Java Programming
(MCA 4253)

Inheritance

- **Meaning of inheritance**
  - Mechanism in which one object acquires properties of another object
  - Create new classes that are built upon existing (parent) class
    - Methods and fields of parent class can be reused
    - New methods and fields can be added

Inheritance

Superclass and Subclass

- Create subclass from superclass
  - Subclass inherits all (except private) instance variables and methods (except constructors) of super class and adds its own unique instance variables and methods
  - Subclass cannot access *private* members of superclass (they are not inherited into the subclass)
  - Private members of a superclass are accessible by methods of superclass only
- Syntax:

```
class subclass-name extends superclass-name
{
        // body of class
}
```

  - Only one superclass for any subclass (multiple inheritance not supported)
  - Hierarchy of inheritance can be used (multilevel inheritance)

Inheritance

Superclass and Subclass (Continued …)

- Example

```
1. class A
2.  {
3.   int i, j;
4.   void showij ()
5.   {
6.          System.out.println (i + " " + j);
7.   }
8. }
9. class   B   extends   A
10.{
11. int k;
12. void showk ()
13. {
14.   System.out.println (k);
     }
   }
```

```
15.class   SimpleInheritance
16.{
17.   public static void main(String args[])
18.   {
19.         A superOb = new A();
20.         B subOb = new B();
21.         superOb.i = 10;
22.         superOb.j = 20;
23.         superOb.showij();       // 10 20
24.         subOb.i = 30;
25.         subOb.j = 40;
26.         subOb.k = 50;
27.         subOb.showij();         // 30 40
28.         subOb.showk();          // 50
29.   }
30.}
```

```
1  class A
2  {
3     int i;
4     private int j;
5
6     void setij(int x, int y)
7     {
8        i = x;
9        j = y;
10    }
11 }
12
13 class B extends A
14 {
15    int total;
16
17    void sum()
18    {
19       total = i + j;   // Error !
20    }
21 }
```

```
1  class A
2  {
3    int i;
4    private int j;
5
6    void setij(int x, int y)
7    {
8      i = x;
9      j = y;
10   }
11   int get_i() { return i; }
12   int get_j() { return j; }
13 }
14
15 class B extends A
16 {
17   int total;
18
19   void sum()
20   {
21     total = get_i() + get_j();
22   }
23 }
```

```
25  class Access
26  {
27    public static void main(String args[])
28    {
29      B subOb = new B();
30
31      subOb.setij(10, 12);
32
33      subOb.sum();
34      System.out.println("Total is " + subOb.total);
35    }
36  }
```

```java
class Base
{
  Base()
  {
      System.out.println("Base constructor");
  }
}

class Derived extends Base
{
  Derived()
  {
     System.out.println("Derived constructor");
  }
}

class ConstructorTest
{
  public static void main(St
  {
    Derived D1;
    D1 = new Derived();
    Derived D2 = new Derive
  }
}
```

```
Base constructor
Derived constructor
Base constructor
Derived constructor
```

```java
class Base
{
  Base()
  { System.out.println("Base constructor"); }
}

class Derived extends Base
{
  Derived()
  { System.out.println("Derived constructor-1"); }

  Derived( int x )
    {  System.out.println("Derived constructor-2"); }
}

class ConstructorTest
{
  public static void main(St
  {
    Derived D1, D2;
    D1 = new Derived();
    D2 = new Derived(22);
  }
}
```

```
Base constructor
Derived constructor-1
Base constructor
Derived constructor-2
```

Inheritance

- **Keyword *super***
  - Used in a subclass to refer to its immediate superclass
  - Form 1 (Example 1)
    - To call superclass constructors
    - *Super* must always be the first statement inside a subclass' constructor
    - Syntax: super(arg-list);
  - Form 2 (Example 2)
    - To access members of superclass hidden by subclass
    - Syntax: super.member
    - Member can be either a method or an instance variable

Inheritance

- **Keyword *super*** (Continued …)
  - Example 1

```
class Box
{
    double width, height, depth;
    Box (double w, double h, double d)
            { width = w; height = h; depth = d; }
}
class BoxWeight extends Box
{
    double weight;
    BoxWeight (double w, double h, double d, double wt)
    {
            super(w, h, d);
            weight = wt;
    }
}
In main():      BoxWeight box = new BoxWeight (10, 20, 15, 2.5);
```

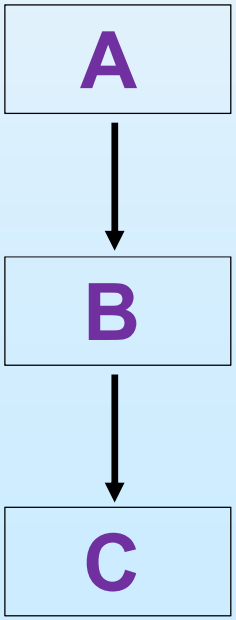
Keyword *super* (Continued …)

- Example 2

```
class A {
    int m;
}
class B extends A {
    int m;        // This m hides m in A
    B (int a, int b)
    {
            super.m = a;
            m = b;
    }
    void show() {
            System.out.println ("m in superclass A = " + super.m);
            System.out.println ("m in subclass B = " + m);
    }
}
In main():      B subOb = new B (10, 20);
                subOb.show ();
```

Inheritance

- **Keyword *super*** (Continued …)
  - *super* in a subclass refers to its immediate superclass
    - Constructors are called in the order of derivation, from superclass to subclass, whether or not *super*() is used
      - If *super*() is used, it must be the first statement in a subclass' constructor
      - If *super*() is not used, the constructor without parameter of each class is executed



```
1  class Box
2  {
3    private double width;
4    private double height;
5    private double depth;
6
7    Box(double w, double h, double d)
8    {
9      width = w;
10     height = h;
11     depth = d;
12   }
13
14   Box()
15   {
16     width = height = depth = -1;
17   }
18
19   double volume()
20   {
21     return width * height * depth;
22   }
23 }
```

```
27  class BoxWeight extends Box
28  {
29    double weight;
30
31    BoxWeight(double w, double h, double d, double m)
32    {
33      super(w, h, d);
34      weight = m;
35    }
36
37    BoxWeight()
38    {
39      super();
40      weight = -1;
41    }
42
43  }
```

```
44  class DemoSuper
45  {
46    public static void main(String args[])
47    {
48      BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
49      BoxWeight mybox2 = new BoxWeight();
50      double vol = mybox1.volume();
51      System.out.println("Volume of mybox1 is " + vol);
52      System.out.println("Weight of mybox1 is " + mybox1.weight);
53
54      vol = mybox2.volume();
55      System.out.println("Volume of mybox2 is " + vol);
56      System.out.println("Weight of mybox2 is " + mybox2.weight);
57    }
58  }
```

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is -1.0
Weight of mybox2 is -1.0
```

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
```

Question-3

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

```
17 class SupSubRef {
18   public static void main(String[] args) {
19     X x = new X(10);
20     X x2;
21     Y y = new Y(5, 6);
22
23     x2 = x;
24     System.out.println("x2.a: " + x2.a);
25
26     x2 = y;
27     System.out.println("x2.a: " + x2.a);
28
29     x2.a = 19;
30     x2.b = 27;  // Error, X doesn't have a b member
31   }
32 }
```

Inheritance

Multilevel hierarchy

- Multiple levels of inheritance
  - Class A → Class B → Class C …
- Example:

```
class A {
        // Body of class A
}
class B extends A {
        // Body of class B
}
class C extends B {
        // Body of class C
}
```

Inheritance

```
4  class A {
5    A() {
6      System.out.println("Constructing A.");
7    }
8  }
9
10 class B extends A {
11   B() {
12     System.out.println("Constructing B.");
13   }
14 }
15
16 class C extends B {
17   C() {
18     System.out.println("Constructing C.");
19   }
20 }
21
22 class OrderOfConstruction {
23   public static void main(String[] args) {
24     C c = new C();
25   }
26 }
```

```
Constructing A.
Constructing B.
Constructing C.
```

```
1  class Box {
2    private double width;
3    private double height;
4    private double depth;
5
6    Box(double w, double h, double d)
7    {  width = w; height = h; depth = d; }
8
9    Box()
10   { width = -1; height = -1; depth = -1; }
11
12     // ....other methods
13 }
14
15 class BoxWeight extends Box {   // Add weight.
16   double weight; // weight of box
17
18   BoxWeight(double w, double h, double d, double m)
19   { super(w, h, d); weight = m; }
20
21   BoxWeight()
22   { super(); weight = -1; }
23
24     // ... other methods
25  }
```

```
27  class Shipment extends BoxWeight // Add shipping costs
28  {
29    double cost;
30
31    Shipment(double w, double h, double d, double m, double c)
32    { super(w, h, d, m); cost = c; }
33
34    Shipment()
35    {  super(); cost = -1; }
36
37      // ... other methods
38  }
39
40  class DemoShipment
41  {
42    public static void main(String args[])
43    {
44      Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
45      Shipment shipment2 = new Shipment();
46
47      // ... Other statements
48    }
49  }
```

Inheritance

Method overriding

- When a subclass method has same name as a superclass method
  - If type of signatures (number and data type) are same (Example 1)
    - Method in the subclass overrides the method in superclass
    - When called, subclass method is used (superclass method is hidden)
  - If type of signatures (number and data type) are not same (Example 2)
    - Two methods are overloaded

Method overriding example

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

```
test() of B
```

Method overriding example

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

```
test() of A
test() of B
```

```java
class A
{
    void show (int n)
    { System.out.println ("In A: "+ n); }
}

class B extends A
{
    void show (String s)
    { System.out.println ("In B: " + s); }
}

class OverLoad
{
    public static void main(String args[])
    {
        B subOb = new B();
        subOb.show ("MCA");
        subOb.show(10);
    }
}
```

```
In B: MCA
In A: 10
```

Inheritance

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

```
class A { }

class B extends A { }

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

Inheritance

- **Dynamic Method Dispatch** (Continued …)
  - Example 1:

```
class A {
        void callMe() { System.out.println ("Inside A's callMe method"); }
}
class B extends A {
        void callMe() { System.out.println ("Inside B's callMe method"); }
}
class C extends A {
        void callMe() { System.out.println ("Inside C's callMe method"); }
}
```

In main(): A aObj = new A(); B bObj = new B(); C cObj = new C();

```
A refVar;                                   // reference variable for A
refVar = aObj; refVar.callMe();             // calls A's version of callMe
refVar = bObj; refVar.callMe();             // calls B's version of callMe
refVar = cObj; refVar.callMe();             // calls C's version of callMe
```

```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
```

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

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

```
6   class Bank1 extends
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  }
```

Inheritance

- **Use of overridden methods**
  - Polymorphism (one interface, multiple methods)
    - Allows a general class to specify methods that will be common to all its subclasses
    - Allows subclasses to define specific implementations of some or all these methods
  - 3 methods to implement polymorphism
    - Method overloading
    - Method overriding
    - Interfaces

Inheritance

Abstract classes

- Abstract class is a class that cannot be instantiated
- 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);

Inheritance

- **Abstract classes** (Continued …)
  - 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

Note: A non-abstract class is called a concrete class

```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.
```

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 }
```

error: B 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  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   }
```

Inheritance

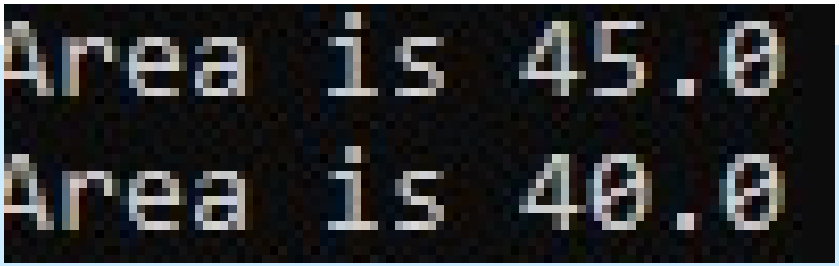
- **Run-time polymorphism**
  - Not possible to instantiate objects of Abstract classes; but, object references can be created
  - Run-time polymorphism implemented through the use of superclass references
  - Possible to create a reference to an abstract class so that it can be used to point to a subclass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

Example: Polymorphism in Figure class

Example: Abstract class- Figure

```
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: Abstract class Figure(2)

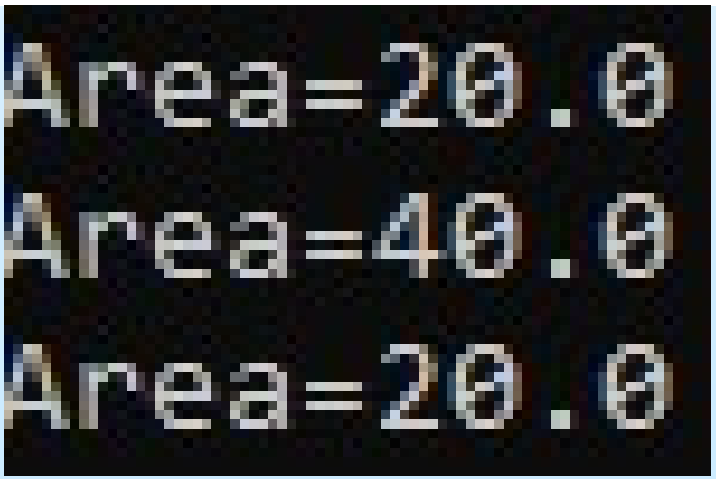
```
1  abstract class Figure
2  {
3     abstract double area();
4  }
5  class Rectangle extends Figure
6  {
7    double length, width;
8
9    Rectangle(double l, double w)
10   { length = l; width = w; }
11
12   double area()  // override area for rectangle
13   {  return length * width;  }
14 }
15 class Triangle extends Figure
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 AbstractAreas2
27  {
28    public static void main(String args[])
29    {
30     // Figure f = new Figure(10, 10); // illegal now
31      Rectangle r = new Rectangle(9, 5);
32      Triangle t = new Triangle(10, 8);
33
34      Figure figref; // this is OK, no object is created
35
36      figref = r;
37      System.out.println("Area is " + figref.area());
38
39      figref = t;
40      System.out.println("Area is " + figref.area());
41    }
42  }
```

Example: Abstract class- Shape

```
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  }
```



Inheritance

Using *final* with inheritance

Prevent Overriding

- To prevent a method from overriding, specify *final* as a modifier at the start of its declaration
- Example:

```
class A
{
        final void meth()
        {
                System.out.println ("This is a final method.");
        }
}
class B extends A
{
        void meth()
        {                       // ERROR! Can't override.
                System.out.println ("Illegal");
        }
}
```

```
1  abstract class A
2  {
3    abstract final void meth();
4  }
5
6  class B extends A
7  {
8    void meth()
9    {
10       // Some statements
11   }
12 }
```

Error: illegal combination of modifiers: abstract and final

Inheritance

Using *final* with inheritance (Continued …)

Prevent Inheritance

- To prevent a class from being inherited, specify *final* as a modifier at the start of its declaration
  - Declaring a class as final implicitly declares all its methods as final
- Example:

```
final class A
{
        //…
 }
class B extends A    // ERROR: Cannot inherit from final A
 {
        //…
 }
```

```
1  final abstract class A
2  {
3     abstract final void meth();
4  }
```

Error: illegal combination of modifiers: abstract and final

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

Error: illegal combination of modifiers: abstract and final

Packages

- **What is a package ?**
  - Container for classes
  - Classes with same name possible in different packages
  - Classes in a package can be imported into required programs
  - Both a naming and visibility control mechanism
    - We can define classes inside a package that are not accessible outside that package
    - We can also define class members that are only exposed to other members of the same package

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;
  - Storing a package: Store the package in a folder with the same name as package name (case is significant)

Packages

- **Defining a package** (Continued …)
  - More than one file can include the same *package* statement
    - Store all classes in all these files in the same folder
  - Possible to create hierarchy of packages
    - Separate each package name from the one above it using a period (.)
    - Syntax: package pkg1[.pkg2[.pkg3]];
    - Example: package java.awt.image; (store in folder java\awt\image)

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);
   }
}
```

```
2  import mybookpack.Book; // This class is in package mybookpack.
3  class UseBook // Use the Book Class from mybookpack.
4  {
5    public static void main(String[] args)
6    {
7      Book[] books = new Book[5];
8
9      books[0] = new Book("The Art of Computer Programming, Vol 3",
10                         "Knuth", 1973);
11     books[1] = new Book("Moby Dick",
12                         "Melville", 1851);
13     books[2] = new Book("Thirteen at Dinner",
14                         "Christie", 1933);
15     books[3] = new Book("Red Storm Rising",
16                        "Clancy", 1986);
17     books[4] = new Book("On the Road",
18                        "Kerouac", 1955);
19
20     for(int i=0; i < books.length; i++)
21     {
22       books[i].show(); System.out.println();
23     }
24   }
25 }
```

Packages

Finding packages and CLASSPATH

- To search for packages, three options
  1. Java run-time system uses current working directory as its starting point for search
  2. We can specify a directory path or paths by setting CLASSPATH environmental variable
  3. We can use *–classpath* option with *javac* and *java* to specify path to the classes
- Example:
  - set CLASSPATH=C:\MyPrograms\Java (if the required class is present in this folder)
  - javac –classpath C:\MyPrograms\Java sample.java
  - java –classpath C:\MyPrograms\Java sample

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)

Note: When a class is public, it must be the only public class in the file and the file must have the same name as the class

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

```
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

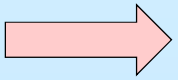
1. Class Protection2
2. Class OtherPackage
3. Class Demo

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 }
```

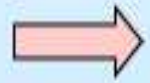
Contents of package p2:

```
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 }
```

In Package p1 

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

Contents of package p2:

In Package p1 

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

```
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  }
```

In Package p1 

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

Packages

Importing Packages

- A class from another package can be accessed using 2 methods
  - By specifying the fully qualified class name
  - Using **import** statement
- The *import* statement is used immediately after *package* statement (if it exists) and before any class definitions
  - Syntax: import pkg1[.pkg2].(classname|*);
  - Examples: import java.util.Date;

    import java.lang.*; // basic language functions (default)
- Fully qualified class name

```
import java.util.*;
class myDate extends Date { …}
```

OR

```
class myDate extends java.util.Date { … }
```

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 They are implicitly public, static, and final.
  - 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      static final float pi=3.14F;
4      float compute (float x, float y);
5  }
6
7  class Rectangle implements Area
8  {
9      public float compute (float x, float y)
10         { return x * y; }
11 }
12
13 class Circle implements Area
14 {
15     public float compute (float x, float y)
16         { return pi * x * x; }
17 }
```

```
18  class InterfaceDemo
19  {
20      public static void main(String args[])
21      {
22          Rectangle rec = new Rectangle();
23          Circle cir = new Circle();
24          Area ar;     // interface object reference
25          ar = rec;    // ar refers to rec object
26          System.out.println ("Area of rectangle: " +
27                  ar.compute(20, 10));
28          ar = cir;    // ar refers to rec object
29          System.out.println ("Area of circle: " +
30                  ar.compute(20, 10));
31      }
32  }
```

```
Area of rectangle: 200.0
Area of circle: 1256.0
```

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
}
```

```java
// One interface can extend another.
interface A
{
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2(),it adds meth3()
interface B extends A
{
  void meth3();
}
```

```
14  // This class must implement all of A and B
15  class MyClass implements B
16  {
17    public void meth1()
18    {
19      System.out.println("Implement meth1().");
20    }
21    public void meth2()
22    {
23      System.out.println("Implement meth2().");
24    }
25    public void meth3()
26    {
27      System.out.println("Implement meth3().");
28    }
29  }
30  class InterfaceTest
31  {
32    public static void
33    {
34      MyClass ob = new
35      ob.meth1();  ob.
36    }
37  }
```

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

Interfaces

- **Extending Interfaces** (Continued …)
  - Example 1

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

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  }
```

```
1001    AC
```

```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
```

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.
- 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.
- Some methods inherited from Object that are:

Some methods of Object class

protected Object clone() throws CloneNotSupportedException

- Creates a new object that is the same as the invoking object.

public boolean equals(Object obj)

- Determines whether one object is equal to another.

protected void finalize() throws Throwable

- Called before an unused object is recycled. 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()

- Obtains the class of an object at run time.

public int hashCode()

- Returns the hash code associated with the invoking object.

public String toString()

- Returns a string representation of the object.

Eg-1: object comparison using ==

```
1  class Person
2  {
3      private String name;
4      private int age,weight;
5
6     public Person(String n, int a, int w)
7     {  name = n ; age = a ; weight = w; }
8   }
9  class ObjectEqualsDemo2
10 {
11    public static void main(String []args)
12    {
13       Person p1 = new Person("abc", 35, 120 );
14       Person p2 = new Person("abc", 35, 120 );
15       Person p3 = p1;
16       System.out.println( p1.getClass() );
17       if( p1 == p2 )
18         System.out.println("p1 == p2 ");
19       else
20         System.out.println("p1 != p2 ");
21       if( p1 == p3 )
22         System.out.println("p1 == p3 ");
23     }
24 }
```

```
class Person
p1 != p2
p1 == p3
```

Eg-2: object comparison using equals()

```
1  class Person
2  {
3      private String name;
4      private int age;
5      private int weight;
6
7     public Person(String n, int a, int w)
8     {  name = n ; age = a ; weight = w; }
9   }
10 class ObjectEqualsDemo1
11 {
12    public static void main(String []args)
13    {
14       Person p1 = new Person("abc", 35, 120 );
15       Person p2 = new Person("abc", 35, 120 );
16
17       if( p1.equals(p2) )
18         System.out.println("p1 == p2 ");
19       else
20         System.out.println("p1 != p2 ");
21     }
22 }
```

```
p1 != p2
```

Eg-3: object comparison using equals()

```
1  class Person
2  {
3      private String name;
4      private int age,weight;
5
6     public Person(String n, int a, int w)
7     {  name = n ; age = a ; weight = w; }
8   }
9  class ObjectEqualsDemo2
10 {
11    public static void main(String []args)
12    {
13       Person p1 = new Person("abc", 35, 120 );
14       Person p2 = new Person("abc", 35, 120 );
15       Person p3 = p1;
16
17       if( p1.equals(p2) )
18         System.out.println("p1 == p2 ");
19       else
20         System.out.println("p1 != p2 ");
21       if( p1.equals(p3) )
22         System.out.println("p1 == p3 ");
23     }
24 }
```

```
p1 != p2
p1 == p3
```

Overriding equals()

```
1  class Person
2  {
3       private String name;
4       private int age;
5       private int weight;
6
7     public Person(String n, int a, int w)
8     {   name = n ; age = a ; weight = w; }
9
10    public boolean equals(Person person_obj)
11    {
12       return age == person_obj.age &&
13              weight == person_obj.weight &&
14              name.equals(person_obj.name);
15    }
16 }
```

```
19  class ObjectEqualsDemo
20  {
21      public static void main(String []args)
22      {
23          Person p1 = new Person("abc", 35, 120 );
24          Person p2 = new Person("xyz", 40, 120 );
25          Person p3 = new Person("abc", 35, 120 );
26
27          if( p1.equals(p2) )
28            System.out.println("p1 == p2 ");
29          else
30            System.out.println("p1 != p2 ");
31
32          if( p1.equals(p3) )
33              System.out.println("p1
34          else
35            System.out.println("p1 !=
36      }
37  }
```

```
p1 != p2
p1 == p3
```

Question : Compare 2 Emp objects based on salary

```
1   class Emp
2   {
3       private String name;
4       private int age;
5       private int salary;
6
7
8
9
10
11
12
13
14  }
```

Question…

```
15  class EmpEqualsDemo
16  {
17      public static void main(String []args)
18      {
19          Emp E1 = new Emp("abc", 35, 22000 );
20          Emp E2 = new Emp("xyz", 40, 60000 );
21          Emp E3 = new Emp("abc", 35, 60000 );
22
23          if( E1.equals(E2) )
24            System.out.println("E1 == E2 ");
25          else
26            System.out.println("E1 != E2 ");
27
28          if( E2.equals(E3) )
29              System.out.println("E2
30          else
31            System.out.println("E2 !
32      }
33  }
```

```
E1 != E2
E2 == E3
```

Question…

```
1   class Emp
2   {
3       private String name;
4       private int age;
5       private int salary;
6
7      public Emp(String n, int a, int s)
8      {   name = n ; age = a ; salary = s; }
9
10     public boolean equals(Emp Emp_obj)
11     {
12        return salary == Emp_obj.salary;
13     }
14  }
```

Cloneable interface

- Creating a new reference that points to the same memory location is called a Shallow copy.
- Creating a copy of object in a different memory location is called a Deep copy.

```
1  // Demonstrate the clone() method.
2  class TestClone implements Cloneable
3  {
4    int a;
5    double b;
6
7     public TestClone clone() throws CloneNotSupportedException
8     {
9        // call Object.clone()
10       TestClone cloned = (TestClone) super.clone();
11
12       return cloned;
13    }
14 }
```

```
16  class CloneDemo
17  {
18    public static void main(String args[])
19    {
20      TestClone x1 = new TestClone();
21      TestClone x2;
22
23      x1.a = 10;
24      x1.b = 20.98;
25
26      System.out.println("x1: " + x1.a + " " + x1.b);
27      try
28      {
29          x2 = x1.clone(); // clone x1
30          System.out.println("x2: " + x2.a + " " + x2.b);
31      }
32      catch(CloneNotSupportedException e)
33      {
34          System.out.println("Cloning not allowed.");
35      }
36    }
37  }
```

Clone Example-2:

```
1  // Demonstrate the clone() method.
2  class TestClone implements Cloneable
3  {
4    int a;
5    double b;
6
7     public TestClone clone() throws CloneNotSupportedException
8     {
9        // call Object.clone()
10       TestClone cloned = (TestClone) super.clone();
11
12       return cloned;
13    }
14 }
```

```
x1: 10 20.5
x2: 10 20.5

x1: 10 20.5
x2: 11 20.5
```

```java
class CloneDemo
{
  public static void main(String args[])
  {
    TestClone x1 = new TestClone();
    TestClone x2 = null;

    x1.a = 10;
    x1.b = 20.5;

    System.out.println("x1: " + x1.a + " " + x1.b);
    try
    {
        x2 = x1.clone(); // clone x1
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
    catch(CloneNotSupportedException e)
    {
        System.out.println("Cloning not allowed.");
    }
    x2.a = 11;
    System.out.println("\nx1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

Question:

```
1  class Test implements Cloneable
2  {
3    int a, b;
4
5     public Test clone() throws CloneNotSupportedException
6     {
7        Test cloned = (Test) super.clone();
8
9        return cloned;
10    }
11 }
```

Question…

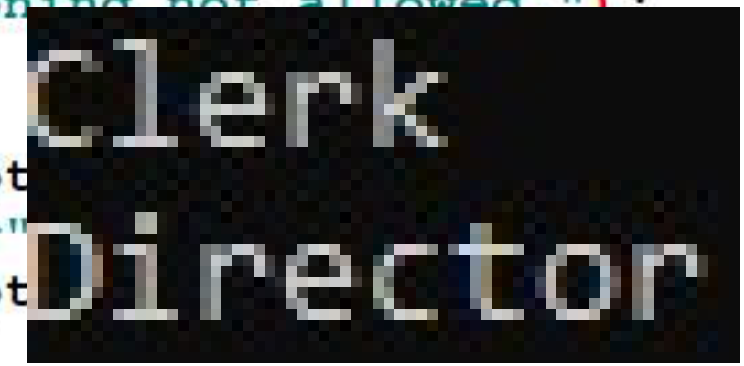
```
13  class CloneDemo
14  {
15    public static void main(String args[])
16    {
17      Test x1 = new Test();
18      Test x2 = new Test();
19      Test x3 = new Test();
20
21      x1.a = 10;  x1.b = 20; x3 = x1;
22
23      try
24      {
25          x2 = x1.clone();
26      }
27      catch(CloneNotSupportedException e)
28      {
29          System.out.println("Cloning not allowed.");
30      }
31      x1.b = 22; x2.b = 25; x3.a = 11;
32      System.out.println("x1: " + x1.a + " " + x1.b);
33      System.out.println("x2: " + x2.a + " " + x2.b);
34      System.out.println("x3: " + x3.a + " " + x3.b);
35    }
36  }
```

```
x1: 11 22
x2: 10 25
x3: 11 22
```

Shallow copy- Limitation

```
class Dept
{
    String empId, grade, desig;

    public Dept(String id, String g, String d)
    { empId = id; grade = g; desig = d; }
}

class Employee implements Cloneable
{
    int id;
    String name;
    Dept dept;

    public Employee(int id, String name, Dept dept)
    {
        this.id = id;
        this.name = name;
        this.dept = dept;
    }
// Default version of clone() method:creates shallow copy of an object
    public Employee clone() throws CloneNotSupportedException

    {  return (Employee)super.clone();  }
}
```

```
27  public class ShallowCopyInJava
28  {
29      public static void main(String[] args)
30      {
31          Dept dept1 = new Dept ("1", "A", "Clerk");
32          Employee emp1 = new Employee (111, "John", dept1);
33          Employee emp2 = null;
34
35          try
36          {
37         // Creating a clone of emp1 and assigning it to emp2
38              emp2 = (Employee) emp1.clone();
39          }
40          catch(CloneNotSupportedException e)
41          {
42              System.out.println("Cloning not allowed ");
43          }
44
45          System.out.println(emp1.dept
46          emp2.dept.desig = "Director"
47          System.out.println(emp1.dept
48      }
49  }
```

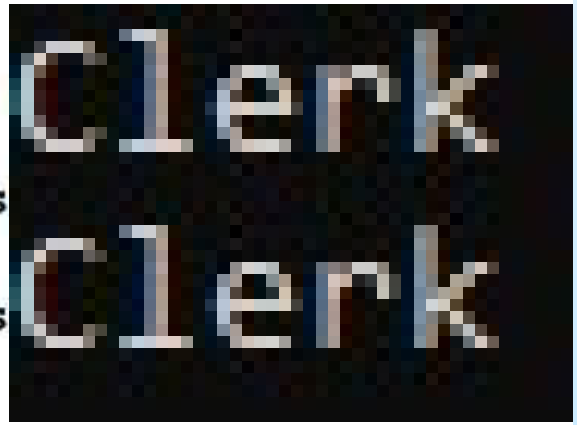


Deep copy

```
1  // Deep Copy Demo
2  class Dept implements Cloneable
3  {
4      String empId, grade, desig;
5
6      public Dept(String id, String g, String d)
7      { empId = id; grade = g; desig = d; }
8
9      //Default version of clone() method.
10     public Dept clone() throws CloneNotSupportedException
11     {
12         return (Dept)super.clone();
13     }
14 }
```

```
15  class Employee implements Cloneable
16  {
17      int id;
18      String name;
19      Dept dept;
20
21      public Employee(int id, String name, Dept dept)
22      {
23          this.id = id;
24          this.name = name;
25          this.dept = dept;
26      }
27
28  // Overriding clone() method to create a deep copy of an object
29      public Employee clone() throws CloneNotSupportedException
30      {
31          Employee emp = (Employee) super.clone();
32          emp.dept = (Dept) dept.clone();
33          return emp;
34      }
35  }
```

```
37  public class DeepCopyInJava
38  {
39      public static void main(String[] args)
40      {
41          Dept dept1 = new Dept("1", "A", "Clerk");
42          Employee emp1 = new Employee(111, "John", dept1);
43          Employee emp2 = null;
44
45          try
46          {
47      // Creating a clone of emp1 and assigning it to emp2
48              emp2 = (Employee) emp1.clone();
49          }
50          catch (CloneNotSupportedException e)
51          {
52              System.out.println("Cloning
53          }
54          System.out.println(emp1.dept.des
55          emp2.dept.desig = "Director";
56          System.out.println(emp1.dept.des
57      }
58  }
```



Garbage Collection

- C++ delete operator: dynamically allocated (using new operator) objects manually released (recycled)
- Java handles deallocation automatically using a technique called *garbage collection*
  - When no references to an object exists (that object is assumed to be no longer needed), the memory occupied by the object is reclaimed (at random)
  - Different Java run-time implementations will take varying approaches to garbage collection

The *finalize*() method

- Finalization is used to define specific actions to be done when an object is about to be reclaimed by garbage collector (a part of JVM)
  - Java run-time system calls this method whenever it is about to recycle an object of that class
  - Example: Before an object is destroyed, any non-Java resources (such as file handle) have to be released
- Done using *finalize*() method

  (protected non-static method of java.lang.Object class)
  - Specify all required actions in this method

```
protected void finalize() throws Throwable
{ // Required code }
```

  - Keyword *protected* prevents access to *finalize*() method by code defined outside its class

Note: The finalize() method approximates the function of a destructor (not supported by Java)

The End