

**G.PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY**

**Kurnool – 518002**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Lecture notes of**

**OBJECT ORIENTED PROGRAMMING USING THROUGH JAVA**

**Prepared by: P. Rama Rao, Asst. Prof.**

**Dept of CSE**

## Unit -3

### Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.
// Create a superclass.
class A {
    int i, j;
    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
}
// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk() {
        System.out.println("k: " + k);
    }
    void sum() {
        System.out.println("i+j+k: " + (i+j+k));
    }
}

class SimpleInheritance {
    public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();
        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();
        /* The subclass has access to all public members of
        its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

```
}  
}
```

The output from this program is shown here:

Contents of superOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**. Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass. The general form of a **class** declaration that inherits a superclass is shown here:

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

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

### Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain  
private to their class.
```

```
This program contains an error and will not  
compile.
```

```
*/
```

```
// Create a superclass.
```

```
class A {  
int i; // public by default  
private int j; // private to A  
void setij(int x, int y) {  
i = x;  
j = y;  
}  
}
```

```
// A's j is not accessible here.
```

```
class B extends A {  
int total;
```

```

void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}

```

This program will not compile because the reference to **j** inside the **sum()** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

### A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```

class RefDemo {
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}

```

### Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box()**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that

keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. **super** has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Each use is examined here.

### Using **super** to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of **super**:  
`super(arg-list);`

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor. To see how **super( )** is used, consider this improved version of the **BoxWeight( )** class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
}
```

Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box( )** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired. In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super( )** is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

### Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```

// Extend BoxWeight to include shipping costs.
// Start with Box.
class Box {
private double width;
private double height;
private double depth;

// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Add weight.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
}

```

```

// default constructor
BoxWeight() {
super();
weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}

// Add shipping costs.
class Shipment extends BoxWeight {
double cost;
// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}
// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}
// default constructor
Shipment() {
super();
cost = -1;
}
// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);
cost = c;
}
}

class DemoShipment {
public static void main(String args[]) {
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "

```

```

+ shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);

vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}
}

```

The output of this program is shown here:

```

Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code. This example illustrates one other important point: **super()** always refers to the constructor in the closest superclass. The **super()** in **Shipment** calls the constructor in **BoxWeight**. The **super()** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a subclass needs parameters of its own.

### When Constructors Are Called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given 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. Further, since **super()** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```

// Demonstrate when constructors are called.
// Create a super class.
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();
}
}

```

The output from this program is shown here:

Inside A's constructor

Inside B's constructor

Inside C's constructor

As you can see, the constructors are called in order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

### Method Overriding

In a class hierarchy, 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:

// 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() {
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
}
}

```

The output produced by this program is shown here:

k: 3

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass' version. This allows all instance variables to be displayed.

```

class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}

```

If you substitute this version of **A** into the previous program, you will see the following output:

i and j: 1 2

k: 3

Here, **super.show()** calls the superclass version of **show()**.

### Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will 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. This is the case with the class **Figure** used in the preceding example. The definition of **area()** is simply a placeholder. It will not compute and display the area of any type of object..

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message.

While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method*. You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you 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**. Here is a simple example of a class with an abstract method, followed by a class which implements that method:

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

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use

of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

### Using **final** with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

### Using **final** to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

The following fragment illustrates **final**:

```
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!");
    }
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result. Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

### Using **final** to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

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

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

## Packages

A unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

## Defining a Package

To create a package is quite easy: simply 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. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven’t had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java\awt\image** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

## Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:

```
package MyPack
```

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**. When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is

```
C:\MyPrograms\Java\MyPack
```

Then the class path to **MyPack** is

```
C:\MyPrograms\Java
```

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

### A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// 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 + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];

current[0] = new Balance("K. J. Fielding", 123.23);
```

```
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command. As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

**AccountBalance** must be qualified with its package name.

### Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. 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. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

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

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (\*), which indicates that the Java compiler should import the entire package.

This code fragment shows both forms in use:

```
import java.util.Date;
```

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

### Interfaces

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 that 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. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since 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.

### Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback()** that takes a single integer parameter.

```
interface Callback {
    void callback(int param);
}
```

### Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the



methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

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

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {  
// Implement Callback's interface  
public void callback(int p) {  
System.out.println("callback called with " + p);  
}  
}
```

Notice that **callback()** is declared using the **public** access specifier.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback()** and adds the method **nonInterfaceMeth()**:

```
class Client implements Callback {  
// Implement Callback's interface  
public void callback(int p) {  
System.out.println("callback called with " + p);  
}  
void nonInterfaceMeth() {  
System.out.println("Classes that implement interfaces " +  
"may also define other members, too.");  
}  
}
```

### Partial Implementations

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

```
abstract class Incomplete implements Callback {  
int a, b;  
void show() {  
System.out.println(a + " " + b);  
}  
// ...  
}
```

Here, the class **Incomplete** does not implement **callback()** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback()** or be declared **abstract** itself.

### Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified. Here is an example that demonstrates a nested interface:

```
// A nested interface example.
// This class contains a member interface.
class A {

    // this is a nested interface
    public interface NestedIF {
        boolean isNotNegative(int x);
    }
}

// B implements the nested interface.
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false : true;
    }
}

class NestedIFDemo {
    public static void main(String args[]) {
        // use a nested interface reference
        A.NestedIF nif = new B();
        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}
```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying implements **A.NestedIF**. Notice that the name is fully qualified by the enclosing class' name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations ) providing access protection and namespace management.

Some of the existing packages in Java are –

- **java.lang** – bundles the fundamental classes
- **java.io** – classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

### Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

### Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* –

```
/* File name : Animal.java */
```

```
package animals;
```

```
interface Animal {  
    public void eat();  
    public void travel();  
}
```

Now, let us implement the above interface in the same package *animals* –  
package animals;

```
/* File name : MammalInt.java */
```

```
public class MammalInt implements Animal {  
  
    public void eat() {  
        System.out.println("Mammal eats");  
    }  
  
    public void travel() {  
        System.out.println("Mammal travels");  
    }  
  
    public int noOfLegs() {  
        return 0;  
    }  
  
    public static void main(String args[]) {  
        MammalInt m = new MammalInt();  
        m.eat();  
    }  
}
```

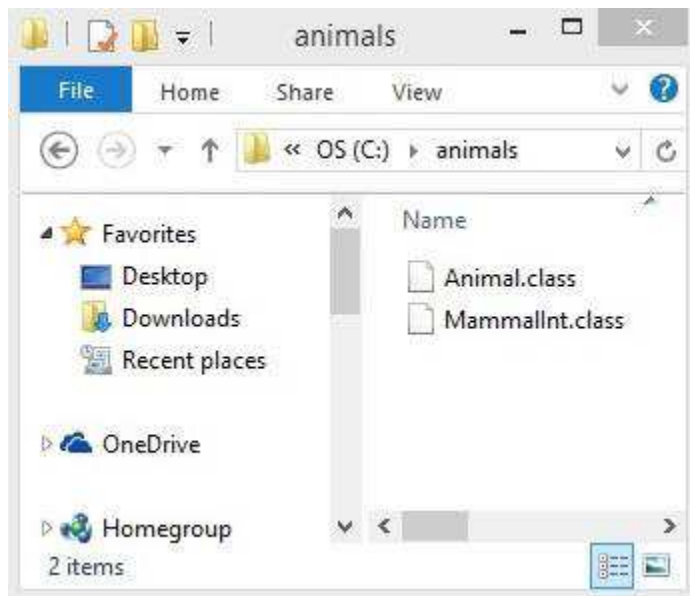
```
m.travel();  
}  
}
```

Now compile the java files as shown below –

```
$ javac -d . Animal.java
```

```
$ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



You can execute the class file within the package and get the result as shown below.

Mammal eats

Mammal travels

The import Keyword

If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;

public class Boss {

    public void payEmployee(Employee e) {

        e.mailCheck();

    }

}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example –

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (\*). For example –

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example –

```
import payroll.Employee;
```

**Note** – A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

### The Directory Structure of Packages

Two major results occur when a class is placed in a package –

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java –

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**.

For example –

```
// File Name : Car.java
```

```
package vehicle;
```

```
public class Car {
```

```
    // Class implementation.
```

```
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs –

```
....\vehicle\Car.java
```

Now, the qualified class name and pathname would be as follows –

- Class name → vehicle.Car
- Path name → vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names.

**Example** – A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

**Example** – The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this –

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**.

For example –

```
// File Name: Dell.java
```

```
package com.apple.computers;
```

```
public class Dell {  
  
}
```

```
class Ups {  
  
}
```

Now, compile this file as follows using -d option –

```
$javac -d . Dell.java
```

The files will be compiled as follows –

```
.\com\apple\computers\Dell.class
```

```
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in `\com\apple\computers\` as follows –

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as –

```
<path-one>\sources\com\apple\computers\Dell.java
```

```
<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give access to the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, `<path-two>\classes`, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say `<path-two>\classes` is the class path, and the package name is `com.apple.computers`, then the compiler and JVM will look for .class files in `<path-two>\classes\com\apple\computers`.



A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

#### Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell) –

- In Windows → C:\> set CLASSPATH
- In UNIX → % echo \$CLASSPATH

To delete the current contents of the CLASSPATH variable, use –

- In Windows → C:\> set CLASSPATH =
- In UNIX → % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable –

In Windows → set CLASSPATH = C:\users\jack\java\classes

In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- Checked exceptions:** A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;
public class FileNotFound_Demo {
    public static void main(String args[]){
        File file=new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

If you try to compile the above program, you will get the following exceptions.

```
C:\>javac FileNotFound_Demo.java
```

```
FileNotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught
or declared to be thrown
```

```
FileReader fr = new FileReader(file);
```

```
^
```

```
1 error
```

**Note:** Since the methods **read()** and **close()** of **FileReader** class throws **IOException**, you can observe that the compiler notifies to handle **IOException**, along with **FileNotFoundException**.

□ **Unchecked exceptions:** An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsException* occurs.

```
public class Unchecked_Demo {
    public static void main(String args[]){
        int num[]={1,2,3,4};
        System.out.println(num[5]);
    }
}
```

If you compile and execute the above program, you will get the following exception.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
```

```
at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

□ **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the

runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class. Following is a list of most common checked and unchecked Java's Built-in Exceptions.

### **Built-in Exceptions**

Java defines several exception classes inside the standard package **java.lang**.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

<b>Exception</b>	<b>Description</b>
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with the current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang. Exception

## Exception

ClassNotFoundException  
CloneNotSupportedException

IllegalAccessException  
InstantiationException

InterruptedException

NoSuchFieldException

## Description

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.

## Exceptions Methods

Following is the list of important methods available in the Throwable class

. Sr. No.	Methods with Description
1	<b>public String getMessage()</b> Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	<b>public Throwable getCause()</b> Returns the cause of the exception as represented by a Throwable object.
3	<b>public String toString()</b> Returns the name of the class concatenated with the result of getMessage().
4	<b>public void printStackTrace()</b> Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	<b>public StackTraceElement [] getStackTrace()</b> Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	<b>public Throwable fillInStackTrace()</b> Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

## Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
//Protected code
}catch(ExceptionName e1)
{
//Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

### **Example**

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{
public static void main(String args[]){
try{
int a[] = new int[2];
System.out.println("Access element three : " + a[3]);
}catch(ArrayIndexOutOfBoundsException e){
System.out.println("Exception thrown : " + e);
}
System.out.println("Out of the block");
}
}
```

This will produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

### **Multiple Catch Blocks**

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first

catch block in the list. If the data type of the exception thrown matches `ExceptionType1`, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

### Example

Here is code segment showing how to use multiple try/catch statements.

```
try
{
file = new FileInputStream(fileName);
x = (byte) file.read();
} catch(IOException i)
{
    i.printStackTrace();

return -1;
} catch(FileNotFoundException f) //Not valid!
{
f.printStackTrace();
return -1;
}
```

### Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it:

```
catch (IOException|FileNotFoundException ex) {
logger.log(ex);
throw ex;
```

### The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a `RemoteException`:

```
import java.io.*;
public class className
{
public void deposit(double amount) throws RemoteException
{
// Method implementation
throw new RemoteException();
}
//Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a `RemoteException` and an `InsufficientFundsException`:

```
import java.io.*;
public class className
{
public void withdraw(double amount) throws RemoteException,
InsufficientFundsException
{
// Method implementation
}
//Remainder of class definition
}
```

### **The Finally Block**

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}finally
{
//The finally block always executes.
}
```

### **Example**

```
public class ExcepTest{
public static void main(String args[]){
int a[] = new int[2];
try{
System.out.println("Access element three : " + a[3]);
}catch(ArrayIndexOutOfBoundsException e){
System.out.println("Exception thrown : " + e);
}
finally{
```

```
a[0] = 6;
System.out.println("First element value: " +a[0]);
System.out.println("The finally statement is executed");
}
}
}
```

This will produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

Note the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

### User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{
}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

#### 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;  
}  
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// 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;
```

```
}
```

```
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
```

```
public class BankDemo
```

```

{
public static void main(String [] args)
{
CheckingAccount c = new CheckingAccount(101);
System.out.println("Depositing $500...");
c.deposit(500.00);
try
{
System.out.println("\nWithdrawing $100...");
c.withdraw(100.00);
System.out.println("\nWithdrawing $600...");
c.withdraw(600.00);
} catch (InsufficientFundsException e)
{
System.out.println("Sorry, but you are short $" + e.getAmount());
e.printStackTrace();
}
}
}
}

```

Compile all the above three files and run BankDemo. This will produce the following result:

Depositing \$500...

Withdrawing \$100...

    Withdrawing \$600...

Sorry, but you are short \$200.0

InsufficientFundsException

at CheckingAccount.withdraw(CheckingAccount.java:25)

at BankDemo.main(BankDemo.java:13)

### Common Exceptions

In Java, it is possible to define two categories of Exceptions and Errors.

□ **JVM Exceptions:** These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.

□ **Programmatic Exceptions:** These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.