



# Java - Inheritance



# basics

The process of extending a class in object-oriented programming is called inheritance.

- ✓ Class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In the terminology of Java, a *class that is inherited is called a superclass*.
- The class that does the inheriting is called a *subclass*.
- Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- In a subclass you can add new methods and new fields as well as override existing methods in the parent class to change their behaviors.
- Inheritance gives you the opportunity to add some functionality that does not exist in the original class.
- Inheritance can also change the behaviors of the existing class to better suit your needs.
- The subclass and the superclass has an "is-a" relationship.

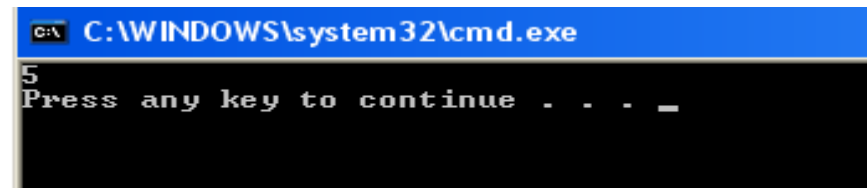
To inherit a class, you can use the **extends** keyword.

- Can only have one superclass for any subclass.
- Java does not support the multiple inheritance. A class can be a superclass of itself.

# Example

```
class A {  
    int x;  
    int y;  
    int get(int p, int q){  
        x=p; y=q; return(0);  
    }  
    void Show(){  
        System.out.println(x);  
    }  
}
```

```
class B extends A{  
    public static void main(String args[]){  
        A a = new A();  
        a.get(5,6);  
        a.Show();  
    }  
    void display(){  
        System.out.println("B");  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
5  
Press any key to continue . . . _
```

When a subclass is derived from a derived class then this mechanism is known as the **multilevel inheritance**.

```
class A {  
    int x;  
    int y;  
    int get(int p, int q){  
        x=p; y=q; return(0);  
    }  
    void Show(){  
        System.out.println(x);  
    }  
}  
  
class B extends A{  
    void Showb(){  
        System.out.println("B");  
    }  
}
```

```
class C extends B{  
    void display(){  
        System.out.println("C");  
    }  
    public static void main(String args[]){  
        A a = new A();  
        a.get(5,6);  
        a.Show();  
    }  
}
```



C:\WINDOWS\system32\cmd.exe



5  
Press any key to continue . . .

# Example

```
C:\WINDOWS\system32\cmd.exe
i and j: 10 0
i and j: 7 0
k: 0
i+j+k: 7
Press any key to continue . . .
```

```
class Base {
    int i, j;
    void showBase() {
        System.out.println("i and j: " + i + " " + j);
    } }

class Child extends Base {
    int k;
    void showChild() {
        System.out.println("k: " + k);
    }

    void sum() {
        System.out.println("i+j+k: " + (i + j + k));
    } }
```

```
public class InheritMain {
    public static void main(String args[]) {
        Base superOb = new Base();
        Child subOb = new Child();

        superOb.i = 10;
        superOb.showBase();
        System.out.println();

        subOb.i = 7;
        subOb.showBase();
        subOb.showChild();
        System.out.println();
        subOb.sum();
    }
}
```

# super keyword

- ❖ You can use super in a subclass to refer to its immediate superclass.

super has two general forms.

1. The first calls the superclass' constructor.
2. The second is used to access a member of the superclass.

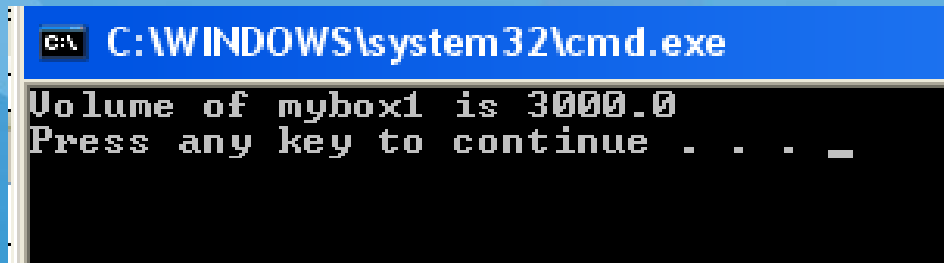
## Using super to Call Superclass Constructors

To call a constructor from its superclass:

*super(parameter-list);*

- parameter-list is defined by the constructor in the superclass.
- super(parameter-list) must be the first statement executed inside a subclass' constructor.

# Example



```
C:\WINDOWS\system32\cmd.exe
Volume of mybox1 is 3000.0
Press any key to continue . . . _
```

```
class Box {
    private double width;
    private double height;
    private double depth;

    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
    }
}
```

```
class BoxWeight extends Box {
    double weight;      // weight of box
    BoxWeight(Box ob) { // pass object to constructor
        super(ob);
    }
}

public class SuperMain {
    public static void main(String args[]) {
        Box mybox1 = new Box(10, 20, 15);

        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
    }
}
```



# Use super to reference members from parent class

```
class Base {  
    int i;  
}  
  
class SubClass extends Base {  
    int i; // this i hides the i in A  
  
    SubClass(int a, int b) {  
        super.i = a;    // i in A  
        i = b;          // i in B  
    }  
  
    void show() {  
  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}
```

```
public class SperMain1 {  
    public static void main(String args[]) {  
  
        SubClass subOb = new SubClass(1, 2);  
        subOb.show();  
    }  
}
```



```
i in superclass: 1  
i in subclass: 2  
Press any key to continue . . .
```



# Overriding method

```
C:\WINDOWS\system32\cmd.exe
k: 3
Press any key to continue . . .
```

- When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.

Consider the following:

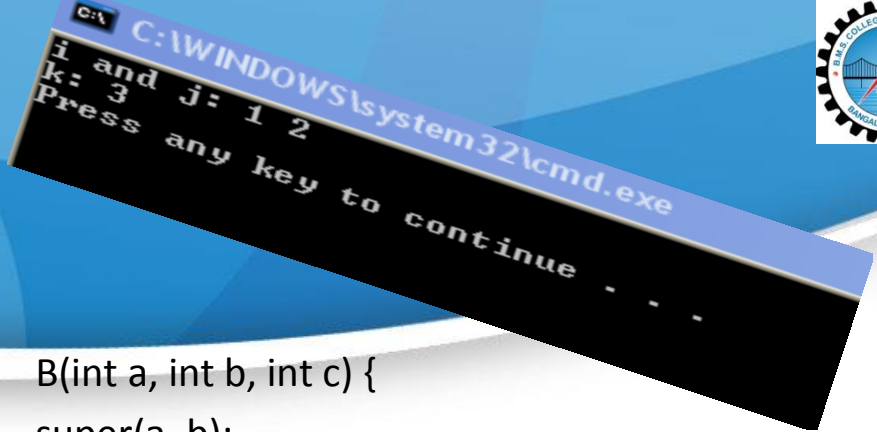
```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

class B extends A {
    int k;
```

```
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // display k – this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[ ]) {
        B subOb = new B(1, 2, 3);
        subOb.show( ); // this calls show() in B
    }
}
```

# Modified version



// Method overriding.

```
class A {
```

```
int i, j;
```

```
A(int a, int b) {
```

```
    i = a;
```

```
    j = b;
```

```
}
```

```
// display i and j
```

```
void show() {
```

```
    System.out.println("i and j: " + i + " " + j);
```

```
}
```

```
}
```

```
class B extends A {
```

```
int k;
```

```
B(int a, int b, int c) {
```

```
    super(a, b);
```

```
    k = c;
```

```
}
```

```
// display k – this overrides show() in A
```

```
void show() {
```

```
    super.show();
```

```
    System.out.println("k: " + k);
```

```
}
```

```
}
```

```
class Override1 {
```

```
public static void main(String args[]) {
```

```
    B subOb = new B (1, 2,3);
```

```
    subOb.show(); // this calls show() in B
```

```
}
```

```
}
```

# Dynamic method dispatch

```
class A {  
    void callme() {  
        System.out.println("Inside A's callme  
        method");  
    }  
}  
class B extends A {  
    void callme() { // override callme()  
        System.out.println("Inside B's callme  
        method");  
    }  
}  
class C extends A {  
    void callme() { // override callme()  
        System.out.println("Inside C's callme  
        method");  
    }  
}  
class Dispatch {  
    public static void main(String args[]) {  
        A a = new A(); // object of type A  
        B b = new B(); // object of type B  
        C c = new C(); // object of type C
```

A r; // obtain a reference of type A

r = a; // r refers to an A object

r.callme(); // calls A's version of callme

r = b; // r refers to a B object

r.callme(); // calls B's version of callme

r = c; // r refers to a C object

r.callme(); // calls C's version of callme

} }

C:\WINDOWS\system32\cmd.exe

```
Inside A's callme method  
Inside B's callme method  
Inside C's callme method  
Press any key to continue . . .
```

# Why to Override method

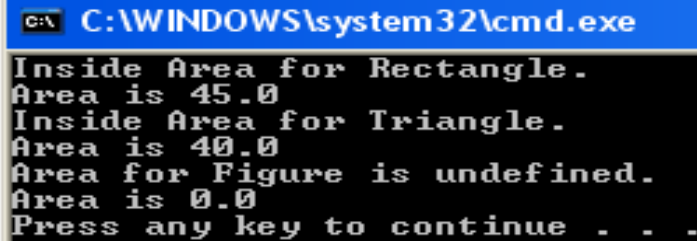
- overridden methods allow Java to support run-time polymorphism.
- Polymorphism is essential to object-oriented programming for allow a general class to specify methods that will be common to all of its derivatives.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- The superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own.
- This allows the subclass the flexibility to define its own methods.
- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.
- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a powerful tool.

# Using Runtime Polymorphism

```
class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b; }
    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0; } }

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b); }
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2; } }

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b); }
```



```
C:\WINDOWS\system32\cmd.exe
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0
Press any key to continue . . .
```

```
double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2; } }

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

        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Figure figref;
        figref = r;

        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
        figref = f;
        System.out.println("Area is " + figref.area());
    } }
```

# Abstract class(1)

- ❖ Sometimes you want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- ❖ That is, want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- ❖ Such a class determines the nature of the methods that the subclasses must implement.
- ❖ One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- ❖ You want to ensure that a subclass does, indeed, override all necessary methods.
- ✓ Java's solution to this problem is the *abstract method*.
  - certain methods be overridden by subclasses by specifying the *abstract* type modifier.
  - To declare an abstract method, use this general form:  
*abstract type name(parameter-list);*
- ✓ Any class that contains *one or more abstract methods must also be declared abstract*.
  - To declare a class **abstract**, simply use the **abstract keyword** in front of the class keyword at the beginning of the class declaration.

# Abstract class(2)

- There can be no objects of an abstract class.
- An abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined.
- Cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

```
abstract class A {  
    abstract void callme();  
    void callmetoo() {  
        System.out.println("This is a concrete method.");  
    }  
}  
class B extends A {  
    void callme() {  
        System.out.println("B's implementation of  
        callme.");  
    }  
}
```

```
class AbstractDemo {  
    public static void main(String args[]) {  
        B b = new B();  
        b.callme();  
        b.callmetoo();  
    }  
}
```



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

```
B's implementation of callme.  
This is a concrete method.  
Press any key to continue . . .
```



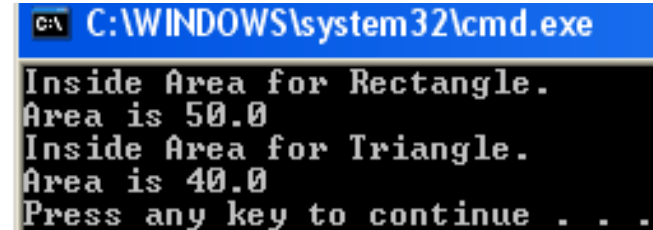
# Using abstract methods and classes

```
abstract class Shape {  
    double height;  
    double width;  
    Shape(double a, double b) {  
        height = a;  
        width = b;  
    }  
    abstract double area();  
}
```

```
class Rectangle extends Shape{  
    Rectangle(double a, double b) {  
        super(a, b);  
    }  
    double area() {  
        System.out.println("Inside Area for Rectangle.");  
        return height * width;  
    }  
}
```

```
class Triangle extends Shape{  
    Triangle(double a, double b) {  
        super(a, b);  
    }  
}
```

```
double area() {  
    System.out.println("Inside Area for Triangle.");  
    return height * width / 2;  
}  
  
public class AbstractMain {  
    public static void main(String args[]) {  
        Rectangle r = new Rectangle(10, 5);  
        Triangle t = new Triangle(10, 8);  
  
        Shape figref;  
        figref = r;  
        System.out.println("Area is " + figref.area());  
        figref = t;  
        System.out.println("Area is " + figref.area());  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
Inside Area for Rectangle.  
Area is 50.0  
Inside Area for Triangle.  
Area is 40.0  
Press any key to continue . . .
```



# Constructors with Inheritance

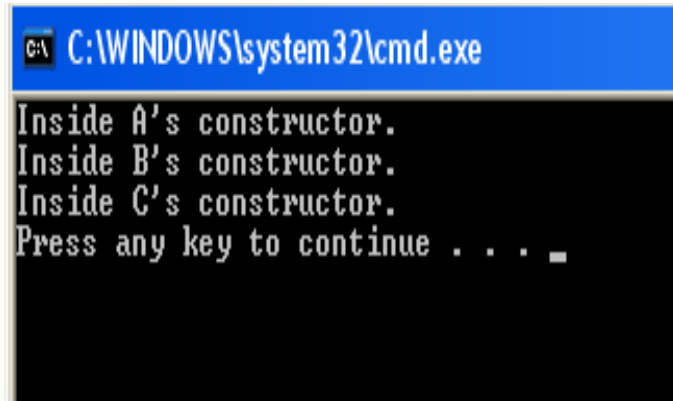
When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?

- a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa?
- ✓ The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.

```
class A {  
    A() {  
        System.out.println("Inside A's constructor.");  
    }  
    // Create a subclass by extending class A.  
    class B extends A {  
        B() {  
            System.out.println("Inside B's constructor.");  
        }  
        // Create another subclass by extending B.  
        class C extends B {  
            C() {  
                System.out.println("Inside C's constructor.");  
            }  
        }  
    }  
}
```

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

```
        C c = new C();  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
Inside A's constructor.  
Inside B's constructor.  
Inside C's constructor.  
Press any key to continue . . .
```

# final with Inheritance

The keyword final has three uses.

1.First, it can be used to create the equivalent of a named constant.

## 2.Use of final to prevent Overriding.

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

## 3.Use of final to prevent Inheritance.

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

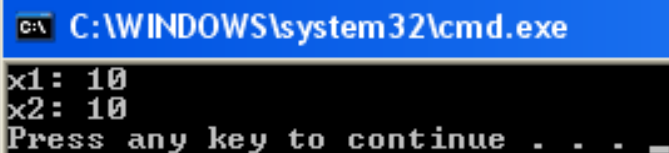
Method	Purpose
Object clone( )	Object clone( ) Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object 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( )	Waits on another thread of execution.
void wait(long milliseconds)	
void wait(long milliseconds, int nanoseconds)	

### Object Class Methods

# Using clone( ) and the Cloneable Interface

```
class TestClone implements Cloneable {  
    int a;  
    public Object clone() {  
        try {  
            return super.clone();  
        }  
        catch (CloneNotSupportedException e) {  
            System.out.println("Cloning not allowed.");  
            return this;  
        }  
    }  
}
```

```
public class ObjectcloneMain {  
    public static void main(String args[]) {  
  
        TestClone x1 = new TestClone();  
        TestClone x2;  
  
        x1.a = 10;  
        // clone() is called directly.  
        x2 = (TestClone) x1.clone();  
  
        System.out.println("x1: " + x1.a);  
        System.out.println("x2: " + x2.a);  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
x1: 10  
x2: 10  
Press any key to continue . . . _
```

# Packages & Interfaces

# Packages & Interfaces

- Java's two most innovative features: **packages** and **interfaces**.
- Packages are containers for classes that are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- Java allows you to fully abstract the interface from its implementation.
- Using interface, you can specify a set of methods which can be implemented by one or more classes.
- The interface, itself, does not actually define any implementation.
- Interfaces have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass .
- Packages and interfaces are two of the basic components of a Java program.

# What is Package ?

- Packages are containers for classes.
- Packages are used to keep the class name space compartmentalized.
- In Java, package is mapped to a folder on your hard drive.

To *define a package*, include a package command as the first statement in a Java source file.

Any classes declared within that file will belong to the specified package.

If you omit the package statement, the class names are put into the default package, which has no name.

This is the general form of the package statement:

**package packageName;**

# Package Example

// A simple package

**package MyPack;**

class Balance {

String name;

double bal;

Balance(String n, double b) {

name = n;

bal = b; }

void show() {

if(bal<0)

System.out.print("--> ");

System.out.println(name + ": Rs" + bal);

}

}

class AccountBalance {

public static void main(String args[]) {

Balance current[] = new Balance[3];

current[0] = new Balance("Sai", 123.23);

current[1] = new Balance("RR", 157.02);

current[2] = new Balance("Raghu", -12.33);

for(int i=0; i<3; i++) current[i].show(); }

}

C:\WINDOWS\system32\cmd.exe

C:\MyPack

Sai: Rs123.23

RR: Rs157.02

--> Raghu: Rs-12.33

Press any key to continue . . .



# Find Packages

**How does the Java run-time system know where to look for packages that you create?**

The answer has two parts.

1.First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.

2.Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.

For example, consider the following package specification.

**package MyPack;**

3.In order for a program to find MyPack, one of two things must be true. Either the

4.program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.

**The easiest way to try is**

1.simply create the package directories below your current development directory,

2.put the .class files into the appropriate directories and

3.then execute the programs from the development directory.

# Access Protection

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.

The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses 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

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

- ✓ Anything declared public can be accessed from anywhere.
- ✓ Anything declared private cannot be seen outside of its class.
- ✓ When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- ✓ If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

# How to use packages

1. Referring to a package member by its qualified name:

```
graphics.Circle myCircle = new graphics.Circle();
```

2. Importing a package member:

```
import graphics.Circle;
```

```
...
```

```
Circle myCircle = new Circle();
```

```
graphics.Rectangle myR = new graphics.Rectangle();
```

3. Importing an entire package:

```
import graphics.*;
```

```
...
```

```
Circle myCircle = new Circle();
```

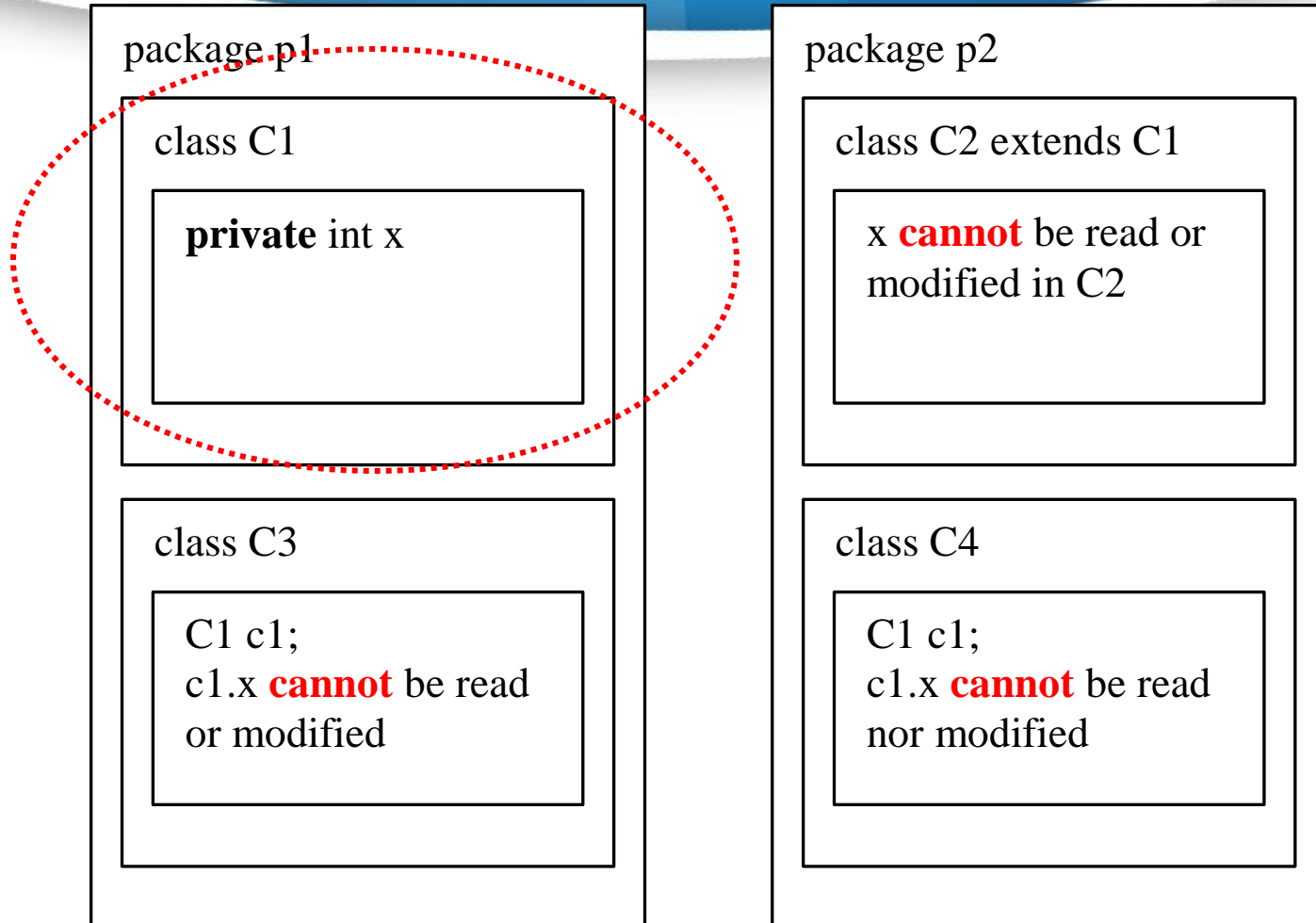
```
Rectangle myRectangle = new Rectangle();
```

# Access to members of the classes

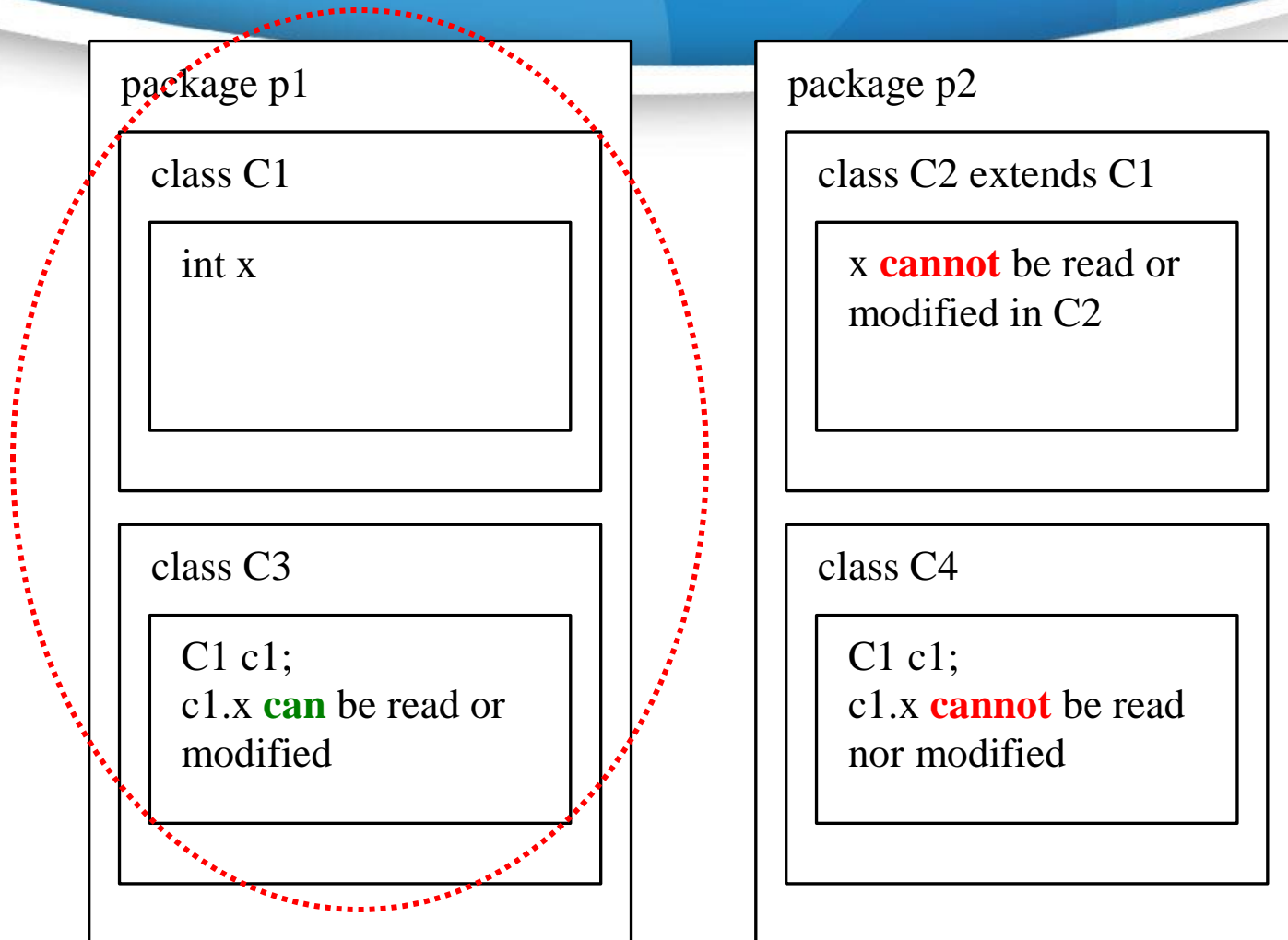
	Private	Default	Protected	Public
The same class	Yes	Yes	Yes	Yes
Sub class in the package	No	Yes	Yes	Yes
Non sub class in the package	No	Yes	Yes	Yes
Subclass in another package	No	No	Yes	Yes
Non sub class in another package	No	No	No	Yes

Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time.

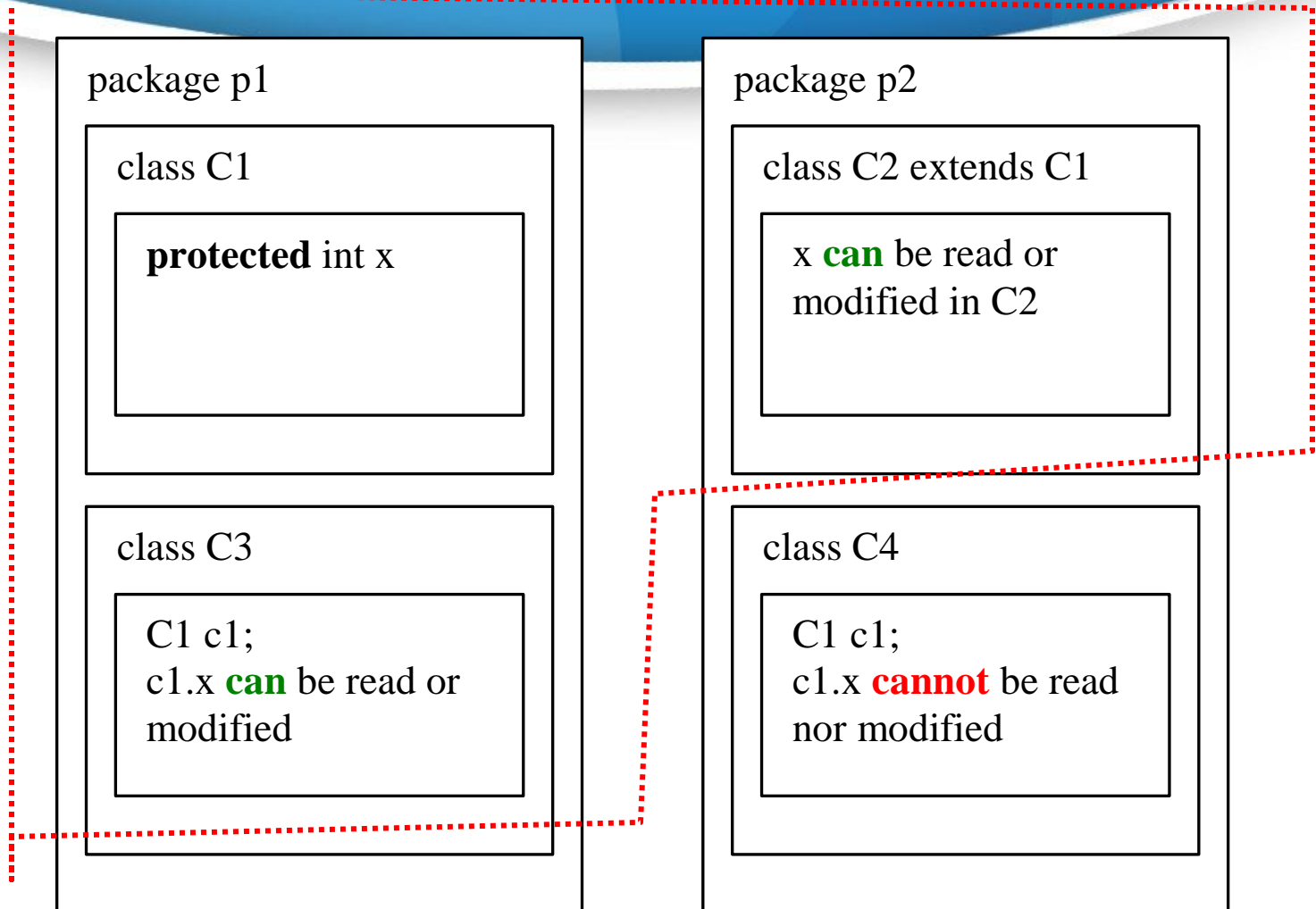
# Access modifier **private**



# Default access modifier



# Access modifier **protected**



# Access modifier `public`

package p1

class C1

**public** int x

class C3

C1 c1;  
c1.x **can** be read or  
modified

package p2

class C2 extends C1

x **can** be read or  
modified in C2

class C4

C1 c1;  
c1.x **can** be read nor  
modified



# Example(1)

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

```
package p1;

class Derived extends Protection {

    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

# Example(2)

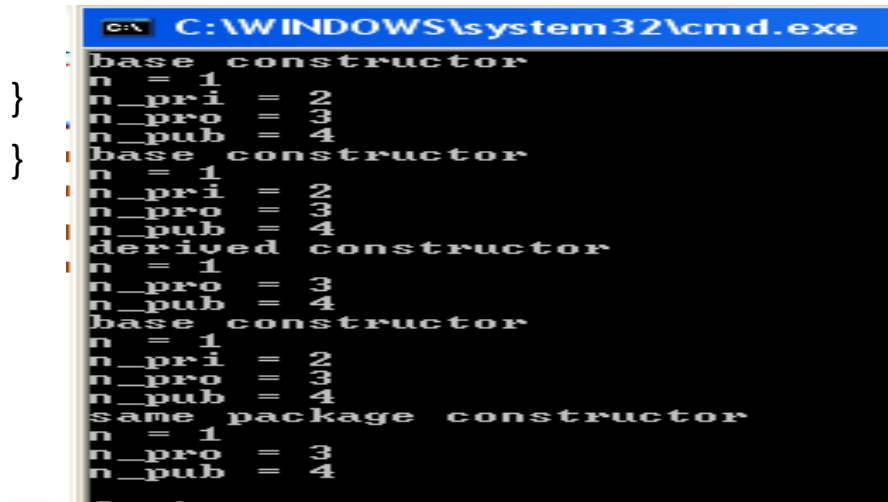
```
package p1;

class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String args[])
    {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived constructor
n = 1
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
same package constructor
n = 1
n_pro = 3
n_pub = 4
```

# Importing packages

- Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.
- ✓ For this reason, Java includes the import statement to bring certain classes, or entire packages, into visibility.
  - Once imported, a class can be referred to directly, using only its name.
  - The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.
  - In a Java source file, import statements occur immediately following the package statement and before any class definitions.

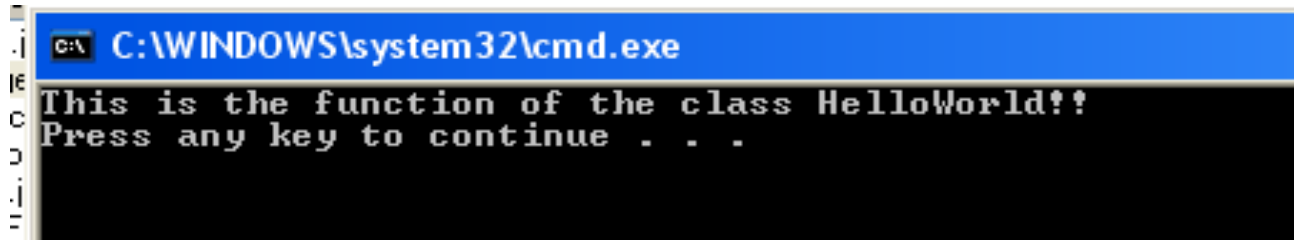
This is the general form of the import statement:

```
import pkg1[.pkg2].(classname|*);
```

# Example

```
package importpackage;  
public class HelloWorld {  
    public void show(){  
        System.out.println("This is the function of the class HelloWorld!!");  
    }  
}
```

```
import importpackage.*;  
class CallPackage{  
    public static void main(String[] args){  
        HelloWorld h2=new HelloWorld();  
        h2.show();  
    }  
}
```

A screenshot of a Windows command prompt window. The title bar is blue and reads "C:\WINDOWS\system32\cmd.exe". The command prompt shows the output of the program: "This is the function of the class HelloWorld!!" followed by "Press any key to continue . . .".

```
C:\WINDOWS\system32\cmd.exe  
This is the function of the class HelloWorld!!  
Press any key to continue . . .
```

# Interfaces(1)

- ❖ An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts.
  - A class describes the attributes and behaviors of an object.
  - An interface contains behaviors that a class implements.

## **An interface is similar to a class in the following ways:**

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

## **However, an interface is different from a class in several ways, including:**

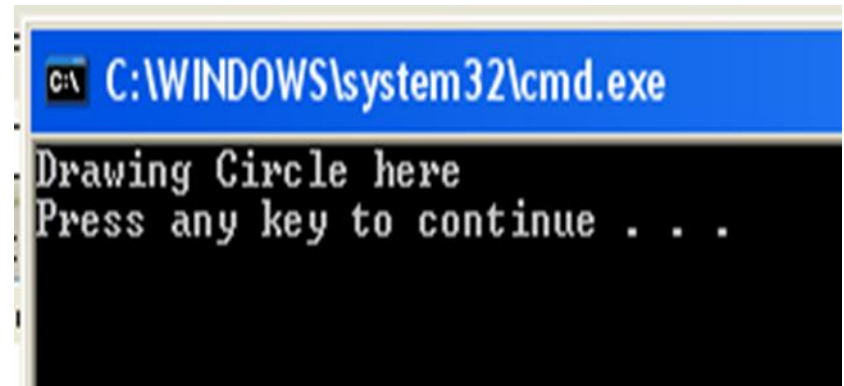
- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

# Interfaces(2)

- ❖ Using the keyword interface, you can fully abstract a class' interface from its implementation.
  - That is, using interface, you can specify what a class must do, but not how it does it.
  - Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
  - In practice, this means that you can define interfaces which don't make assumptions about how they are implemented.
  - Once it is defined, any number of classes can implement an interface.
  - Also, one class can implement any number of interfaces.
  - To implement an interface, a class must create the complete set of methods defined by the interface.
  - However, each class is free to determine the details of its own implementation.
  - By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
  - Interfaces are designed to support dynamic method resolution at run time.
  - Interfaces are designed to disconnect the definition of a method or set of methods from the inheritance hierarchy.
  - Interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.
  - This is where the real power of interfaces is realized.

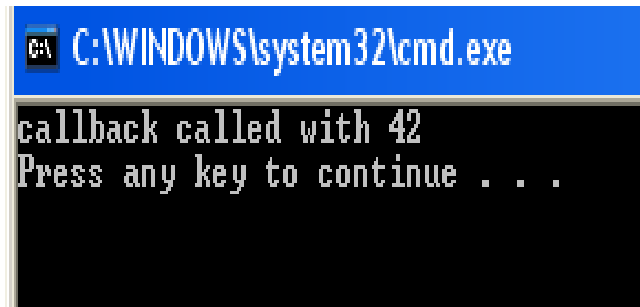
# Simple Interface

```
public class InterfaceMain {  
    public static void main(String[] args) {  
        shape circleshape=new circle();  
        circleshape.Draw();  
    }  
}  
  
interface shape  
{  
    public String baseclass="shape";  
    public void Draw();  
}  
  
class circle implements shape  
{  
    public void Draw() {  
        System.out.println("Drawing Circle here");  
    }  
}
```



# Accessing Implementations Through Interface References

```
interface MyInterface {  
    void callback(int param);  
}  
  
class Client implements MyInterface{  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " +  
p);  
    }  
}  
  
public class InterfaceMain {  
    public static void main(String args[]) {  
        MyInterface c = new Client();  
        c.callback(42);  
  
    }  
}
```



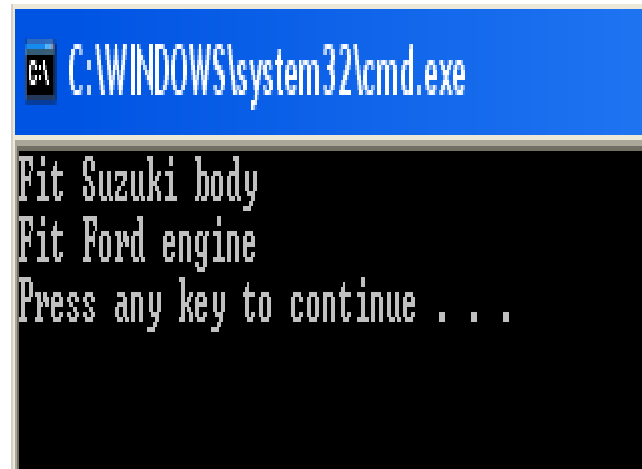
```
C:\WINDOWS\system32\cmd.exe  
callback called with 42  
Press any key to continue . . .
```



# Partial implementation

- Java does not support multiple inheritance, basically.
- But to support multiple inheritance, partially, Java introduced "interfaces".
- An interface is not altogether a new one; just it is a special flavor of abstract class where all methods are abstract. That is, an interface contains only abstract methods

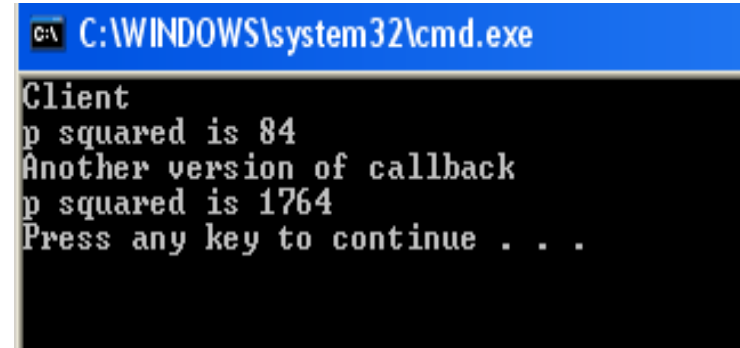
```
interface Suzuki    {  
    public abstract void body();    }  
  
interface Ford      {  
    public abstract void engine();  }  
  
public class MotorCar implements Suzuki, Ford  
{  
    public void body()    {  
        System.out.println("Fit Suzuki body");  
    }  
    public void engine()  {  
        System.out.println("Fit Ford engine");  
    }  
  
    public static void main(String args[])  
    {  
        MotorCar mc1 = new MotorCar();  
        mc1.body();  
        mc1.engine();  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
  
Fit Suzuki body  
Fit Ford engine  
Press any key to continue . . .
```

# Polymorphic and interface

```
interface MyInterface {  
    void callback(int param);  
}  
  
class Client implements MyInterface{  
    public void callback(int p) {  
        System.out.println("Client");  
        System.out.println("p squared is " + (p * 2)); } }  
  
class AnotherClient implements MyInterface{  
    public void callback(int p) {  
        System.out.println("Another version of callback");  
        System.out.println("p squared is " + (p * p)); } }  
  
class TestIface2 {  
    public static void main(String args[]) {  
        MyInterface c = new Client();  
        AnotherClient ob = new AnotherClient();  
        c.callback(42);  
        c = ob;  
        c.callback(42); } }
```

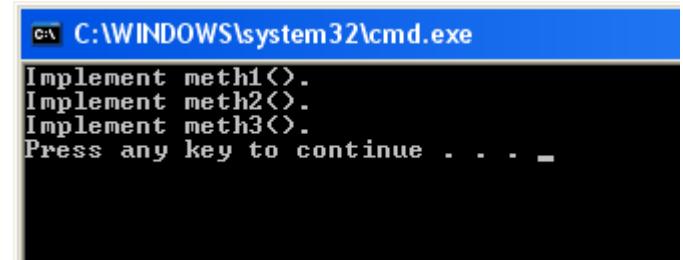


```
C:\WINDOWS\system32\cmd.exe  
Client  
p squared is 84  
Another version of callback  
p squared is 1764  
Press any key to continue . . .
```

# Interfaces Can Be Extended

```
interface InterfaceA {  
    void meth1();  
    void meth2();  
}  
  
interface InterfaceB extends InterfaceA {  
    void meth3();  
}  
  
class MyClass implements InterfaceB {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}
```

```
public class IntExtMain {  
    public static void main(String arg[]) {  
        MyClass ob = new MyClass();  
        ob.meth1();  
        ob.meth2();  
        ob.meth3();  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
Implement meth1<>.  
Implement meth2<>.  
Implement meth3<>.  
Press any key to continue . . . _
```

# EXCEPTION HANDLING

# Introduction(1)

- ❖ A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
  - When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
  - That method may choose to handle the exception itself, or pass it on.
  - Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.

# Introduction(2)

- ❖ **Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.**
- Program statements that you want to monitor for exceptions are contained within a try block.
- If an exception occurs within the try block, it is thrown.
- Your code can catch this exception (using catch) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword throw.
- Any exception that is thrown out of a method must be specified as such by a throws clause.
- Any code that absolutely must be executed before a method returns is put in a finally block.

This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors      }  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1    }  
    finally {  
        // block of code to be executed before try block ends  }
```

# Exception Types

- ❖ All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy.

Throwable are two subclasses that partition exceptions into two distinct branches.

*One branch is headed by Exception.*

- This class is used for exceptional conditions that user programs should catch.
- This is also the class that you will subclass to create your own custom exception types.
- There is an important subclass of Exception, called Runtime Exception.
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

*The other branch is topped by Error.*

- which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
- Stack overflow is an example of such an error.

# Uncaught Exception

what happens when you don't handle Exception ?

This small program includes an expression that intentionally causes a divide-by-zero error.

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;    }    }
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)
```



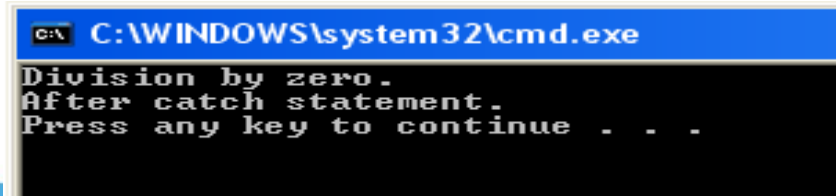
# Using try and catch

- ❖ The default exception handler provided by the Java run-time system is useful for debugging, developer usually want to handle an exception himself.
- It provides two benefits.
  - First, it allows you to fix the error.
  - Second, it prevents the program from automatically terminating.
- ✓ To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block.
- ✓ Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

# Modified version

To illustrate how easily this can be done, the following program includes a try block and a catch clause which processes the `ArithmeticException` generated by the division-by-zero error:

```
class Exc2 {  
    public static void main(String args[]) {  
        int d, a;  
        try { // monitor a block of code.  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) { // catch divide-by-zero error  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
Division by zero.  
After catch statement.  
Press any key to continue . . .
```

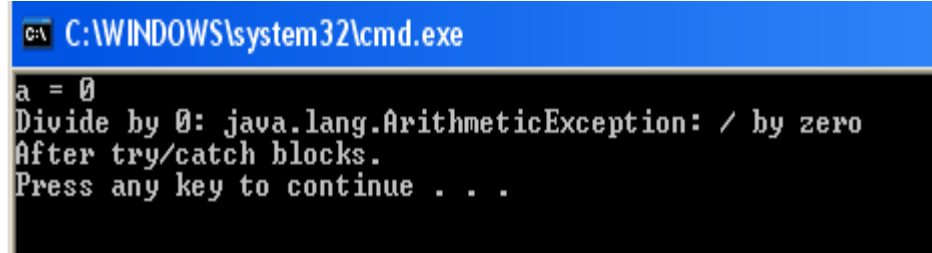
# Multiple catch clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

The following example traps two different exception types:

```
class MultiCatch {  
    public static void main(String args[]) {  
        try {  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            int c[] = { 1 };  
            c[42] = 99;        }  
        catch(ArithmeticException e) {  
            System.out.println("Divide by 0: " + e);    }  
    }  
}
```

```
        catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array index oob: " + e);  
            System.out.println("After try/catch blocks.");  
        }  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
a = 0  
Divide by 0: java.lang.ArithmeticException: / by zero  
After try/catch blocks.  
Press any key to continue . . .
```

# Nested try statements

- The try statement can be nested.
- That is, a try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.

# Example

```
class NestTry {
public static void main(String args[]) {
try {

    int a = args.length;

    /* If no command-line args are present,
    the following statement will generate
    a divide-by-zero exception. */
    int b = 42 / a;
    System.out.println("a = " + a);
try {    // nested try block
    /* If one command-line arg is used,
    then a divide-by-zero exception
    will be generated by the following code. */
```

```
if(a==1) a = a/(a-a); // division by zero
```

```
F:\RR\JAVA -2012 -Odd>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

F:\RR\JAVA -2012 -Odd>java NestTry one
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

F:\RR\JAVA -2012 -Odd>java NestTry one two
a = 2
Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException: 42

F:\RR\JAVA -2012 -Odd>java NestTry one two three
a = 3

F:\RR\JAVA -2012 -Odd>java NestTry one two three four
a = 4

F:\RR\JAVA -2012 -Odd>
```

```
/* If two command-line args are used,
then generate an out-of-bounds exception. */
    if(a==2) {
    int c[] = { 1 };
    c[42] = 99; // generate an out-of-bounds
                exception
    }
    }
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}
catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```



# throw

It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

**throw ThrowableInstance;**

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

There are two ways you can obtain a Throwable object:

1. using a parameter into a catch clause, or
2. creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

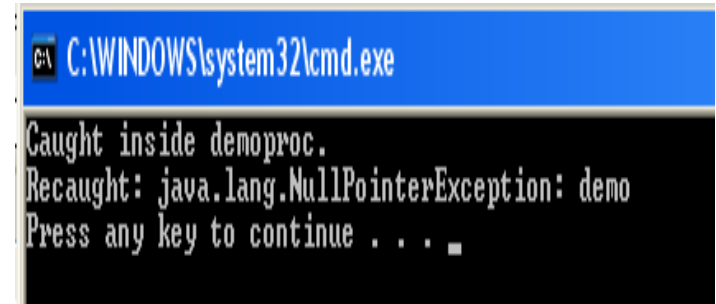
The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception.

If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on.

If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

# Example

```
class ThrowDemo {  
    static void demoproc() {  
        try {  
            throw new NullPointerException("demo");  
        }  
        catch(NullPointerException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e; // rethrow the exception  
        }  
    }  
    public static void main(String args[]) {  
        try {  
            demoproc();  
        }  
        catch(NullPointerException e) {  
            System.out.println("Recaught: " + e);  
        }  
    }  
}
```



# throws

- ❖ If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- ✓ This can be done by including a throws clause in the method's declaration.
  - A throws clause lists the types of exceptions that a method might throw.
  - This is necessary for all exceptions, except those of type `Error` or `RuntimeException`, or any of their subclasses.
  - All other exceptions that a method can throw must be declared in the throws clause.

This is the general form of a method declaration that includes a throws clause:

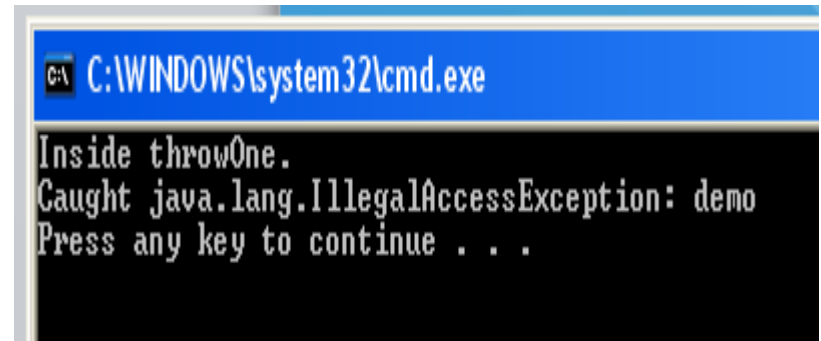
```
type method-name(parameter-list) throws exception-list  
{  
    // body of method  
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.



# Example

```
class ThrowsDemo {  
    static void throwOne() throws IllegalAccessException  
    {  
        System.out.println("Inside throwOne.");  
        throw new IllegalAccessException("demo");  
    }  
    public static void main(String args[]) {  
        try {  
            throwOne();  
        } catch (IllegalAccessException e) {  
            System.out.println("Caught " + e);  
        }  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
Inside throwOne.  
Caught java.lang.IllegalAccessException: demo  
Press any key to continue . . .
```

# finally

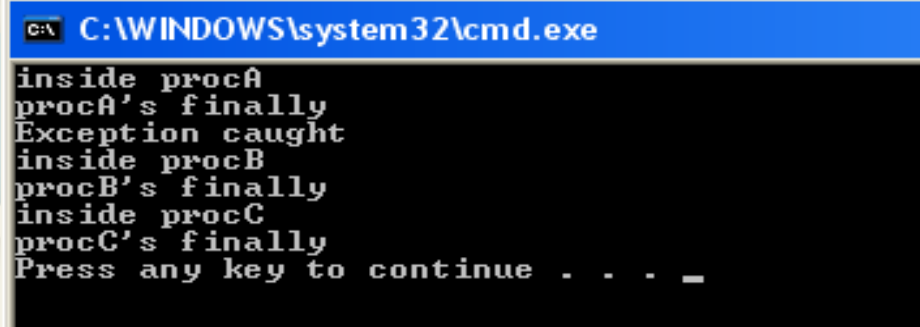


- ❖ finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
  - The finally block will execute whether or not an exception is thrown.
  - If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
  - Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
  - This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
  - The finally clause is optional.
  - However, each try statement requires at least one catch or a finally clause.



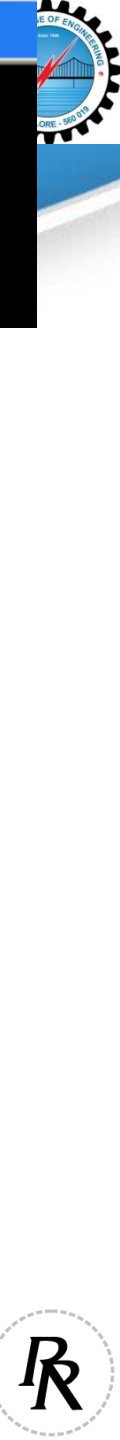
# Example

```
class FinallyDemo {  
    // Through an exception out of the method.  
    static void procA() {  
        try {  
            System.out.println("inside procA");  
            throw new RuntimeException("demo");  
        }  
        finally {  
            System.out.println("procA's finally");  
        }  
    }  
    // Return from within a try block.  
    static void procB() {  
        try {  
            System.out.println("inside procB");  
            return;  
        }  
        finally {  
            System.out.println("procB's finally");  
        }  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
inside procA  
procA's finally  
Exception caught  
inside procB  
procB's finally  
inside procC  
procC's finally  
Press any key to continue . . . _
```

```
// Execute a try block normally.  
static void procC() {  
    try {  
        System.out.println("inside procC");  
    } finally {  
        System.out.println("procC's finally");  
    }  
}  
  
public static void main(String args[]) {  
    try {  
        procA();  
    } catch (Exception e) {  
        System.out.println("Exception caught");  
    }  
    procB();  
    procC();  
}
```



# Java Built-in Exceptions

- ❖ The most general of these exceptions are subclasses of the standard type Runtime Exception. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.
- ❖ The exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.

## Exception

ArithmeticException

ArrayIndexOutOfBoundsException

ArrayStoreException

ClassCastException

IllegalArgumentException

IllegalMonitorStateException

IllegalStateException

IllegalThreadStateException

IndexOutOfBoundsException

NegativeArraySizeException

## Meaning

Arithmetic error, such as divide-by-zero.

Array index is out-of-bounds.

Assignment to an array element of an incompatible type.

Invalid cast.

Illegal argument used to invoke a method.

Illegal monitor operation, such as waiting on an unlocked thread.

Environment or application is in incorrect state.

Requested operation not compatible with current thread state.

Some type of index is out-of-bounds.

Array created with a negative size.

## Exception

NullPointerException

NumberFormatException

SecurityException

StringIndexOutOfBoundsException

UnsupportedOperationException

## Meaning

Invalid use of a null reference.

Invalid conversion of a string to a numeric format.

Attempt to violate security.

Attempt to index outside the bounds of a string.

An unsupported operation was encountered.

## Exception

ClassNotFoundException

CloneNotSupportedException

IllegalAccessException

InstantiationException

InterruptedException

NoSuchFieldException

NoSuchMethodException

## Meaning

Class not found.

Attempt to clone an object that does not implement the Cloneable interface.

Access to a class is denied.

Attempt to create an object of an abstract class or interface.

One thread has been interrupted by another thread.

A requested field does not exist.

A requested method does not exist.

**Java's Checked Exceptions Defined in java.lang**

# Creating your own exceptions

- ❖ Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to applications.
- ✓ This is quite easy to do: just define a subclass of Exception.
  - The Exception class does not define any methods of its own.
  - It inherits those methods provided by Throwable.
  - Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.



## Method

Throwable fillInStackTrace()

Throwable getCause()

String getLocalizedMessage()

String getMessage()

StackTraceElement[] getStackTrace()

Throwable initCause(Throwable  
                            *causeExc*)

## Description

Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.

Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. Added by Java 2, version 1.4.

Returns a localized description of the exception.

Returns a description of the exception.

Returns an array that contains the stack trace, one element at a time as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name. Added by Java 2, version 1.4

Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. Added by Java 2, version 1.4



## Method

`void printStackTrace()`

`void printStackTrace(PrintStream  
                  stream)`

`void printStackTrace(PrintWriter  
                  stream)`

`void setStackTrace(StackTraceElement  
                  elements[ ])`

`String toString()`

## Description

Displays the stack trace.

Sends the stack trace to the specified stream.

Sends the stack trace to the specified stream.

Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use. Added by Java 2, version 1.4

Returns a `String` object containing a description of the exception. This method is called by `println()` when outputting a `Throwable` object.

## The Methods Defined by Throwable (continued)

# Built-in exception example

// File Name **InsufficientFundsException.java**

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

```
public class InsufficientFundsException extends  
Exception
```

```
{  
    private double amount;  
    public InsufficientFundsException(double  
amount)  
    {  
        this.amount = amount;  
    }  
    public double getAmount()  
    {  
        return amount;  
    }  
}
```

// File Name **CheckingAccount.java**

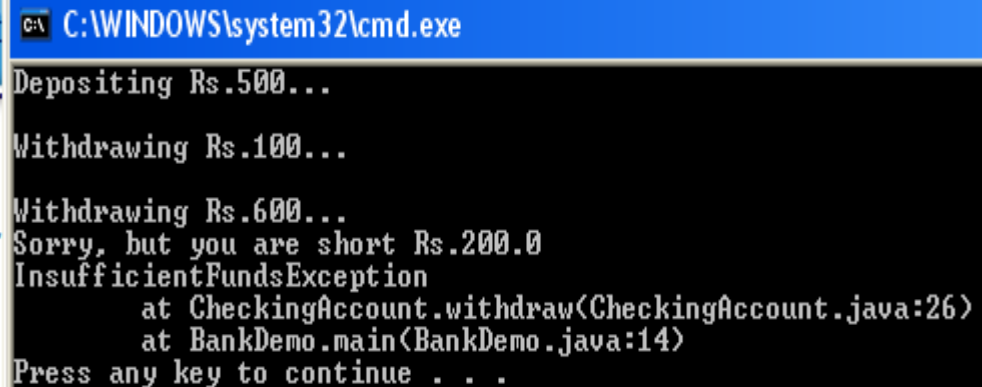
```
import java.io.*;
```

```
public class CheckingAccount
```

```
{    private double balance;  
    private int number;  
    public CheckingAccount(int number)  
    {  
        this.number = number;  
    }  
    public void deposit(double amount)  
    {  
        balance += amount;  
    }  
    public void withdraw(double amount) throws  
        InsufficientFundsException  
    {  
        if(amount <= balance)    {  
            balance -= amount;  
        }  
        else  
        {  
            double needs = amount - balance;  
            throw new InsufficientFundsException(needs);  
        } }  
    public double getBalance()    {  
        return balance;  
    }  
    public int getNumber()  
    {  
        return number;  
    } }
```

```
// File Name BankDemo.java
public class BankDemo
{
    public static void main(String [] args)
    {
        CheckingAccount c = new CheckingAccount(101);
        System.out.println("Depositing Rs. 500...");
        c.deposit(500.00);
        try
        {
            System.out.println("\nWithdrawing Rs. 100...");
            c.withdraw(100.00);
            System.out.println("\nWithdrawing Rs. 600...");
            c.withdraw(600.00);
        }
    }
}
```

```
        catch(InsufficientFundsException e)
        {
            System.out.println("Sorry, but you are short Rs. "
                               + e.getAmount());
            e.printStackTrace();
        }
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
Depositing Rs.500...
Withdrawing Rs.100...
Withdrawing Rs.600...
Sorry, but you are short Rs.200.0
InsufficientFundsException
    at CheckingAccount.withdraw(CheckingAccount.java:26)
    at BankDemo.main(BankDemo.java:14)
Press any key to continue . . .
```

# UNIT – 3 ...