as method is static in outer
System.out.println("a="+a);
// System.out.println("a="+A.this.a);
// Won't work a is non-static in outer
System.out.println("b="+b); // Will refer to its own b
System.out.println("A'sb="+A.b);  // will refer to outer class B

new A(40).show();
// This is how you can call non static methods of outer

}
} // End of inner class B
} // End of class A
```

Example 2 cont….

```
class innerTest1
{
public static void main(String args[])
{
A.B b1 = new A.B(-30);
b1.show();
}
}
```

```
D:\jdk1.3\bin>java innerTest1
b=10
a=-30
b=100
A'sb=10
a=40
b=10
```

Local Inner classes [Classes Within method body]

Class declared within a method body.
Here method is show()
Local inner classes Can not be declared as public,private or protected

```
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
void show()
{
          class B
          {}
}
}
```

1. Class B is visible only in method show().
2. It can be used within this show() method only
3. Local inner classes can only use final variables from its enclosing method.
4. However inner classes can refer to its fields of enclosing class.

```
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
void show()
{
int x=10;
        class B
        {
        private int b;
        B(int b)
        {
        this.b=b;
        }
        void display()
        {
        System.out.println("a="+a);
        System.out.println("b="+b);
        System.out.println("x="+x);
        }
        } // End of class B
} // End of show() method
} // End of A class
```

```
D:\jdk1.3\bin>javac
innerTest2.java
innerTest2.java:23: local
variable x is accessed from
within inner class;
to be declared final
System.out.println("x="+x);
                        ^

1 error
```

Reference for A's a

Reference for B's b

Reference is wrong / errorneous

'x' is local variable inside the local method. Local classes can use only final fields from enclosing method

```
class innertest
{
public static void main(String
args[])
{
final int a1=10;
```

```
new A(20).show();
print();
}// End of main
static void print()
{
/*
A a1 = new A(30);
a1.show();
*/
System.out.println("Hello");
}
}
```

```
class A
{
private int a;
private int b;
int c;
A(int a)
{
this.a =a;
b = a+20;
c = a+40;
}
void show()
{
System.out.println("a1="+a1)
;
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
}
} //End of A
```

OUTPUT

```
E:\oop>java innertest
a1=10
a=20
b=40
c=60
Hello
```

Anonymous Inner classes

- ***Another category of local inner classes***
- ***Classes without any name i.e classes having no name***
- ***Can either implements an interface or extends a class.***
- ***Can not have more than one instance active at a time.***
- ***Whole body of the class is declared in a single statement ending with ;***

Cont…

- Syntax [If extending a class]

```
[variable_type_superclass =]  new superclass_name()  {
                                        // properties and methods
                                 }  [;]
```

- Syntax [If implementing an interface]

```
[variable_type_reference =]  new reference_name()  {
                                        // properties and methods
                                 }  [;]
```

Anonymous Inner Class Example

```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void show()
{
System.out.println("a="+a);
} // End of show()
}// End of class A
```

Anonymous inner class extending super class A

```
class innertest1
{
public static void main(String args[])
{
A a1 = new A(20){
      public void show()
      {
      super.show();
      System.out.println("Hello");
      }
      public void display()
      {
      System.out.println("Hi");
      }
      };
a1.show();
// a1.display();
}
}
```

Calling show from inner class

```
interface X
{
int sum(int a,int b);
int mul(int x,int y);
}
class innertest2
{
public static void main(String args[])
{
```

Anonymous inner class implementing an interface

```
X x1 = new X()
        {
        public int sum(int a,int b)
        {
        return a+b;
        }
        public int mul(int a,int b)
        {
        return a*b;
        }
    };
System.out.println(x1.sum(10,20));
System.out.println(x1.mul(10,20));
}// End of main
}// End of innertest2
```

Home Exercise

- Write 5 BOX Comparator classes using anonymous inner classes.

PES UNIVERSITY ONLINE

THANK YOU

Dr. N MEHALA

Department of Computer Science and Engineering

mehala@pes.edu