Polymorphism & Abstraction & Interface & packages

Method Overriding

```java
class Base {
    int a;
    Base(int a) {
        this.a = a;
    }
    void show() {
        System.out.println(a);
    }
}

class Child extends Base {
    int b;

    Child(int a, int b) {
        super(a);
        this.b = b;
    }

    // the following method overrides Base class's show()
    @Override // this is an annotation (optional but recommended)
    void show() {
        System.out.println(a + ", " + b);
    }
}
```

```java
public class MethodOverride {
    public static void main(String[] args) {
        Child o = new Child( a: 10,  b: 20);
        o.show();
        Base b = o;
        b.show(); // will call show of Override
    }
}
```

P : MethodOverride.java

Question-1:

```
3  class X {
4    int a;
5
6    X(int i) { a = i; }
7  }
8
9  class Y {
10   int a;
11
12   Y(int i) { a = i; }
13 }
14
15 class TestClass {
16   public static void main(String[] args) {
17     X x = new X(10);
18     X x2;
19     Y y = new Y(5);
20
21     x2 = x;
22
23     x2 = y;     // Error, not of same type
24   }
25 }
```

Question-2

```
2  class X
3  {
4    int a;
5
6    X(int i) { a = i; }
7  }
8
9  class Y extends X
10 {
11   int b;
12
13   Y(int i, int j)
14   {
15     super(j);
16     b = i;
17   }
18 }
```

```
20 class SupSubRef2 {
21   public static void main(String[] args)
22   {
23     X x = new X(10);
24     X x2;
25     Y y = new Y(5, 6);
26
27     x2 = x; // OK, both of same type
28     System.out.println("x2.a: " + x2.a);
29
30     x2 = y;
31     System.out.println("x2.a: " + x2.a);
32
33     x2.a = 19;
34   }
35 }
```

```
x2.a: 10
x2.a: 6
```

Dynamic Method Dispatch

- Mechanism by which a call to overridden method is resolved at run time, rather than at compile time
  - Basis for run-time polymorphism
- Principle used: A superclass reference variable can refer to a subclass object
  - When a overridden method is called through a superclass reference, Java determines which version of that method to execute at that time
  - Decision is made based on the type of the object being referred to and not on the type the reference variable

Dynamic Method Dispatch

- Upcasting: Reference variable of superclass referring to object of subclass
  - Example:

```
class A { }

class B extends A { }

In main(): A obj = new B();      // upcasting
```

```java
// Dynamic Method Dispatch
class A
{
     void callme()
     {
      System.out.println("A's callme method");
     }
}

class B extends A
{
  void callme() // override callme()
  {
    System.out.println("B's callme method");
  }
}

class C extends A {

  void callme() // override callme()
  {
    System.out.println("C's callme method");
  }
}
```

```
26  class Dispatch
27  {
28    public static void main(String args[])
29    {
30      A a = new A(); // object of type A
31      B b = new B(); // object of type B
32      C c = new C(); // object of type C
33      A r; // obtain a reference of type A
34
35      r = a; // r refers to an A object
36      r.callme(); // calls A's version of callme
37
38      r = b; // r refers to a B object
39      r.callme(); //
40
41      r = c; // r re
42      r.callme(); //
43    }
44  }
```

```
A's callme method
B's callme method
C's callme method
```

```
Rate of interest
Bank 1: 8
Bank 2: 7
```

```
1  class Bank
2  {
3      int getRate() {ret
4  }
```

```
6  class Bank1 extends Ba
7  {
8      int getRate()
9      { return 8; }
10 }
```

```
15     { return 7; }
16 }
```

```
18 class BankTest
19 {
20     public static void main(String args[])
21     {
22         Bank b;
23         b = new Bank1();
24         System.out.println ("Rate of interest");
25         System.out.println ("Bank 1: "+ b.getRate());
26         b = new Bank2();
27         System.out.println ("Bank 2: "+ b.getRate());
28     }
29 }
```

Dynamic Method Dispatch

```
3  class P {
4      void call() {
5          System.out.println("Inside P's call method");
6      }
7  }
8  class Q extends P {
9      void call() {
10         System.out.println("Inside Q's call method");
11     }
12 }
13 class R extends Q {
14     void call() {
15         System.out.println("Inside R's call method");
16     }
17 }
18
19 public class DynamicDispatchTest {
20     public static void main(String[] args) {
21         P p = new P(); // object of type P
22         Q q = new Q(); // object of type Q
23         R r = new R(); // object of type R
24         P  x;          // reference of type P
25         x = p;         // x refers to a P object
26         x.call();      // invoke P's call
27         x = q;         // x refers to a Q object
28         x.call();      // invoke Q's call
29         x = r;         // x refers to a R object
30         x.call();      // invoke R's call
31     }
32 }
```

- **DMD is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.**
- **DMD is a way Java implement run time polymorphism.**

P : DynamicDispatchTest.java
FindAreas.java

Abstract Class

- Abstract class is a class that cannot be instantiated
- Its like a Superclass declares the structure of a given abstraction without providing a complete implementation of every method
  - Defines a generalized form that will be shared by all its subclasses, leaving it to each subclass to fill in the details
- To specify that certain methods must be overridden by subclasses, specify *abstract* type modifier with superclass
  - No implementation in superclass
  - Subclass' responsibility to implement them
  - Syntax of declaring an abstract method:

abstract type_name (parameter-list);

Abstract Class

- Any class that contains one or more abstract methods must be declared abstract
  - use ***abstract*** keyword before the keyword class
  - Cannot create objects of abstract class because such objects are of no use
  - Cannot declare abstract constructors
  - Cannot have abstract static methods
- Any subclass of an abstract class must either implement all abstract methods specified in the superclass or be itself an abstract class

Abstract Class

Note:

- ***A non-abstract class is called a concrete class***

  abstract class A

- Abstract class contains abstract method

  abstract method f()

- No instance can be created of an abstract class
- The subclass must implement the abstract method
- Otherwise the subclass will be an abstract class too

```java
// A Simple demonstration of abstract.
abstract class A
{
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo()
  {
    System.out.println("This is a concrete method.");
  }
}

class B extends A
{
  void callme()
  {
    System.out.println("B's implementation of callme.");
  }
}
```

```
21  class AbstractDemo
22  {
23    public static void main(String args[])
24    {
25      B b = new B();
26
27      b.callme();
28      b.callmetoo();
29    }
30  }
```

```
B's implementation of callme.
This is a concrete method.
```

Abstract Class

```
3  abstract class S {
4      // abstract method
5      abstract void call();
6      // concrete methods are still allowed in abstract classes
7      void call2() {
8          System.out.println("This is a concrete method");
9      }
10 }
11
12 class T extends S {
13     void call() {
14         System.out.println("T's implementation of call");
15     }
16 }
17
18 class AbstractDemo {
19     public static void main(String args[]) {
20         //S s = new S(); // S is abstract; cannot be instantiated
21         T t = new T();
22         t.call();
23         t.call2();
24     }
25 }
```

P: AbstractAreas.java
AbstractDemo.java

Question 1

```
1  abstract class A
2  {
3    abstract void Method1();
4    abstract void Method2();
5  }
6
7  class B extends A
8  {
9    void Method1()
10   {
11     System.out.println("B's implementation of Method1()");
12   }
13 }
14
15 class AbstractTest
16 {
17   public static void main(String args[])
18   {
19     B b = new B();
20     b.Method1();
21   }
22 }
```

Solution:

```
1  abstract class A
2  {
3    abstract void Method1();
4    abstract void Method2();
5  }
6  class B extends A
7  {
8    void Method1()
9    {
10     System.out.println("B's implementation of Method1()");
11   }
12   void Method2()
13   {
14     System.out.println("B's implementation of Method2()");
15   }
16 }
17
18 class AbstractTest
19 {
20   public static void main(String args[])
21   {
22     B b = new B();
23     b.Method1();
24   }
25 }
```

Question 2:

```
1  abstract class A
2  {
3    abstract void Method1();
4    abstract void Method2();
5  }
6
7  class B extends A
8  {
9    void Method1()
10   {
11     System.out.println("B's implementation of Method1()");
12   }
13 }
14
15 class AbstractTest
16 {
17   public static void main(String args[])
18   {
19
20   }
21 }
```

Error: B is not abstract and does not override abstract method Method2()

Solution:

```
1  abstract class A
2  {
3    abstract void Method1();
4    abstract void Method2();
5  }
6
7  abstract class B extends A
8  {
9    void Method1()
10   {
11     System.out.println("B's implementation of Method1()");
12   }
13 }
14
15 class AbstractTest
16 {
17   public static void main(String args[])
18   {
19
20   }
21 }
```

Question-3:

```
1   class A
2   {
3     abstract void Method1();
4     abstract void Method2();
5   }
6
7   class Test
8   {
9     public static void main(String args[])
10    {
11
12    }
13  }
```

Error: A is not abstract and does not override abstract method Method2() in A

Solution:

```
1   abstract class A
2   {
3     abstract void Method1();
4     abstract void Method2();
5   }
6
7   class Test
8   {
9     public static void main(String args[])
10    {
11
12    }
13  }
```

Run-time polymorphism

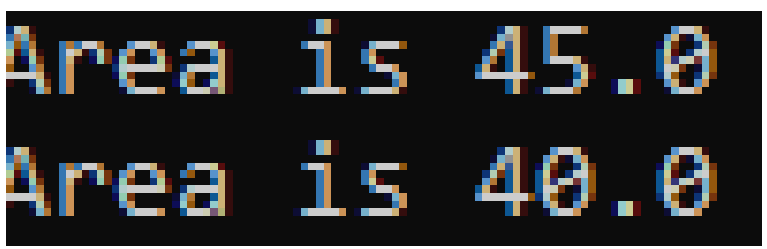
- Not possible to instantiate objects of Abstract classes; but, object references can be created
- Run-time polymorphism implemented through the use of superclass references
- Possible to create a reference to an abstract class so that it can be used to point to a subclass object

Example: Polymorphism in Figure class

Example: Figure - Abstract class

```
1  abstract class Figure
2  {
3    double dim1, dim2;
4
5    Figure(double a, double b)
6    {  dim1 = a;  dim2 = b; }
7
8    abstract double area();
9  }
10 class Rectangle extends Figure
11 {
12   Rectangle(double a, double b)
13   { super(a, b); }
14
15   double area() // override area for rectangle
16   {
17     return dim1 * dim2;
18   }
19 }
20  class Triangle extends Figure
21  {
22    Triangle(double a, double b)
23    {  super(a, b); }
24
25    double area() // override area for right triangle
26    { return dim1 * dim2 / 2; }
27  }
```

```
29  class AbstractAreas
30  {
31    public static void main(String args[])
32    {
33     // Figure f = new Figure(10, 10); // illegal now
34      Rectangle r = new Rectangle(9, 5);
35      Triangle t = new Triangle(10, 8);
36
37      Figure figref; // this is OK, no object is created
38
39      figref = r;
40      System.out.println("Area is " + figref.area());
41
42      figref = t;
43      System.out.println("Area is " + figref.area());
44    }
45  }
```



Example: Shape - Abstract class

```
1  abstract class Shape
2  {
3      abstract double area();
4  }
5  class Rectangle extends Shape
6  {
7    double length, width;
8
9    Rectangle(double l, double w)
10   { length = l; width = w; }
11
12   double area()  // override area for rectangle
13   {  return length * width;  }
14 }
15 class Triangle extends Shape
16 {
17   double base , height;
18
19   Triangle( double b, double h)
20   {  base = b; height = h; }
21
22   double area() // override area for right triangle
23   { return base * height / 2; }
24 }
```

```
26  class DynamicShapes
27  {
28    public static void main(String args[])
29    {
30      Rectangle r = new Rectangle(4, 5);
31      Triangle t = new Triangle(8, 5);
32
33      Shape[] shapes = { r , new Triangle(10, 8), t };
34
35      for( Shape s : shapes )
36          System.out.println("Area="+s.area());
37    }
38  }
```

```
Area=20.0
Area=40.0
Area=20.0
```

Final

- Declare a final variable, prevents its contents from being modified
- final variable must initialize when it is declared
- It is common coding convention to choose all uppercase identifiers for final variables

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Using final with Inheritance

To prevent overriding

```
class A {
    final void f() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void f() { // Error! Can't override.
        System.out.println("Illegal!");
    }
}
```

To prevent inheritance

```
final class A {
    //...
}

// The following class is illegal.
class B extends A { // Error! Can't subclass A
    //...
}
```

Nested / Inner Classes

Nested Classes

- It is possible to define a class within another classes, such classes are known as nested classes
- The scope of nested class is bounded by the scope of its enclosing class. That means if class B is defined within class A, then B doesn’t exists without A
- The nested class has access to the members (including private!) of the class in which it is nested
- The enclosing class doesn’t have access to the members of the nested class

Nested Classes

```
1  class Outer{
2      int a;
3      public void show() {
4          System.out.println("print outer class");
5      }
6
7      class Inner{
8          public void display() {
9              System.out.println("print inner class");
10         }
11     }
12 }
13 public class InnerDemo {
14
15     public static void main(String[] args) {
16         Outer out = new Outer();
17         out.show();
18 //      Inner inn = new Inner();
19         Outer.Inner inn = out.new Inner();
20         inn.display();
21
22     }
23
24 }
25
```

Nested Classes

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

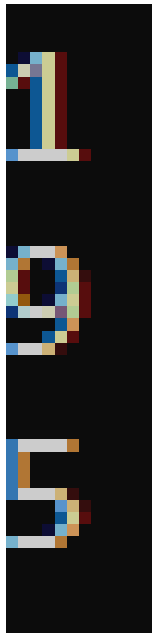
Output from this application is shown here:

```
display: outer_x = 100
```

```
1  class Outer
2  {
3    int[] nums;
4
5    Outer(int[] n)
6    {
7      nums = n;
8    }
9
10   void analyze()
11   {
12     Inner inOb = new Inner();
13
14     System.out.println( inOb.min() );
15     System.out.println( inOb.max() );
16     System.out.println( inOb.avg() );
17   }
```

```
19   class Inner
20   {
21     int min()
22     {
23       int m = nums[0];
24
25       for(int i=1; i < nums.length; i++)
26         if(nums[i] < m) m = nums[i];
27       return m;
28     }
29
30     int max()
31     {
32       int m = nums[0];
33       for(int i=1; i < nums.length; i++)
34         if(nums[i] > m) m = nums[i];
35       return m;
36     }
```

```
38        int avg()
39        {
40          int a = 0;
41          for(int i=0; i < nums.length; i++)
42            a += nums[i];
43          return a / nums.length;
44        }
45      }
46    }
47    class NestedClassDemo
48    {
49      public static void main(String[] args)
50      {
51        int[] x = { 3, 2, 1, 5, 6, 9, 7, 8 };
52        Outer outOb = new Outer(x);
53        outOb.analyze();
54      }
55    }
```



Inner Classes

```java
class Outer1
{
    private int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }
    // this is an inner class
    class Inner {
        void display() {
            System.out.println(outer_x);
        }
    }
}

public class InnerClassDemo1 {
    public static void main(String[] args) {
        Outer1 outer = new Outer1();
        outer.test();
        Outer1.Inner innerObj = outer.new Inner();
        innerObj.display();
    }
}
```

It can access private members of an outer class

Inner Classes

```
1  class Outer2
2  {
3      int outer_x = 100;
4
5      void test() {
6          Inner inner = new Inner();
7          inner.display();
8      }
9
10     class Inner {
11         int y = 10; // y is local to Inner
12         void display() { System.out.println(outer_x); }
15     }
16
17     void showy() {
18         //System.out.println(y); // error, y not known here!
19     }
20 }
21
22 public class InnerClassDemo2 {
23     public static void main(String[] args) {
24         Outer2 outer = new Outer2();
25         outer.test();
26     }
27 }
```

Y is not in the scope of outer class

Object Class

- The Object class, in the java.lang package, is at the top of the class hierarchy. Every class is a descendant, direct or indirect, of the Object class.
- That is, Object is a superclass of all other classes
- This means that a reference variable of type Object can refer to an object of any other class
- Also, since arrays are implemented as classes, a variable of type Object can also refer to any array

Object Class Methods

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Methods of Object class

protected Object clone() throws CloneNotSupportedException

Creates and returns a copy of this object.

public boolean equals(Object obj)

Indicates whether some other object is "equal to" this one.

protected void finalize() throws Throwable

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object

public final Class getClass()

Returns the runtime class of an object.

public int hashCode()

Returns a hash code value for the object.

public String toString()

Returns a string representation of the object.

Interfaces

Interfaces

Meaning of Interface

- Java does not support multiple inheritance
  - class A extends B extends C { …} is not permitted
- Interface: A kind of class – with the following differences from a class
  - Defines only abstract methods and final fields.
  - Does not specify any code to implement these methods
  - Data fields contain only constants.
  - It is the responsibility of the class that implements an interface to define the code for implementing these methods.
- Syntax:

```
interface InterfaceName
        {
                variables declaration;
                methods declaration;
        }
```

- Designed to support dynamic method resolution at run time

Interfaces

Implementing an Interface

- Syntax:

```
class classname extends superclass implements interface[,interface…] {
        // class body
}
```

- Rules:
  - Methods that implement an interface must be declared *public*
  - Type signatures of the implementing method must match exactly the type signature specified in the interface definition

```
1  interface Area
2  {
3      double pi = 3.14;
4      double computeArea();
5  }
6  class Rectangle implements Area
7  {
8      private double length, width;
9
10     public Rectangle(double len,double wid)
11     { length = len; width = wid; }
12
13     public double computeArea()
14     {  return length * width; }
15 }
16 class Circle implements Area
17 {
18     private double radius;
19
20     public Circle(double rad)
21     { radius = rad; }
22
23     public double computeArea()
24     { return pi * radius * radius; }
25 }
```

```
26  class InterfaceDemo
27  {
28      public static void main(String args[])
29      {
30          Rectangle rec = new Rectangle(12,7);
31          Circle cir = new Circle( 10 );
32          Area ar;     // interface object reference
33          ar = rec;    // ar refers to rec object
34          System.out.println ("Area of rectangle: " +
35                  ar.computeArea());
36          ar = cir;    // ar refers to circle object
37          System.out.println ("Area of circle: " +
38                  ar.computeArea());
39      }
40  }
```

```
Area of rectangle: 84.0
Area of circle: 314.0
```

```
26  class InterfaceDemo
27  {
28      public static void main(String args[])
29      {
30          Area ar;     // interface object reference
31          ar = new Rectangle(12,7);    // ar refers to rec object
32          System.out.println ("Area of rectangle: " +
33                  ar.computeArea());
34          ar = new Circle( 10 );  // ar refers to circle object
35          System.out.println ("Area of circle: " +
36                   ar.computeArea());
37      }
38  }
```

Partial Implementations

- If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**.

```
Interface  I_Test
{
        void method1();
        void method2();
}
```

```
abstract class Incomplete  implements  I_Test
 {
        int a, b;

        void method1()
        {
                // …
        }

        // …
}
```

Interfaces

Extending Interfaces

- Syntax:

```
interface name2 extends name1
{
        // body of name2
}
```

```
1  // One interface can extend another.
2  interface A
3  {
4    void meth1();
5    void meth2();
6  }
7
8  // B now includes meth1() and meth2(),it adds meth3()
9  interface B extends A
10 {
11   void meth3();
12 }
```

```java
// This class must implement all of A and B
class MyClass implements B
{
  public void meth1()
  {
    System.out.println("Implement meth1().");
  }
  public void meth2()
  {
    System.out.println("Implement meth2().");
  }
  public void meth3()
  {
    System.out.println("Implement meth3().");
  }
}
class InterfaceTest
{
  public static void main(String ar
  {
    MyClass ob = new MyClass();
    ob.meth1();  ob.meth2();  ob.meth3();
  }
}
```

```
Implement meth1().
Implement meth2().
Implement meth3().
```

Interfaces

Extending Interfaces (Continued …)

- Example 2

```
interface ItemConstants
{
        int code = 1001;
        String name = "AC";
}
interface ItemMethods
{
        void display();
}
interface Item extends ItemConstants, ItemMethods
{
        // …
}
```

```java
interface ItemConstants
{
    int code = 1001;
    String name = "AC";
}
interface ItemMethods
{
    public void display();
}


class Test implements ItemConstants,ItemMethods
{
    public void display()
    {
        // code++; Invalid !
        System.out.println(code+ "\t" +name);
    }
}
```

```
21  class MultipleInterfaceDemo
22  {
23      public static void main(String args[])
24      {
25          Test t = new Test();
26          t.display();
27      }
28  }
```

```java
interface ItemConstants
{
    int code = 1001;
    String name = "AC";
}
interface ItemMethods extends ItemConstants
{
    public void display();
}
interface Item extends ItemMethods
{
    // …
}

class Test implements Item
{
    public void display()
    {
        // code++; Invalid !
        System.out.println(code+ "\t" +name);
    }
}
```

```
24  class MultipleInterfaceDemo2
25  {
26      public static void main(String args[])
27      {
28          Test t = new Test();
29          t.display();
30      }
31  }
```

```
1001    AC
```

Default Interface Methods

- Prior to Java 8, an interface could not define any implementation whatsoever
- The release of Java 8 has changed this by adding a new capability to interface called the *default method*
  - A default method lets you define a default implementation for an interface method
  - Its primary motivation was to provide a means by which interfaces could be expanded without breaking existing code

```java
interface MyIF
{
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();
    // This is a default method. Notice that it provides a default implementation.
    default String getString()
    {
        return "Default String";
    }
}

// Implement MyIF.
class MyIFImp implements MyIF
{
    // Only getNumber() defined by MyIF needs to be implemented.
    // getString() can be allowed to default.
    public int getNumber() {
        return 100;
    }
}
// Use the default method.
class DefaultMethodDemo
{
    public static void main(String[] args)
    {
        MyIFImp obj = new MyIFImp();
        // Can call getNumber(), because it is explicitly implemented by MyIFImp:
        System.out.println(obj.getNumber());
        // Can also call getString(), because of default implementation:
        System.out.println(obj.getString());
    }
}
```

Cloneable interface

protected Object clone() throws CloneNotSupportedException

Creates and returns a copy of this object.

```java
class Employee implements Cloneable
{
    int id;
    String name;
    double salary;

    public Employee(int id, String name, double sal )
    {
        this.id = id; this.name = name; salary = sal;
    }

    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee)super.clone();
    }
    void raiseSalary()
    { salary += 5000; }

    public String toString()
    {   return "id: "+id+", name: "+name+" , salary: "+salary; }
}
```

```
23  public class ObjectCloneDemo
24  {
25      public static void main(String[] args)
26      {
27          Employee emp1 = new Employee(111, "John", 35000);
28          Employee emp2 = emp1;
29          Employee emp3 = null;
30
31          try
32          {
33         // Creating a clone of emp1 and assigning it to emp2
34              emp3 = emp1.clone();
35          }
36          catch(CloneNotSupportedException e)
37          {
38              System.out.println("Cloning not allowed.");
39          }
40
41          emp2.raiseSalary();
42
43          System.out.println( "emp1: "+emp1);
44          System.out.println( "emp2: "+emp2);
45          System.out.println( "emp3: "+emp3);
46      }
47  }
```

```
emp1: id: 111, name: John , salary: 40000.0
emp2: id: 111, name: John , salary: 40000.0
emp3: id: 111, name: John , salary: 35000.0
```

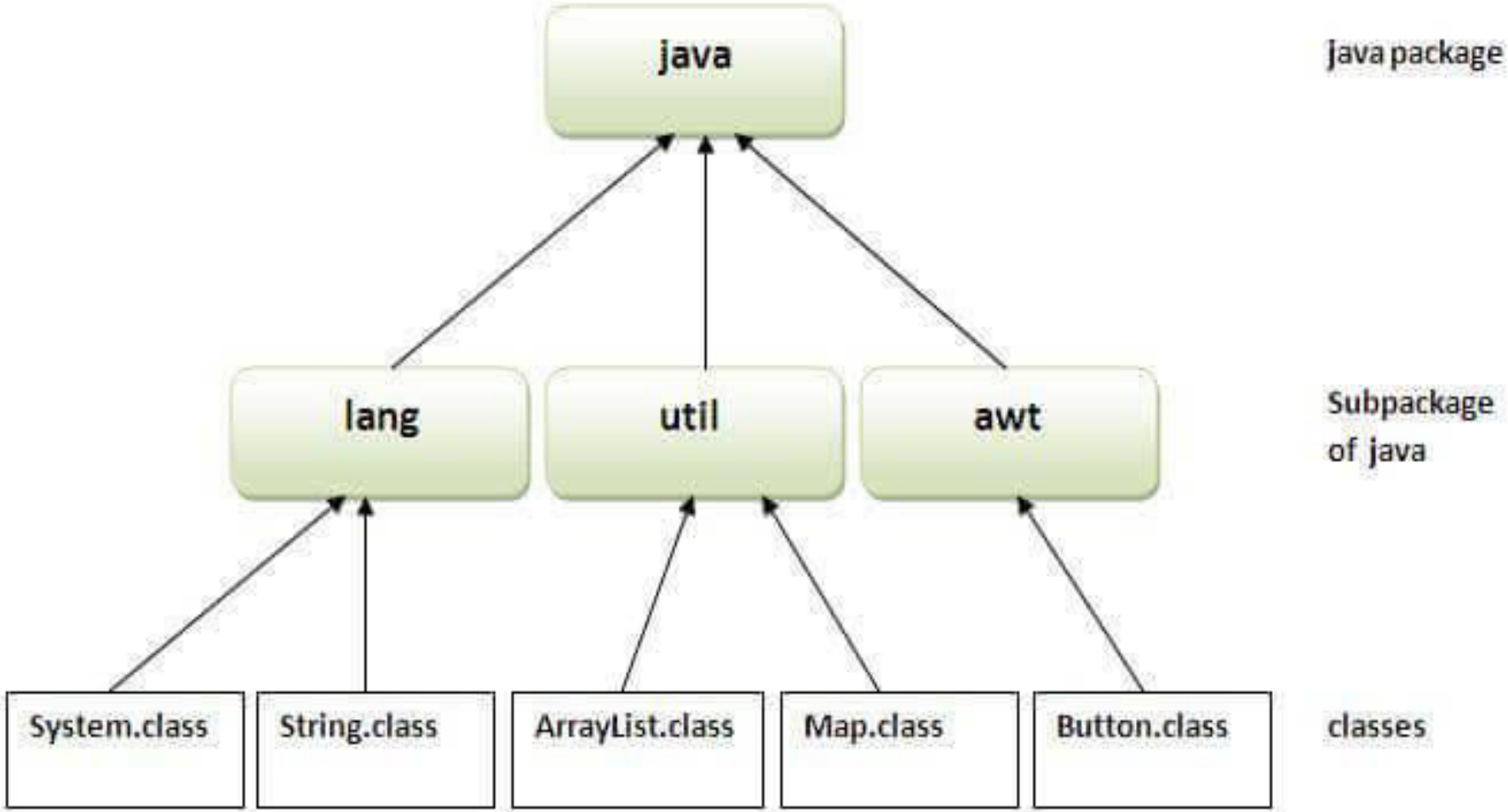
Exercise:

Write a java program to calculate total fees of *'n'* students by implementing *AcademicFees* and *NonAcademicFees* interfaces. Academic fee is calculated based on – Admission fee, Tuition fee, Exam fee. Nonacademic fee is calculated based on – Hostel fees, Sports Fees, Annual-day fees.

Packages

Defining a package

- Include *package* statement as the first one in a Java source file
  - Any class declared within that file belongs to that package
- *Package* statement defines a name space in which classes are stored
  - Absence of *package* statement puts class names in default package (without any name)
- Usage:
  - Syntax: package package-name;
  - Example: package myPackage;


java
lang
util
awt
System.class
String.class
ArrayList.class
Map.class
Button.class
Java package
Subpackage of java
classes

Packages

- **Defining a package** (Continued …)
  - Example 2: (Two classes in two files Balance.java and AccountBalance.java)

```
package mypack;                          // In MyPack folder
public class Balance {
        String name; double bal;
        public Balance(String n, double b) { name = n; bal = b; }
        public void show() { System.out.println (name + " " + bal); }
}

import mypack.*;
class AccountBalance {                   // In the parent folder of MyPack
        public static void main(String args[]) {
                Balance current = new Balance ("Herbert Schildt", 1243.78);
                current.show();
        }
}
```

```java
package mybookpack;

public class Book
{
   private String title;
   private String author;
   private int pubDate;

   public Book(String t, String a, int d)
   {
      title = t;
      author = a;
      pubDate = d;
   }

   public void show()
   {
      System.out.println(title);
      System.out.println(author);
      System.out.println(pubDate);
   }
}
```

```java
import mybookpack.Book; // This class is in package mybookpack.
class UseBook // Use the Book Class from mybookpack.
{
  public static void main(String[] args)
  {
    Book[] books = new Book[5];

    books[0] = new Book("The Art of Computer Programming, Vol 3",
                        "Knuth", 1973);
    books[1] = new Book("Moby Dick",
                        "Melville", 1851);
    books[2] = new Book("Thirteen at Dinner",
                        "Christie", 1933);
    books[3] = new Book("Red Storm Rising",
                       "Clancy", 1986);
    books[4] = new Book("On the Road",
                       "Kerouac", 1955);

    for(int i=0; i < books.length; i++)
    {
      books[i].show(); System.out.println();
    }
  }
}
```

Packages

Access Protection

- Java provides four categories of visibility for class members
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- Access specifiers used
  - public: can be accessed anywhere
  - private: can’t be seen outside of its class
  - protected: can be seen outside the current package only to direct subclasses of that class
  - (default (package-private): visible to subclasses as well as other classes in the same package)

Packages

- **Access Protection** (Continued …)

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Package P1

```
public class   C1
{
      private  int  a1;
      int  a2;
      protected  int  a3;
      public  int  a4;
}
class   C2
{
      // a2 , a3 , a4
}
class C3  extends  C1
{
      // a2 , a3 , a4
}
```

Package P2

```
class C4   extends   C1
{
      //  a3 , a4
}


class   C5
{
      //  a4
}
```

Packages

Access Protection (Continued …)

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Case – I:

Package p1 contents:

Package - p1

1. Class Protection
2. Class Derived
3. Class SamePackage
4. Class Demo

//class Protection is accessed by other 3 classes

```
1  public class Protection
2  {
3     int n = 1;
4     private int n_pri = 2;
5     protected int n_pro = 3;
6     public int n_pub = 4;
7
8     public Protection() {
9        System.out.println("base constructor");
10       System.out.println("n = " + n);
11       System.out.println("n_pri = " + n_pri);
12       System.out.println("n_pro = " + n_pro);
13       System.out.println("n_pub = " + n_pub);
14    }
15 }
```

Package p1 contents:

```
1  class Derived extends Protection
2  {
3    Derived() {
4       System.out.println("derived constructor");
5       System.out.println("n = " + n);
6
7  //   class only
8  //   System.out.println("n_pri = " + n_pri);
9
10      System.out.println("n_pro = " + n_pro);
11      System.out.println("n_pub = " + n_pub);
12   }
13 }
```

Package p1 contents:

```
1  class SamePackage
2  {
3     SamePackage()
4     {
5        Protection p = new Protection();
6        System.out.println("same package constructor");
7        System.out.println("n = " + p.n);
8
9  //    class only
10 //    System.out.println("n_pri = " + p.n_pri);
11
12       System.out.println("n_pro = " + p.n_pro);
13       System.out.println("n_pub = " + p.n_pub);
14    }
15 }
```

Package p1 contents:

```
1  // Instantiate the various classes in p1.
2  public class Demo
3  {
4    public static void main(String args[])
5    {
6      Protection ob1 = new Protection();
7      Derived ob2 = new Derived();
8      SamePackage ob3 = new SamePackage();
9    }
10 }
```

Case - II

Package - p1

Class Protection

Package – p2

1. Class Protection2
2. Class OtherPackage
3. Class Demo

//class Protection is accessed by other 3 classes

Package p1 contents:

```
1  public class Protection
2  {
3     int n = 1;
4     private int n_pri = 2;
5     protected int n_pro = 3;
6     public int n_pub = 4;
7
8     public Protection() {
9        System.out.println("base constructor");
10       System.out.println("n = " + n);
11       System.out.println("n_pri = " + n_pri);
12       System.out.println("n_pro = " + n_pro);
13       System.out.println("n_pub = " + n_pub);
14    }
15 }
```

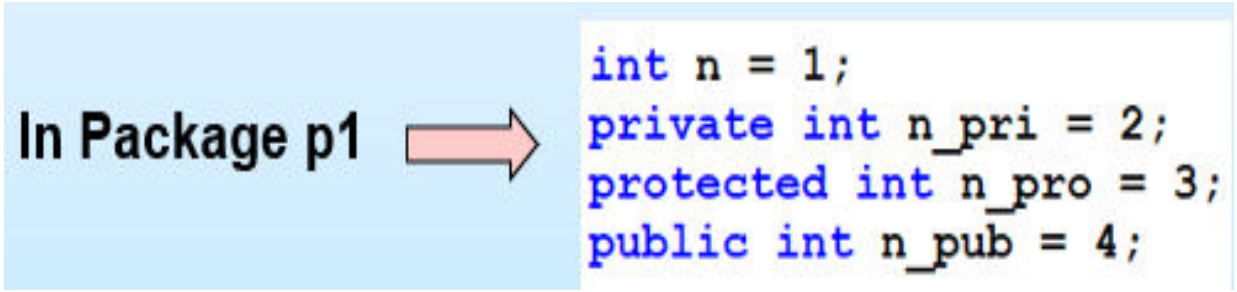
Contents of package p2:

```
1  class Protection2 extends Protection
2  {
3    Protection2()
4    {
5      System.out.println("derived other package constructor");
6
7  //   class or package only
8  //   System.out.println("n = " + n);
9
10 //   class only
11 //   System.out.println("n_pri = " + n_pri);
12
13     System.out.println("n_pro = " + n_pro);
14     System.out.println("n_pub = " + n_pub);
15   }
16 }
```

Package p1 →class protection

```
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
```

Contents of package p2:




```
1   class OtherPackage
2   {
3     OtherPackage()
4     {
5       Protection p = new Protection();
6       System.out.println("other package constructor");
7
8   //    class or package only
9   //    System.out.println("n = " + p.n);
10
11  //    class only
12  //    System.out.println("n_pri = " + p.n_pri);
13
14  //    class, subclass or package only
15  //    System.out.println("n_pro = " + p.n_pro);
16
17      System.out.println("n_pub = " + p.n_pub);
18    }
19  }
```

Contents of package p2:

```
1  public class Demo
2  {
3    public static void main(String args[])
4    {
5      Protection2 ob1 = new Protection2();
6      OtherPackage ob2 = new OtherPackage();
7    }
8  }
```

Package p1 →class protection

```
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
```

The End