Inheritance

•Inheritance allows re-usability of the code.

•In Java, we use the terminology as ***super class*** and ***sub class.***

•Inheritance is achieved using the keyword ***extends.***

•Java **does not support multiple inheritance.**

```
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();
        }
        }
```

- 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.
- **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.

```
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;
            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***

- 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.

```
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;               //code redundancy
                colour=c;
        }
}
```

- Also, if the data members of super class are private, then we can’t even write such a code in subclass constructor.
- To avoid such problems, Java provides a keyword called ***super.***
- 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.
- If we use **super( )** to call superclass constructor, then it must be the first statement executed inside a subclass constructor.

```
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);
                    colour=c;
            }
    }

    class Demo
    {
            public static void main(String args[])
            {
                    ColourBox b=new ColourBox(2,3,4, 5);
            }
    }
```

- The ***super*** keyword can also be used to access superclass member (variable or method).
- This second 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;//this a hides a in A

    B(int x, int y)
    {
        super.a=x;
        a=y;
    }

    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.

```
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;
     }
     void show()    //suppressed
     {
          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;
     }
     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
```

- In the above example, we can see that ***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;
          r = a;
          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.

- Example of Shape class
- Example of Employee class

Abstract Methods and 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.

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

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

Using ***final***

- The keyword **final** has three uses.
  - To create the equivalent of a named constant.
  - To prevent overriding
  - To prevent Inheritance

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

- 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.
- 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

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

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( ) void wait(long milliseconds) void wait(long milliseconds, 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 Package

- To create a include a ***package*** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- If you omit the package statement, the class names are put into the default package, which has no name.
- This is the general form of the **package statement:**

  package *pkg;*

  Example –

  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.
- 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.
- 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* **MyPack**.

- 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

```
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 class

	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

- Example

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.
- This is the general form of the **import** statement:

  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 which can contain the declarations of constants and methods.
- Interfaces are syntactically similar to classes, but they lack 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
{
    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.

- All the methods declared are 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**.

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
}

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

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

- 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);
    }

    void test()
    {
            System.out.println("ordinary method");
    }
}
```

```
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 c = new Client();
            c.callback(42);

            Client2 ob=new Client2();
            c=ob;
            c.callback(5);


            ICallback x[]={new Client(), new Client2()};
            for(int i=0;i<2;i++)
                        x[i].callback(5);
    }
}
```

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 is by default final.
Can have the members with private, protected, etc..	Members of a Java interface are public by 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.

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);
    }
    // ...
}
```

- Now, 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.

- When a nested interface is used outside of its enclosing scope, it must use its fully qualified name.

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

class NestedIFDemo
{
     public static void main(String args[])
     {
             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");
     }
}
```

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 as if that class were importing the constant fields into the class name space as **final** variables.

```
interface SharedConst
{
    int FAIL=0;
    int PASS=1;
}

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 by use of 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 via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Working of these keywords is as explained below –
- A suspected code segment is kept inside ***try*** block.
- Whenever run-time error occurs, the code must ***throw*** an exception.
- The remedy is written within ***catch*** block.
- If a method can not 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 is –

```
try
{
   // block of code to monitor for 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.***
- ***Throwable*** 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);
       }
     }
}
```

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 Clauses

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

```
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.");
          }
          /* This catch is never reached because ArithmeticException is a
          subclass of Exception. */

          catch(ArithmeticException e)   // ERROR - unreachable
          {
                    System.out.println("This is never reached.");
          }
    }
}
```

Nested try Statements

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

```
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[42] = 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.");
                        throw e;   // rethrow the exception
            }
    }
    public static void main(String args[])
    {
            try
            {
                        demoproc();
            } catch(NullPointerException e)
            {
                        System.out.println("Recaught: " + e);
            }
    }
}
```

- 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.

```
// This program contains an error and will not compile.
class ThrowsDemo
{
    static void throwOne()
    {
          System.out.println("Inside throwOne.");
          throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
          throwOne();
    }
}
```

```
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.
- Some times 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.

- **finally** 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.

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.

Java's Checked Exceptions Defined in java.lang

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 inherit those methods provided by **Throwable**.
- Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.

void setStackTrace(StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

• 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
{
    private int detail;
    MyException(int a)
    {
           detail = a;
    }

    public String toString()
    {
           return "MyException[" + detail + "]";
    }
}

class ExceptionDemo
{
    static void compute(int a) throws MyException
    {
           System.out.println("Called compute(" + a + ")");
           if(a > 10)
                   throw new MyException(a);
           System.out.println("Normal exit");
    }
```

```
public static void main(String args[])
{
   try
   {
           compute(1);
           compute(20);
    } catch (MyException e)
     {
       System.out.println("Caught " + e);
      }
   }
}
```

Output:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

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.

- The chained exception methods added to **Throwable** are
  - Throwable getCause( )
  - Throwable initCause(Throwable *causeExc*)
- The **getCause( )** method returns the exception that underlies the current exception.
- If there is no underlying exception, **null** is returned.
- The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception.
- Thus, you can associate a cause with an exception after the exception has been created.

```
class ChainExcDemo
{
    static void demoproc()
    {
        NullPointerException e = new NullPointerException("top layer");
        // add a cause
        e.initCause(new ArithmeticException("cause"));
        throw e;
    }
    public static void main(String args[])
    {
        try {
                demoproc();
            } catch(NullPointerException e)
            {
                // display top level exception
                System.out.println("Caught: " + e);

                // display cause exception
                System.out.println("Original cause: " + e.getCause());
            }
    }
}
```

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

- 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.