JGI JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications (MCA)

OOP's Using JAVA

21MCAC101

Unit – 1 : Classes, Inheritance, package and Interface

Prepared by
Raghavendra R
Assistant professor
School of CS & IT

r.raghavendra@jainuniversity.ac.in +91-86601-61661 raguramesh88@gmail.com
+91-96116-77907

Class

Class is a basis of OOP languages. It is a logical construct which defines shape and nature of an object. Entire Java is built upon classes.

Class Fundamentals

Class can be thought of as a user-defined data type. We can create variables (objects) of that data type. So, we can say that class is a template for an object and an object is an instance of a class. Most of the times, the terms object and instance are used interchangeably.

The General Form of a Class

A class contains data (member or instance variables) and the code (member methods) that operate on the data. The general form can be given as –

```
class classname
{
type var1;
type var2;
…….
type method1(para_list)
{
//body of method1
}
type method2(para_list)
{
//body of method2
}
……..
}
```

Here, classname is any valid name given to the class. Variables declared within a class are called as instance variables because every instance (or object) of a class contains its own copy of these variables. The code is contained within methods. Methods and instance variables collectively called as members of the class.

A Simple Class

Here we will consider a simple example for creation of class, creating objects and using members of the class. One can store the following program in a single file called BoxDemo.java. (Or, two classes can be saved in two different files with the names Box.java and BoxDemo.java.)

```
class Box
{
double w, h, d;
```



```
}
class BoxDemo
{
```

```
public static void main(String args[])
{
Box b1=new Box();
Box b2=new Box();
double vol;
b1.w=2;
b1.h=4;
b1.d=3;
b2.w=5;
b2.h=6;
b2.d=2;
vol=b1.w*b1.h*b1.d;
System.out.println("Volume of Box1 is " + vol);
vol=b2.w*b2.h*b2.d;
System.out.println("Volume of Box2 is " + vol);
}
}
```

The output would be –
Volume of Box1 is 24.0
Volume of Box1 is 60.0

When you compile above program, two class files will be created viz. Box.class and BoxDemo.class.

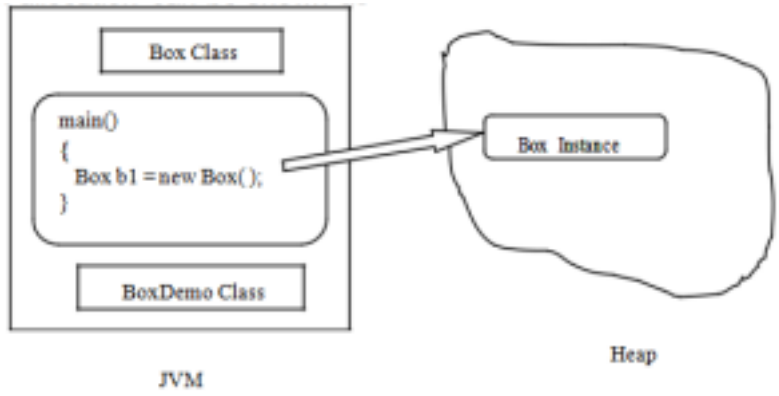
Since main() method is contained in BoxDemo.class, you need to execute the same. In the above example, we have created a class Box which contains 3 instance variables w, h, d. Box b1=new Box();

The above statement creates a physical memory for one object of Box class. Every object is an instance of a class, and so, b1 and b2 will have their own copies of instance variables w, h and d. The memory layout for one object allocation can be shown as –

```
Box Class
main() Box Instance
{
Box b1 = new Box( );
```



```
}
```



Declaring Objects

Creating a class means having a user-defined data type. To have a variable of this new data type, we should create an object. Consider the following declaration:

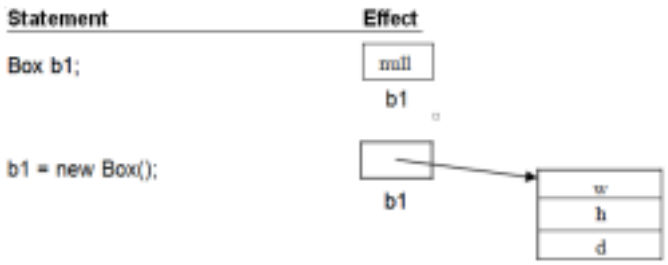
Box b1;

This statement will not actually create any physical object, but the object name b1 can just refer to the actual object on the heap after memory allocation as follows –

b1 = new Box ();

We can even declare an object and allocate memory using a single statement – Box b1=new Box();

Without the usage of new, the object contains null. Once memory is allocated dynamically, the object b1 contains the address of real object created on the heap. The memory map is as shown in the following diagram –



Closer look at new

The general form for object creation is –

obj_name = new class_name();

Here, class_name() is actually a constructor call. A constructor is a special type of member

function invoked automatically when the object gets created. The constructor usually contains the code needed for object initialization. If we do not provide any constructor, then Java supplies a default constructor.

Java treats primitive types like byte, short, int, long, char, float, double and boolean as ordinary variables but not as an object of any class. This is to avoid extra overhead on the heap memory and also to increase the efficiency of the program. Java also provides the class-version of these primitive types that can be used only if necessary. We will study those types later in detail.

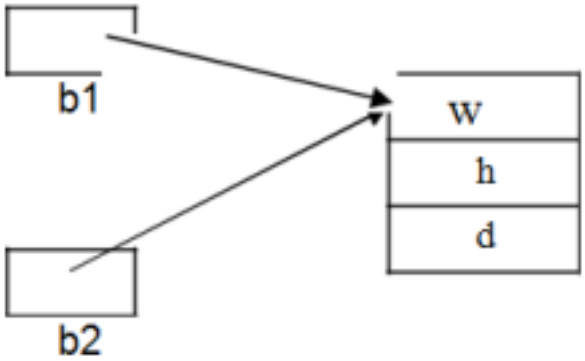
With the term dynamic memory allocation, we can understand that the keyword new allocates memory for the object during runtime. So, depending on the user's requirement memory will be utilized. This will avoid the problems with static memory allocation (either shortage or wastage of memory during runtime). If there is no enough memory in the heap when we use new for memory allocation, it will throw a run-time exception.

Assigning Object Reference Variables

When an object is assigned to another object, no separate memory will be allocated. Instead, the second object refers to the same location as that of first object. Consider the following declaration –

```
Box b1= new Box();
Box b2= b1;
```

Now both b1 and b2 refer to same object on the heap. The memory representation for two objects can be shown as –



Thus, any change made for the instance variables of one object affects the other object also. Although b1 and b2 both refer to the same object, they are not linked in any other way. For

example, a subsequent assignment to b1 will simply unhook b1 from the original object without

affecting the object or affecting
b2. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.
NOTE that when you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

A class can consist of instance variables and methods. We have seen declaration and usage of instance variables in Program 2.1. Now, we will discuss about methods. The general form of a method is –

```
ret_type method_name(para_list)
{
//body of the method
return value;
}
```

Here,
ret_type specifies the data type of the variable returned by the method. It may be any primitive type or any other derived type including name of the same class. If the method does not return any value, the ret_type should be specified as void.
method_name is any valid name given to the method
para_list is the list of parameters (along with their respective types) taken the method. It may be even empty also.
body of method is a code segment written to carryout some process for which the method is meant for.
return is a keyword used to send value to the calling method. This line will be absent if the ret_type is void.

Adding Methods to Box class

Though it is possible to have classes with only instance variables as we did for Box class of Program 2.1, it is advisable to have methods to operate on those data. Because, methods acts as interface to the classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide

access to data, you can also define methods that are used internally by the class itself. Consider the following example –

```
class Box
{
double w, h, d;
void volume()
{
System.out.println("The volume is " + w*h*d);
}
}
class BoxDemo
{
public static void main(String args[])
{
Box b1=new Box();
Box b2=new Box();
b1.w=2;
b1.h=4;
b1.d=3;
b2.w=5;
b2.h=6;
b2.d=2;
b1.volume();
b2.volume();
}
}
```

The output would be –

```
The volume is 24.0
The volume is 60.0
```

In the above program, the Box objects b1 and b2 are invoking the member method volume() of the Box class to display the volume. To attach an object name and a method name, we use dot (.) operator. Once the program control enters the method volume(), we need not refer to object name to use the instance variables w, h and d.

Returning a value

In the previous example, we have seen a method which does not return anything. Now we will modify the above program so as to return the value of volume to main() method.

```
class Box
{
double w, h, d;
double volume()
{
return w*h*d;
}
}
class BoxDemo
{
public static void main(String args[])
{
Box b1=new Box();
Box b2=new Box();
double vol;
b1.w=2;
b1.h=4;
b1.d=3;
b2.w=5;
b2.h=6;
b2.d=2;
vol = b1.volume();
System.out.println("The volume is " + vol);
System.out.println("The volume is " + b2.volume());
}
}
```

The output would be –
The volume is 24.0
The volume is 60.0

As one can observe from above example, we need to use a variable at the left-hand side of the assignment operator to receive the value returned by a method. On the other hand, we can directly make a method call within print statement as shown in the last line of above program. There are two important things to understand about returning values:

The type of data returned by a method must be compatible with the return type specified by the

method. For example, if the return type of some method is boolean, you could not return an integer.

The variable receiving the value returned by a method (such as vol, in this case) must also be compatible with the return type specified for the method.

Adding Methods that takes Parameters

Having parameters for methods is for providing some input information to process the task. Consider the following version of Box class which has a method with parameters.

```
class Box
{
double w, h, d;
double volume()
{
return w*h*d;
}
void set(double wd, double ht, double dp)
{
w=wd;
h=ht;
d=dp;
}
}
class BoxDemo
{
public static void main(String args[])
{
Box b1=new Box();
Box b2=new Box();
b1.set(2,4,3);
b2.set(5,6,2);
System.out.println("The volume of b1 is " + b1.volume());
System.out.println("The volume of b2 is " + b2.volume());
}
}
```

The output would be –
The volume of b1 is 24.0

The volume of b2 is 60.0

In the above program, the Box class contains a method set() which take 3 parameters. Note that, the variables wd, ht and dp are termed as formal parameters or just parameters for a method. The

values passed like 2, 4, 3 etc. are called as actual arguments or just arguments passed to the method.

Constructors

Constructor is a special type of member method which is invoked automatically when the object gets created. Constructors are used for object initialization. They have same name as that of the class. Since they are called automatically, there is no return type for them. Constructors may or may not take parameters.

```
class Box
{
double w, h, d;
double volume()
{
return w*h*d;
}
Box() //ordinary constructor
{ w=h=d=5;
}
Box(double wd, double ht, double dp) //parameterized constructor
{
w=wd;
h=ht;
d=dp;
}
}
class BoxDemo
{
public static void main(String args[])
{
Box b1=new Box();
Box b2=new Box();
Box b3=new Box(2,4,3);
System.out.println("The volumeof b1 is " + b1.volume());
```

```
System.out.println("The volumeof b2 is " + b2.volume());
System.out.println("The volumeof b3 is " + b3.volume());
}
}
```

The output would be –


```
The volume of b1 is 125.0
The volume of b2 is 125.0
The volume of b3 is 24.0
```

When we create two objects b1 and b2, the constructor with no arguments will be called and the all the instance variables w, h and d are set to 5. Hence volume of b1 and b2 will be same (that is 125 in this example). But, when we create the object b3, the parameterized constructor will be called and hence volume will be 24.
Few points about constructors:

Every class is provided with a default constructor which initializes all the data members to respective default values. (Default for numeric types is zero, for character and strings it is null and default value for Boolean type is false.)
In the statement
classname ob= new classname();
the term classname() is actually a constructor call.

If the programmer does not provide any constructor of his own, then the above statement will call default constructor.

If the programmer defines any constructor, then default constructor of Java can not be used.

So, if the programmer defines any parameterized constructor and later would like to create an object without explicit initialization, he has to provide the default constructor by his own.

For example, the above program, if we remove ordinary constructor, the statements like Box b1=new Box();
will generate error. To avoid the error, we should write a default constructor like – Box(){ }
Now, all the data members will be set to their respective default values.

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is

always a reference to the object which invokes the method call. For example, in the Program 2.5, the method volume() can be written as –

```
double volume()
{
return this.w * this.h * this.d;
}
```


Here, usage of this is not mandatory as it is implicit. But, in some of the situations, it is useful as explained in the next section.

Instance Variable Hiding

As we know, in Java, we can not have two local variables with the same name inside the same or enclosing scopes. But we can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. That is, if we write following code snippet for a constructor in Program 2.5, we will not get an expected output –

```
Box(double w, double h, double d)
{
w=w;
h=h;
d=d;
}
```

Here note that, formal parameter names and data member names match exactly. To avoid the problem, we can use –

```
Box(double w, double h, double d)
{
this.w=w; //this.w refers to data member name and w refers to formal parameter
this.h=h;
this.d=d;
}
```

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such

situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the finalize() method. The Java run time calls that method whenever it is about to recycle an object of that class.

The finalize( ) method has this general form:

```
protected void finalize( )
{
```



```
// finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. Note that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope. So, we can not know when finalize() method is called, or we may be sure whether it is called or not before our program termination. Therefore, if at all our program uses some resources, we should provide some other means for releasing them and must not depend on finalize() method.

Overloading Constructors

One can have more than one constructor for a single class if the number and/or type of arguments are different. Consider the following code:

```
class OverloadConstruct
{
int a, b;
OverloadConstruct()
{
System.out.println("Constructor without arguments");
}
OverloadConstruct(int x)
{
a=x;
System.out.println("Constructor with one argument:"+a);
}
OverloadConstruct(int x, int y)
{
a=x;
b=y;
System.out.println("Constructor with two arguments:"+ a +"\t"+ b);
```

```
}
}
class OverloadConstructDemo
{
public static void main(String args[])
{
OverloadConstruct ob1= new OverloadConstruct();
OverloadConstruct ob2= new OverloadConstruct(10);
OverloadConstruct ob3= new OverloadConstruct(5,12);
```



```
}
}
```

Output:
Constructor without arguments
Constructor with one argument: 10 12
Constructor with two arguments: 5

Using Objects as Parameters

Just similar to primitive types, even object of a class can also be passed as a parameter to any method. Consider the example given below –

```
class Test
{
int a, b;
Test(int i, int j)
{
a = i;
b = j;
}
boolean equals(Test ob)
{
if(ob.a == this.a && ob.b == this.b)
return true;
else
return false;
}
}
class PassOb
```

```
{
public static void main(String args[])
{
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```


Output:
ob1 == ob2: true
ob1 == ob3: false

Using one object to initialize the other:
Sometimes, we may need to have a replica of one object. The usage of following statements will not serve the purpose.

```
Box b1=new Box(2,3,4);
Box b2=b1;
```

In the above case, both b1 and b2 will be referring to same object, but not two different objects. So, we can write a constructor having a parameter of same class type to clone an object.

```
class Box
{
double h, w, d;
Box(double ht, double wd, double dp)
{
h=ht; w=wd; d=dp;
}

Box (Box bx) //observe this constructor
{
h=bx.h; w=bx.w; d=bx.d;
}
void vol()
{
```

```
System.out.println("Volume is " + h*w*d);
}
public static void main(String args[])
{
Box b1=new Box(2,3,4);
Box b2=new Box(b1); //initialize b2 using b1
b1.vol();
b2.vol();
}
}
```

Output:
Volume is 24

Volume is 24

A Closer Look at Argument Passing

In Java, there are two ways of passing arguments to a method.
Call by value : This approach copies the value of an argument into the formal parameter of the method. Therefore, changes made to the parameter of the method have no effect on the argument.

Call by reference: In this approach, a reference to an argument is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

In Java, when you pass a primitive type to a method, it is passed by value. When you pass an object to a method, they are passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

```
class Test
{
int a, b;
Test(int i, int j)
{
```

```
a = i; b = j;
}
void meth(Test o)
{
o.a *= 2;
o.b /= 2;
}
}
class CallByRef
{
public static void main(String args[])
{
Test ob = new Test(15, 20);
System.out.println("before call: " + ob.a + " " + ob.b);
ob.meth(ob);
```



```
System.out.println("after call: " + ob.a + " " +
ob.b); }
}
```

Output:
before call: 15 20
after call: 30 10

Returning Objects

In Java, a method can return an object of user defined class. class Test

```
{
int a;
Test(int i)
{
a = i;
}
Test incrByTen()
{
Test temp = new Test(a+10);
return temp;
}
}
```

```
class RetOb
{
public static void main(String args[])
{
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: " +
 ob2.a); }
}
```

Output:
ob1.a: 2
ob2.a: 12

ob2.a after second increase: 22

Introducing Access Control

Encapsulation feature of Java provides a safety measure viz. access control. Using access specifiers, we can restrict the member variables of a class from outside manipulation. Java provides following access specifiers:

public
private
protected

Along with above access specifiers, Java defines a default access level.

Some aspects of access control are related to inheritance and package (a collection of related classes). The protected specifier is applied only when inheritance is involved. So, we will now discuss about only private and public.

When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its

package. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class. An access specifier precedes the rest of a member's type specification. For example,

public int x;
private char ch;
Consider a program given below –

```
class Test
{
int a;
public int b;
private int c;
void setc(int i)
{
c = i;
}
int getc()
{
```



```
return c;
}
}
class AccessTest
{
public static void main(String args[])
{
Test ob = new Test();
ob.a = 10;
ob.b = 20;
// ob.c = 100; // inclusion of this line is Error!
ob.setc(100);
System.out.println("a, b, and c: " + ob.a + " " + ob.b + " "+ob.getc());
}
}
```

Understanding static

When a member is declared static, it can be accessed before any objects of its class are created,

and without reference to any object. Instance variables declared as static are global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:

• They can only call other static methods. They must only access static data. •

They cannot refer to this or super in any way.

• If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. class UseStatic

```
{
static int a = 3;
static int b;
static void meth(int x) //static method
{
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static //static block
{
```



```
System.out.println("Static block initialized."); b = a * 4;
}
public static void main(String args[])
{
meth(42);
}
}
```

Output:

Static block initialized.

x = 42

a = 3

b = 12

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.

The general form is –

classname.method();

Consider the following program:

```
class StaticDemo
{
static int a = 42;
static int b = 99;
static void callme()
{
System.out.println("Inside static method, a = " + a);
}
}
class StaticByName
{
public static void main(String args[])
{
StaticDemo.callme();
System.out.println("Inside main, b = " + StaticDemo.b);
}
}
```

Output:

Inside static method, a = 42

Inside main, b = 99

Inheritance

Inheritance is one of the building blocks of object oriented programming languages. It allows creation of classes with hierarchical relationship among them. Using inheritance, one can create a general class that defines traits common to a set of related items. This 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 add its own, unique elements. Through inheritance, one can achieve re-usability of the code.

In Java, inheritance is achieved using the keyword extends. The syntax is given below: class A

```
{
```

```
 //super class
 //members of class A
}
 class B extends A {
 //sub class
 //members of B
}
```
Consider a program to understand the concept:

```
class A
{
int i, j;
void showij()
{
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A
{
int k;
void showk()
{
System.out.println("k: " + k);
}
void sum()
```



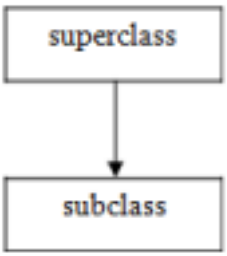
```
{
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance
{
public static void main(String args[])
{
A superOb = new A();
B subOb = new B();
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
```

```
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println("Sum of i, j and k in subOb:"); subOb.sum();
}
}
```

Note that, private members of the super class can not be accessed by the sub class. The subclass contains all non-private members of the super class and also it contains its own set of members to achieve specialization.

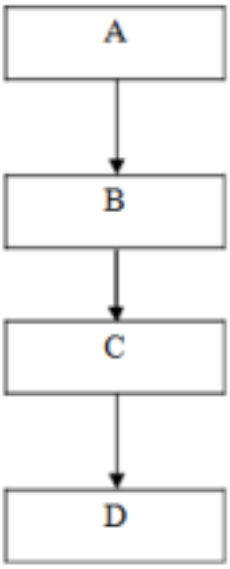
Type of Inheritance

Single Inheritance: If a class is inherited from one parent class, then it is known as single inheritance. This will be of the form as shown below –



The previous program is an example of single inheritance.

Multilevel Inheritance: If several classes are inherited one after the other in a hierarchical manner, it is known as multilevel inheritance, as shown below –



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. Consider the following for illustration:

```
class Base
{
void dispB()
{
System.out.println("Super class " );
}
}
class Derived extends Base
{
void dispD()
{
System.out.println("Sub class ");
}
}
class Demo
{
public static void main(String args[])
{
Base b = new Base();
Derived d=new Derived();
```



```
b=d; //superclass reference is holding subclass object
```

```
b.dispB();
//b.dispD(); error!!
 }
 }
```

Note that, the type of reference variable decides the members that can be accessed, but not the type of the actual object. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

Using super

In Java, the keyword super can be used in following situations:

- To invoke superclass constructor within the subclass constructor
- To access superclass member (variable or method) when there is a duplicate member name in the subclass
- Let us discuss each of these situations:
- To invoke superclass constructor within the subclass constructor: Sometimes, we may need to initialize the members of super class while creating subclass object. Writing such a code in subclass constructor may lead to redundancy in code.

For example,

```
class Box
{
double w, h, b;
Box(double wd, double ht, double br)
 {
w=wd; h=ht; b=br;
 }
 }
class ColourBox extends Box
 {
int colour;
ColourBox(double wd, double ht, double br, int c)
 {
  w=wd; h=ht; b=br;
colour=c;
  //code redundancy
   }
 }
```

Also, if the data members of super class are private, then we can't even write such a code in subclass constructor. If we use super() to call superclass constructor, then it must be the first statement executed inside a subclass constructor as shown below –

```
class Box
{
double w, h, b;
Box(double wd, double ht, double br)
{
w=wd; h=ht; b=br;
}
}
class ColourBox extends Box
{
int colour;
ColourBox(double wd, double ht, double br, int c)
{
super(wd, ht, br); //calls superclass constructor
colour=c;
}
}
class Demo
{
public static void main(String args[])
{
ColourBox b=new ColourBox(2,3,4, 5);
}
}
```

Here, we are creating the object b of the subclass ColourBox . So, the constructor of this class is invoked. As the first statement within it is super(wd, ht, br), the constructor of superclass Box is invoked, and then the rest of the statements in subclass constructor ColourBox are executed. To access superclass member variable when there is a duplicate variable name in the subclass: This form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
class A
{
int a;
}
```

```
class B extends A
{
int a; //duplicate variable a
B(int x, int y)
{ //accessing superclass a
super.a=x;
a=y; //accessing own member a
}
void disp()
{
System.out.println("super class a: "+ super.a);
System.out.println("sub class a: "+ a);
}
}
class SuperDemo
{
public static void main(String args[])
{
B ob=new B(2,3);
ob.disp();
}
}
```

Creating Multilevel Hierarchy

Java supports multi-level inheritance. A sub class can access all the non-private members of all of its super classes. Consider an illustration:

```
class A
{
int a;
}
class B extends A
{
int b;
}
class C extends B
{
int c;
C(int x, int y, int z)
```

```
{
a=x; b=y; c=z;
}
void disp()
{
System.out.println("a= "+a+ " b= "+b+" c="+c);
}
}
class MultiLevel
{
public static void main(String args[])
{
C ob=new C(2,3,4);
ob.disp();
}
}
```

When Constructors are called

When class hierarchy is created (multilevel inheritance), the constructors are called in the order of their derivation. That is, the top most super class constructor is called first, and then its immediate sub class and so on. If super is not used in the sub class constructors, then the default constructor of super class will be called.

```
class A
{
A()
{
System.out.println("A's constructor.");
}
}
class B extends A
{
B()
{
System.out.println("B's constructor.");
}
}
```

```
class C extends B
{
C()
{
```



```
System.out.println("C's constructor.");
}
}
class CallingCons
{
public static void main(String args[])
{
C c = new C();
}
}
```

Output:
A's constructor
B's constructor
C's constructor

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.

```
class A
{
int i, j;
A(int a, int b)
{
i = a;
j = b;
} //suppressed
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;
} //Overridden method
void show()
{
System.out.println("k: " + k);
}
}
class Override
{
public static void main(String args[])
{
B subOb = new B(1, 2, 3);
subOb.show();
}
}
```

Output:
k: 3

Note that, above program, only subclass method show() got called and hence only k got displayed. That is, the show() method of super class is suppressed. If we want superclass method also to be called, we can re-write the show() method in subclass as –

```
void show()
{
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
```

Method overriding occurs only when the names and the type signatures of the two methods (one

in superclass and the other in subclass) are identical. If two methods (one in superclass and the other in subclass) have same name, but different signature, then the two methods are simply overloaded.

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is

resolved at run time, rather than compile time. Java implements run-time polymorphism using dynamic method dispatch. We know that, a superclass reference variable can refer to subclass object. Using this fact, Java resolves the calls to overridden methods during runtime. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
class A
{
void callme()
{
System.out.println("Inside A");
}
}
class B extends A
{
void callme()
{
System.out.println("Inside B");
}
}
class C extends A
{
void callme()
{
System.out.println("Inside C");
```

```
}
}
class Dispatch
{
public static void main(String args[])
{
A a = new A();
B b = new B();
C c = new C();
```



```
A r; //Superclass reference
r = a; //holding subclass object
r.callme();
r = b;
r.callme();
r = c;
r.callme();
}
}
```

Why overridden methods?

Overridden methods are the way that Java implements the "one interface, multiple methods" aspect of polymorphism. superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, 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, yet still enforces a consistent interface. Thus, 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 objectoriented design brings to bear on code reuse and robustness.

Using Abstract Classes

Sometimes, the method definition will not be having any meaning in superclass. Only the subclass (specialization) may give proper meaning for such methods.In such a situation, having a definition for a method in superclass is absurd. Also, we should enforce the subclass to override such a method. A method which does not contain any definition in the superclass is termed as abstract method. Such a method declaration should be preceded by the keyword abstract. These methods are sometimes referred to as subclasser responsibility because they have no

implementation specified in the superclass.

A class containing at least one abstract method is called as abstract class. Abstract classes can not be instantiated, that is one cannot create an object of abstract class. Whereas, a reference can be created for an abstract class.

```
abstract class A
{
abstract void callme();
void callmetoo()
{
```



```
System.out.println("This is a concrete method.");
}
}
class B extends A
{
void callme() //overriding abstract method
{
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo
{
public static void main(String args[])
{
B b = new B(); //subclass object
b.callme(); //calling abstract method
b.callmetoo(); //calling concrete method
}
}
```

Example: Write an abstract class shape, which has an abstract method area(). Derive three classes Triangle, Rectangle and Circle from the shape class and to override area(). Implement run-time polymorphism by creating array of references to supeclass. Compute area of different shapes and display the same.

Solution:

```
abstract class Shape
```

```
{
final double PI= 3.1416;
abstract double area();
}
class Triangle extends Shape
{
int b, h;
Triangle(int x, int y) //constructor
{
b=x;
h=y;
}
double area()
```



```
{
 //method overriding
 System.out.print("\nArea of Triangle is:");
return 0.5*b*h;
}
}
class Circle extends Shape
{
int r;
Circle(int rad) //constructor
{
r=rad;
}
double area() //overriding
{
System.out.print("\nArea of Circle is:");
return PI*r*r;
}
}
class Rectangle extends Shape
{
int a, b;
Rectangle(int x, int y) //constructor
{
a=x;
```

```
b=y;
}
double area() //overriding
{
System.out.print("\nArea of Rectangle is:");
return a*b;
}
}
class AbstractDemo
{
public static void main(String args[])
{
Shape r[]={new Triangle(3,4), new Rectangle(5,6),new
Circle(2)}; for(int i=0;i<3;i++)
System.out.println(r[i].area());
```



```
}
}
```

Output:
Area of Triangle is:6.0
Area of Rectangle is:30.0
Area of Circle is:12.5664

Note that, here we have created array r, which is reference to Shape class. But, every element in r is holding objects of different subclasses. That is, r[0] holds Triangle class object, r[1] holds Rectangle class object and so on. With the help of array initialization, we are achieving this, and also, we are calling respective constructors. Later, we use a for-loop to invoke the method area() defined in each of these classes.

Using final

The keyword final can be used in three situations in Java:

• To create the equivalent of a named constant. To prevent method overriding •

To prevent Inheritance

• To create the equivalent of a named constant: A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared.

For example:

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

It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.

To prevent method overriding: Sometimes, we do not want a superclass method to be overridden in the subclass. Instead, the same superclass method definition has to be used by every subclass. In such situation, we can prefix a method with the keyword final as shown below –

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

To prevent Inheritance: As we have discussed earlier, the subclass is treated as a specialized class and superclass is most generalized class. During multi-level inheritance, the bottom most class will be with all the features of real-time and hence it should not be inherited further. In such situations, we can prevent a particular class from inheriting further, using the keyword final. For example –

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

```
class B extends A // ERROR! Can't subclass A
{
// ...
}
```

Note:
Declaring a class as final implicitly declares all of its methods as final, too. 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

The Object Class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array. Object defines the following methods, which means that they are available in every object.

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

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You may override the others. The equals( ) method compares the contents of two objects. It returns true if the objects are equivalent, and false otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The toString( ) method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using println( ). Many classes override this method.

Packages

When we have more than one class in our program, usually we give unique names to classes. In a real-time development, as the number of classes increases, giving unique meaningful name for each class will be a problem. To avoid name-collision in such situations, Java provides a concept of packages. A package is a collection of classes. 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, include a package command as the first statement in a Java source file. Any class 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.

General form of the package statement:
package pkg;
Example – package MyPackage;

Java uses file system directories to store packages. For example, the .class file for any class 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 class in other files from being part of that same package.

One 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. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

As we have seen, packages are reflected with directories. This will raise the question that - how does Java run-time know where to look for the packages that we create?

By default, Java run-time uses current working directory as a starting point. So, if our package is in a sub-directory of current working directory, then it will be found.

We can set directory path using CLASSPATH environment variable.

We can use –classpath option with javac and java to specify path of our classes.

Assume that we have created a package MyPackage. When the second two options are used, the class path must not include MyPackage. It must simply specify the path to MyPackage. For example, in a Windows environment, if the path to MyPackage is

C:\MyPrograms\Java\MyPackage
Then the class path to MyPackage is
C:\MyPrograms\Java
Consider the program given below –

```
package MyPackage;
class Test
```



```
{
int a, b;
Test(int x, int y)
{
a=x; b=y;
}
void disp()
{
System.out.println("a= "+a+" b= "+b);
}
}
class PackDemo
{
public static void main(String args[])
{
Test t=new Test(2,3);
t.disp();
}
}
```

Access Protection

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. 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.

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

Even a class has accessibility feature. A class can be kept as default or can be declared as public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class

Accessibility of members of the class can be better understood using the following table.

Private No Modifier Protected Public

Same class Yes Yes Yes Yes Same package Subclass No Yes Yes Yes Same package non-subclass No Yes Yes Yes Different package Subclass No No Yes Yes Different package non-subclass No No No Yes

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.

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

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

For example,
import java.util.Date;
import java.io.*;

The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in java.lang, it is implicitly imported by the compiler for all programs.

This is equivalent to the following line being at the top of all of your
programs: import java.lang.*;

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

The import statement is optional. Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy. For example,

import java.util.*;
class MyDate extends Date
{ ……}
Can be written as –
class MyDate extends java.util.Date

{ …}

Interfaces

Interface is an abstract type that can contain only the declarations of methods and constants. Interfaces are syntactically similar to classes, but they do not contain instance variables, and their methods are declared without any body. Any number of classes can implement an interface. One class may implement many interfaces. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are

alternative means for multiple inheritance in Java.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface: access interface name
{
type final-varname1 = value;
type final-varname2 = value;
……
return-type method-name1(parameter-list); return-type method-name2(parameter-list);
……
}

Few key-points about interface:
When no access specifier is mentioned for an interface, then it is treated as default and the interface is only available to other members of the package in which it is declared. When an interface is declared as public, the interface can be used by any other code.

All the methods declared are abstract methods and hence are not defined inside interface. But, a class implementing an interface should define all the methods declared inside the interface.

Variables declared inside of interface are implicitly final and static, meaning they cannot be changed by the implementing class.

All the variables declared inside the interface must be initialized. All methods and variables are implicitly public.

Implementing 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 interface1, interface2... {
// class-body
}
Consider the following example:
interface ICallback

```
{
void callback(int param);
}
class Client implements ICallback
{ //note public
public void callback(int p)
{
System.out.println("callback called with " + p);
}
void test()
{
System.out.println(“ordinary method”);
}
}
class TestIface
{
public static void main(String args[])
{
ICallback c = new Client();
c.callback(42);
// c.test() //error!!
}
}
```


Here, the interface ICallback contains declaration of one method callback(). The class Client implementing this interface is defining the method declared in interface. Note that, the method callback() is public by default inside the interface. But, the keyword public must be used while defining it inside the class. Also, the class has its own method test(). In the main() method, we are creating a reference of interface pointing to object of Client class. Through this reference, we can call interface method, but not method of the class.

The true polymorphic nature of interfaces can be found from the following example –

```
interface ICallback
{
void callback(int param);
}
class Client implements ICallback
{
public void callback(int p) //note public
```

```
{
System.out.println("callback called with " + p);
}
}
class Client2 implements ICallback
{
public void callback(int p)
{
System.out.println("Another version of ICallBack"); System.out.println("p squared " +
p*p); }
}
class TestIface
{
public static void main(String args[])
{
ICallback x[]={new Client(), new Client2()};
for(int i=0;i<2;i++)
x[i].callback(5);
}
}
```

Output:
callback called with 5
Another version of ICallBack
p squared 25

In this program, we have created array of references to interface, but they are initialized to class objects.
Using the array index, we call respective implementation of callback() method. Note: Interfaces may look similar to abstract classes. But, there are lot of differences between them as shown in the following table:

Abstract Class	Interface
Can have instance methods that implements a default behavior	Are implicitly abstract and cannot have implementations.
May contain non-final variables.	Variables declared in interface are by default final.
Can have the members with private,	Members of a Java interface are public by

protected, etc..	default.
A Java abstract class should be extended using keyword “extends”.	Java interface should be implemented using keyword “implements”
An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
Not slow	Compared to abstract classes, interfaces are slow as it requires extra indirection

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class all of those variable names will be in scope as constants (Similar to #define in C/C++). If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is just using a set of constants. Consider an example to illustrate the same:

```
interface SharedConst
{
 int FAIL=0;
int PASS=1;
 //these are final by default
  }
class Result implements SharedConst
{
double mr;
Result(double m)
{
mr=m;
```



```
}
int res()
{
if(mr<40)
return FAIL;
else return PASS;
```

```
}
}
class Exam extends Result implements SharedConst
{
Exam(double m)
{
super(m);
}
public static void main(String args[])
{
Exam r = new Exam(56);
switch(r.res())
{
case FAIL:
System.out.println("Fail");
break;
case PASS:
System.out.println("Pass");
break;
}
}
}
```

Interfaces can be extended

One interface can inherit another interface by using the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
interface A
{
void meth1();
void meth2();
}
```



```
interface B extends A
{
void meth3();
}
```

```
class MyClass implements B
{
public void meth1()
{
System.out.println("Implement meth1().");
}
public void meth2()
{
System.out.println("Implement meth2().");
}
public void meth3()
{
System.out.println("Implement meth3().");
}
}
class IFExtend
{
public static void main(String arg[])
{
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Exception Handling

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

Exception Handling Fundamentals

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.

Java exception handling is managed using five keywords:

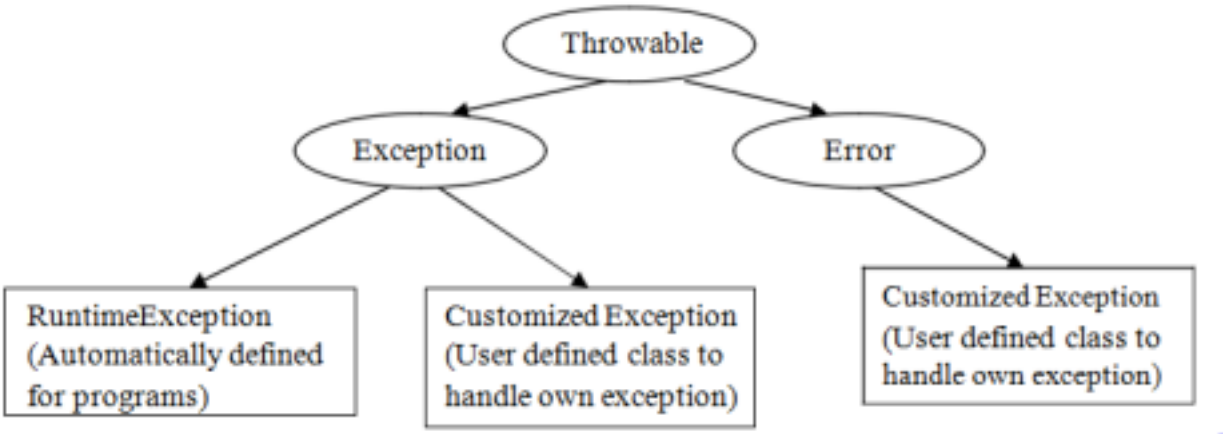
- try: A suspected code segment is kept inside try block.
- catch: The remedy is written within catch block.
- throw: Whenever run-time error occurs, the code must throw an exception. • throws: If a method cannot handle any exception by its own and some subsequent methods needs to handle them, then a method can be specified with throws keyword with its declaration.
- finally: block should contain the code to be executed after finishing try-block.

The general form of exception handling is –

```
try
{
// block of code to monitor errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
...
….
finally
{
// block of code to be executed after try block ends
}
```


Exception Types

All the exceptions are the derived classes of built-in class viz. Throwable. It has two subclasses viz.
Exception and Error.



Exception class is used for exceptional conditions that user programs should catch. We can inherit from this class to create our own custom exception types. There is an important subclass of Exception, called RuntimeException. 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.
Error class defines exceptions that are not expected to be caught under normal circumstances by our 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 Exceptions

Let us see, what happens if we do not handle exceptions.

```
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. Since, in the above program, we have not supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any un-caught exception is handled by 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 exception generated when above example is executed:
java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:6)

The stack trace displays class name, method name, file name and line number causing the exception. Also, the type of exception thrown viz. ArithmeticException which is the subclass of Exception is displayed. The type of exception gives more information about what type of error has occurred. The stack trace will always show the sequence of method invocations that led up to the error.

```
class Exc1
{
static void subroutine()
{
int d = 0;
int a = 10 / d;
}
public static void main(String args[])
{
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:6)
at Exc1.main(Exc1.java:10)

Using try and catch

Handling the exception by our own is very much essential as We can display appropriate error message instead of allowing Java run-time to display stack-trace.
It prevents the program from automatic (or abnormal) termination.
To handle run-time error, we need to enclose the suspected code within try block.

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

Output:
Division by zero.
After catch statement.

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

```
import java.util.Random;
class HandleError
{
public static void main(String args[])
{
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<10; i++)
{
try
{
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e)
```

```
{
System.out.println("Division by zero."); a = 0;
}
System.out.println("a: " + a);
}
}
}
```

The output of above program is not predictable exactly, as we are generating random numbers. But, the loop will execute 10 times. In each iteration, two random numbers (b and c) will be generated. When their division results in zero, then exception will be caught. Even after exception, loop will continue to execute.

Displaying a Description of an Exception: We can display this description in a println() statement by simply passing the exception as an argument. This is possible because Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e)
{
System.out.println("Exception: " + e);
a = 0;
}
```

Now, whenever exception occurs, the output will be –
Exception: java.lang.ArithmeticException: / by zero

Multiple Catch Claues

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.

```
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.");
}
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
```

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42 After try/catch blocks.

While using multiple catch blocks, we should give the exception types in a hierarchy of subclass to superclass. Because, catch statement that uses a superclass will catch all exceptions of its own type plus all that of its subclasses. Hence, the subclass exception given after superclass exception is never caught and is a unreachable code, that is an error in Java.

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

```
try
{
int a = 0;
```



```
int b = 42 / a;
}
catch(Exception e)
{
System.out.println("Generic Exception catch.");
}
catch(ArithmeticException e) // ERROR - unreachable
{
System.out.println("This is never reached.");
}
}
}
```

The above program generates error "Unreachable Code", because ArithmeticException is a subclass of Exception.

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 the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
class NestTry
{
public static void main(String args[])
{
try
{
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
try
```

```
{
if(a==1)
a = a/(a-a);
if(a==2)
{
```



```
int c[] = { 1 };
c[10] = 99;
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-bounds: " + e);
}
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
}
}
```

When a method is enclosed within a try block, and a method itself contains a try block, it is considered to be a nested try block.

```
class MethNestTry
{
static void nesttry(int a)
{
try
{
if(a==1)
a = a/(a-a);
 if(a==2)
{
int c[] = { 1 }; c[42] = 99;
}
}
catch(ArrayIndexOutOfBoundsException e)
```

```
{
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[])
{
try
```



```
{
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
}
}
```

throw

Till now, we have seen catching the exceptions that are thrown by the Java run-time system. 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. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways you can obtain a Throwable object:
– using a parameter in a catch clause, or
– creating one with the new operator.

```
class ThrowDemo
{
static void demoproc()
```

```
{
try
{
throw new NullPointerException("demo");
}
catch(NullPointerException e)
{
System.out.println("Caught inside demoproc: " + e);
}
```



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

Here, new is used to construct an instance of NullPointerException. Many of Java's built-in run time exceptions have at least two constructors:
– one with no parameter and
– one that takes a string parameter

When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print() or println(). It can also be obtained by a call to getMessage(), which is defined by Throwable.

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. You do this 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. If they are not, a compile-time error will result.

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.

```
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);
}
}
}
```

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Sometimes it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address such situations.

The finally clause creates a block of code that will be executed after a try/catch block has completed and before the next code of 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. The finally clause is optional. However,

each try statement requires at least one catch or a finally clause.

```
class FinallyDemo
{
static void procA()
{
try
{
System.out.println("inside procA");
throw new RuntimeException("demo");
}
 finally
{
```



```
System.out.println("procA's
 finally"); }
}
static void procB()
{
try
{
System.out.println("inside
procB"); return;
}
finally
{
System.out.println("procB's
 finally"); }
}
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();
}
}
```


Output:
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

Java's Built-in Exceptions

Inside the standard package java.lang, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type RuntimeException. These exceptions need not be included in any method's throws list. Such exceptions are called as unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. Java.lang defines few checked exceptions which needs to be listed out by a method using throws list if that method generate one of these exceptions and does not handle it itself. Java defines several other types of exceptions that relate to its various class libraries.
Table: Java's Unchecked Exceptions

Exception	Meaning
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.
EnumConstantNotPresentException An attempt is made to use an undefined enumeration value.
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 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.
StringIndexOutOfBounds Attempt to index outside the bounds of a string.
TypeNotPresentException Type not found.
UnsupportedOperationException An unsupported operation was encountered.

Table: Java's Checked Exceptions
Exception Meaning
ClassNotFoundException Class not found.
CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException Access to a class is denied.
InstantiationException Attempt to create an object of an abstract class or interface.
InterruptedException One thread has been interrupted by another thread.
NoSuchFieldException A requested field does not exist.
NoSuchMethodException A requested method does not exist.

Creating your own Exception Subclasses

Although Java's built-in exceptions handle most common errors, sometimes we may want to create our own exception types to handle situations specific to our applications. This is achieved by defining a subclass of Exception class. Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. 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.

Methods are
Throwable fillInStackTrace( ), Throwable getCause( ), String getLocalizedMessage( ), String

getMessage(), StackTraceElement[] getStackTrace(), Throwable initCause(Throwable causeExc), void printStackTrace( )
We may wish to override one or more of these methods in exception classes that we create. Two of the constructors of Exception are:
Exception( )
Exception(String msg)

Though specifying a description when an exception is created is often useful, sometimes it is better to override toString( ). The version of toString( ) defined by Throwable (and inherited by Exception) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding toString( ), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

```
class MyException extends Exception
{
int marks;
MyException (int m)
```



```
{
marks=m;
}
public String toString()
{
return "MyException: Marks cannot be Negative";
}
}
class CustExceptionDemo
{
static void test(int m) throws MyException
{
System.out.println("Called test(): "+m);
if(m<0)
throw new MyException(m);
System.out.println("Normal exit");
}
 public static void main(String args[])
{
try{
test(45);
```

```
test(-2);
}
catch (MyException e)
{
System.out.println("Caught " + e);
}}
}
```

Chained Exceptions

The concept of chained exception allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:
– Throwable(Throwable causeExc)
– Throwable(String msg, Throwable causeExc)

In the first form, causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the Error, Exception, and RuntimeException classes.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design. Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic. Unlike some

other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

Note that Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.