

Inheritance

Inheritance is the mechanism of deriving new class from old one, old class is known as superclass and new class is known as subclass. The subclass inherits all of its instances variables and methods defined by the superclass and it also adds its own unique elements. Thus we can say that subclass are specialized version of superclass.

Benefits of Java's Inheritance

1. Reusability of code
2. Code Sharing
3. Consistency in using an interface

Classes

Superclass(Base Class)	Subclass(Child Class)
It is a class from which other classes can be derived.	It is a class that inherits some or all members from superclass.

Types of Inheritance in Java

1. Single Inheritance - one class extends one class only

```
class inherit1
{
    static int i=10;
    inherit1()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    inherit2()
    {
        super();
        System.out.println("Value of i in Child class is"+i);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
    }
}
```

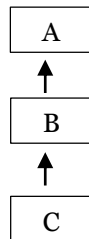
Base Class

Base Class

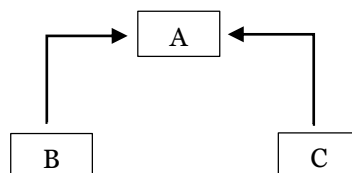
Child Class

Child Class inherited from Base Class. Note: “**extends**” keyword is used to inherit a sub class from superclass.

2. Multilevel Inheritance – It is a ladder or hierarchy of single level inheritance. It means if **Class A is extended by Class B and then further Class C extends Class B** then the whole structure is termed as Multilevel Inheritance. Multiple classes are involved in inheritance, but one class extends only one. The lowermost subclass can make use of all its super classes' members.



3. Hierarchical Inheritance - one class is extended by many subclasses. It is **one-to-many** relationship.



Example of Member Access and Inheritance

Case 1: Member Variables have no access modifier (default access modifier)

```
class inherit1
{
    int i=10;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        System.out.println("Value of i in Child Class is: "+(i*5));
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

Variable "i" with no access modifier and we are using it in child class. Program runs with no error in this case as members with default access modifier can be used in child class

Output

```
C:\Achin Jain>javac inherit2.java
C:\Achin Jain>java inherittest
Value of i in Child Class is:50
```

Case 2: Member Variables have public/protected access modifier. If a member of class is declared as either public or protected than it can be accessed from child or inherited class.

```
class inherit1
{
    public int i=10;
    protected int j=5;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        j=j+i;
        System.out.println("Value of j in Child Class is:"+j);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

Case 3: Member Variables have private access modifier. If a member of class is declared as private than it cannot be accessed outside the class not even in the inherited class.

```
class inherit1
{
    public int i=10;
    private int j=5;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        j=j+i;
        System.out.println("Value of j in Child Class is:"+j);
    }
}
```

Variable 'j' is declared as Private in class inherit1 which means it cannot be accessed outside the class scope.

In the inherited class an attempt to modify the value of a private variable is made which results in compile time error

Output:

```
C:\Achin Jain>javac inherit2.java
inherit2.java:14: error: j has private access in inherit1
    j=j+i;
    ^
inherit2.java:14: error: j has private access in inherit1
    j=j+i;
    ^
inherit2.java:15: error: j has private access in inherit1
    System.out.println("Value of j in Child Class is:"+j);
    ^
3 errors
```

super Keyword:

super is a reference variable that is used to refer immediate parent class object. Uses of super keyword are as follows:

1. super() is used to invoke immediate parent class constructors
2. super is used to invoke immediate parent class method
3. super is used to refer immediate parent class variable

Example:

```
class inherit1
{
    int i=10;
}
class inherit2 extends inherit1
{
    int i=20;
    void meth1()
    {
        System.out.println("Value of Parent class variable i is :"+super.i);
        System.out.println("Value of Child class variable i is :"+i);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

To print the value of Base class variable in a child class use **super.variable** name

This "i" will print value of local class variable

Output:

```
C:\Achin Jain>javac inherit2.java
C:\Achin Jain>java inherittest
Value of Parent class variable i is :10
Value of Child class variable i is :20
```

Example to call Immediate Parent Class Constructor using super Keyword

The super() keyword can be used to invoke the Parent class constructor as shown in the example below.

Note: super() is added in each class constructor automatically by compiler. As default constructor is provided by compiler automatically but it also adds super() for the first statement. If you are creating your own constructor and you don't have super() as the first statement, compiler will provide super() as the first statement of the constructor.

```

class inherit1
{
    inherit1()
    {
        int i=10;
        System.out.println("Base Class COnstructor is invoked");
    }
}
class inherit2 extends inherit1
{
    int i=20;
    inherit2()
    {
        super();
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
    }
}

```

Output:

```

C:\Achin Jain>javac inherit2.java
C:\Achin Jain>java inherittest
Base Class COnstructor is invoked

```

Example where super() is provided by compiler

```

class inherit2 extends inherit1
{
    int i=20;
    inherit2()
    {
    }
}

```

Same program as above but no super keyword is used. You can see in the o/p below that compiler implicitly adds the super() keyword and same output is seen

Output:

```

C:\Achin Jain>javac inherit2.java
C:\Achin Jain>java inherittest
Base Class COnstructor is invoked

```

Method Overriding

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement. In object-oriented terms, overriding means to override the functionality of an existing method.

```

class inherit1
{
    void meth1()
    {
        System.out.println("Base Method");
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        System.out.println("Child Method");
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}

```

meth1() method is overridden in the child class. Now when meth1() method is invoked from object of child class then compiler will first search for method definition in class from which it is invoked. If method definition is not found then compiler will look for method definition in the parent class.

Ouput:

```

C:\Achin Jain>javac inherit2.java
C:\Achin Jain>java inherittest
Child Method

```

Case – When there is no method definition in child class

```

class inherit1
{
    void meth1()
    {
        System.out.println("Base Method");
    }
}
class inherit2 extends inherit1
{
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}

```

Method definition is not found in child class, so compiler will now search in Parent Class.

Output:

```

C:\Achin Jain>javac inherit2.java
C:\Achin Jain>java inherittest
Base Method

```

Final Keyword

Final keyword can be used in the following ways:

1. **Final Variable** : Once a variable is declared as final, its value cannot be changed during the scope of the program
2. **Final Method** : Method declared as final cannot be overridden
3. **Final Class** : A final class cannot be inherited

Example of Final Variable

```
class finalvar
{
    final int i=10;
    finalvar()
    {
        System.out.println("Value of Final Variable i is :"+i);
        i=i+10;
        System.out.println("Value of Final Variable i after change is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        finalvar obj1 = new finalvar();
    }
}
```

Variable "i" is declared as final

In this code we are trying to modify the value of a Final Variable which will result in error as shown in the output

Output:

```
C:\Achin Jain>javac finaltest.java
finaltest.java:7: error: cannot assign a value to final variable i
    i=i+10;
    ^
1 error
```

Example of Final Method

```
class finalmethod
{
    int i=10;
    final void meth1()
    {
        System.out.println("Value of Final Variable i is :"+i);
    }
}
class childclass extends finalmethod
{
    final void meth1()
    {
        System.out.println("Value of Final Variable i is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        childclass obj1 = new childclass();
        obj1.meth1();
    }
}
```

In this example an attempt to override a final method (meth1()) is made which results in error

Output

```
C:\Achin Jain>javac finaltest.java
finaltest.java:11: error: meth1() in childclass cannot override meth1() in final
method
    final void meth1()
                ^
    overridden method is final
1 error
```

Example of Final Class

```
final class finalmethod
{
    int i=10;
}
class childclass extends finalmethod
{
    childclass()
    {
        System.out.println("Value of Variable i is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        childclass obj1 = new childclass();
    }
}
```

Output

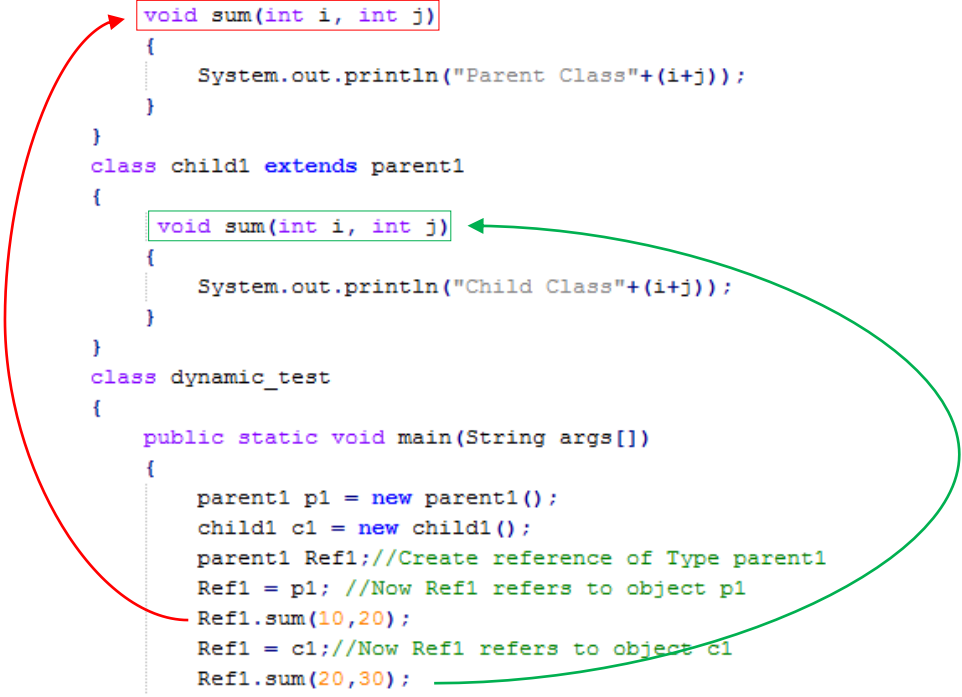
```
C:\Achin Jain>javac finaltest.java
finaltest.java:5: error: cannot inherit from final finalmethod
class childclass extends finalmethod
                        ^
1 error
```

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. Dynamic method dispatch is important because this is how Java implements run-time polymorphism. Method to execution based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. 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.

Example

```
class parent1
{
    void sum(int i, int j)
    {
        System.out.println("Parent Class"+(i+j));
    }
}
class child1 extends parent1
{
    void sum(int i, int j)
    {
        System.out.println("Child Class"+(i+j));
    }
}
class dynamic_test
{
    public static void main(String args[])
    {
        parent1 p1 = new parent1();
        child1 c1 = new child1();
        parent1 Ref1; //Create reference of Type parent1
        Ref1 = p1; //Now Ref1 refers to object p1
        Ref1.sum(10,20);
        Ref1 = c1; //Now Ref1 refers to object c1
        Ref1.sum(20,30);
    }
}
```



In this example a reference is created “**Ref1**” of type parent1. To call an overridden method of any class this reference variable is assigned an object of that class. For ex. To call sum() method of child class Ref1 is assigned object c1 of child class.

Output

```
C:\Achin Jain>javac dynamic_test.java
C:\Achin Jain>java dynamic_test
Parent Class30
Child Class50
C:\Achin Jain>
```

Abstract Classes

When the keyword abstract appears in a class definition, it means that zero or more of its methods are abstract.

- An abstract method has no body.
- Some of the subclass has to override it and provide the implementation.
- Objects cannot be created out of abstract class.
- Abstract classes basically provide a guideline for the properties and methods of an object.

- In order to use abstract classes, they have to be subclassed.
- There are situations in which you want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

Example

```
abstract class parent_class
{
    abstract void meth1();
    void meth2()
    {
        System.out.println("Method of Abstract Class");
    }
}
class child_class extends parent_class
{
    void meth1()
    {
        System.out.println("Method of Child Class");
    }
}
class abstract_test
{
    public static void main(String args[])
    {
        child_class c1 = new child_class();
        c1.meth1();
        c1.meth2();
    }
}
```

Since Method meth1() is declared as abstract it needs to be override in the inherited class. However if the method is not overridden compile time error will be generated as shown below.

```
C:\Achin Jain>javac abstract_test.java
abstract_test.java:10: error: child_class is not abstract and does not override
abstract method meth1() in parent_class
class child_class extends parent_class
^
1 error
```

Output

```
C:\Achin Jain>javac abstract_test.java
C:\Achin Jain>java abstract_test
Method of Child Class
Method of Abstract Class
```

Interface

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

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

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

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

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

Example to Create Interface

```
interface emp
{
    public void salary();
    int increment=10;
}
```

As you can see in the above example method is not declared as abstract explicitly and similarly variable is not declared as static final.

But when Java compiles the code, method is declared as abstract implicitly by compiler and variables are declared as static final.

Example to implement interface

```
class interfacetest implements emp
{
    public void salary()
    {
        System.out.println("Salary of the employee is :"+(increment+5000));
    }
    public static void main(String args[])
    {
        interfacetest obj1 = new interfacetest();
        obj1.salary();
    }
}
```

Definition (Body) of the function salary() declared in the interface emp

"increment" is the variable declared in interface. Value of the variable is 10.

Output

```
C:\Achin Jain>javac interfacetest.java
C:\Achin Jain>java interfacetest
Salary of the employee is :5010
```

Case: When definition of the method declared in interface is not provided in the inherited class. In this case an error will be shown during compile time of the program.

```
class interfacetest implements emp
{
    /*public void salary()
    {
        System.out.println("Salary of the employee is :"+(increment+5000));
    }*/
}
```

Output:

```
C:\Achin Jain>javac interfacetest.java
interfacetest.java:1: error: interfacetest is not abstract and does not override
abstract method salary() in emp
class interfacetest implements emp
^
1 error
```

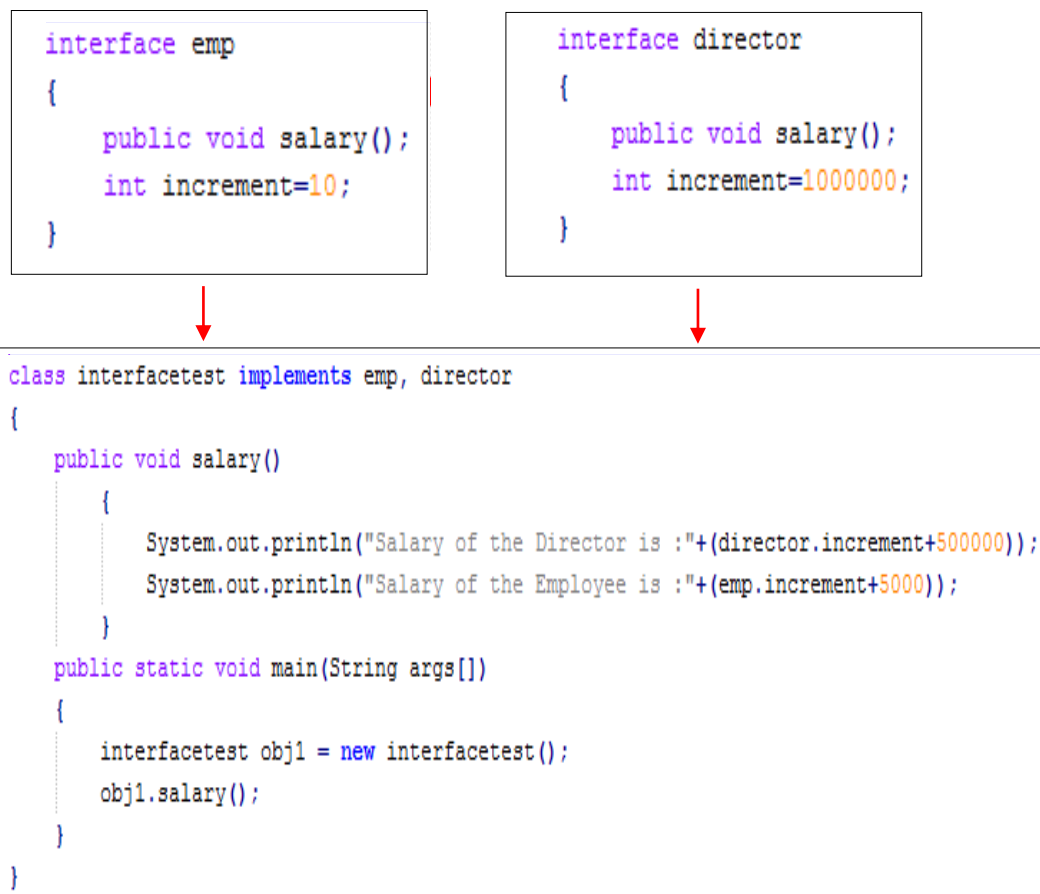
Case: Try to modify the value of variable declared in interface

```
public void salary()
{
    increment = increment+20;
    System.out.println("Salary of the employee is :"+(increment+5000));
}
```

Output

```
C:\Achin Jain>javac interfacetest.java
interfacetest.java:5: error: cannot assign a value to final variable increment
    increment = increment+20;
    ^
1 error
```

How Interface provide Multiple Inheritance in Java



In the above example, two interfaces are created with names “emp” and “director” and both interfaces contain same method salary(). In the class interfacetest we have implemented both the interfaces. Now as per the definition of interface a method

declared must be overridden in the class that implements the interface. Although in the above example there are two salary methods declared in different interfaces, but in the child class there is only single instance of the method. So whenever a call is made to the salary method it is always the overridden method that gets invoked, leaving no room for ambiguity.

Package

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc. A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are:

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

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related. Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Example to Create Package

When creating a package, you should choose a name for the package and put a package statement with that name at the top of every source file that contains the classes, interfaces that you want to include in the package. The package statement should be the first line in the source file.

In the example a package with name “***achin_java***” is created with statement “*package achin_java;*”.

```

package achin_java;

public class add
{
    int a;
    public int b;
    private int c;
    protected int d;
    public void add_meth()
    {
        System.out.println("Sum of variables passed is:"+(a+b+c+d));
    }
}

```

Importing the Package in other Class

```

import achin_java.*;

class packtest
{
    public static void main(String args[])
    {
        add obj = new add();
        obj.add_meth();
    }
}

```

To import a package use keyword “***import***” followed by package name and class name. If you want to import all classes use .* instead of single class name.

Output:

```

C:\Achin Jain>javac packtest.java
C:\Achin Jain>java packtest
Sum of variables passed is:0

```

Case 1: Trying to access a public variable outside the package.

In the code of package “achin_java” four different types of variables are declared. Variable ‘b’ is declared as public. See the following code in which we have assigned a value to ‘b’ in class packtest.

```

add obj = new add();
obj.b=10;
obj.add_meth();

```

Output

```
C:\Achin Jain>javac packtest.java
C:\Achin Jain>java packtest
Sum of variables passed is:10
```

Case 2: Trying to access a variable with default access modifier outside the package.

Variable 'a' is declared as public. See the following code in which we have assigned a value to 'a' in class packtest. You will see an error as shown in the output

```
add obj = new add();
obj.b=10;
obj.a=20;
obj.add_meth();
```

Output:

```
C:\Achin Jain>javac packtest.java
packtest.java:9: error: a is not public in add; cannot be accessed from outside package
    obj.a=20;
        ^
```

Case 3: Trying to access a variable with private access modifier outside the package. This is clear case in which you don't have permission to access private variable outside its scope.

Case 4: Trying to access a variable with protected access modifier outside the package.

In this there are two different cases. One is trying to access the protected variable outside the package in the same way as described above for other cases. Other case if we are inheriting a class defined in a package and then trying to use the protected declared member.

When class is not inherited

```
add obj = new add();
obj.b=10;
obj.d=40;
obj.add_meth();
```


Output:

```
C:\Achin Jain>javac packtest.java
packtest.java:9: error: d has protected access in add
    obj.d=40;
      ^
1 error
```

Case when class is inherited

```
import achin_java.*;

class packtest extends add
{
    public static void main(String args[])
    {
        packtest obj = new packtest();
        obj.b=10;
        obj.d=40;
        obj.add_meth();
    }
}
```

We can access a protected member in sub class of different package. As in this example packtest class inherits class add and now we are accessing protected variable '*d*' which results in no error and program executes as expected.

Output

```
C:\Achin Jain>javac packtest.java
C:\Achin Jain>java packtest
Sum of variables passed is:50
```

Access Protection in Packages

	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

Exceptional Handling

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

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

“Exceptional Handling is a task to maintain normal flow of the program. For this we should try to catch the exception object thrown by the error condition and then display appropriate message for taking corrective actions”

Types of Exceptions

- 1. Checked Exception:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. Checked exception can also be defined as *“The classes that extend the Throwable class except RuntimeException and Error are known as Checked Exceptions”*. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions are checked at compile-time and cannot simply be ignored at the time of compilation. Example of Checked Exception are IOException, SQLException etc.
- 2. Unchecked Exception:** Also known as Runtime Exceptions and they are ignored at the time of compilation but checked during execution of the program. Unchecked Exceptions can also be defined as *“The Classes that extend the RuntimeException class are known as Unchecked Exceptions”*. Example are ArithmeticException, NullPointerException etc.
- 3. Error:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Hierarchy of Exception

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

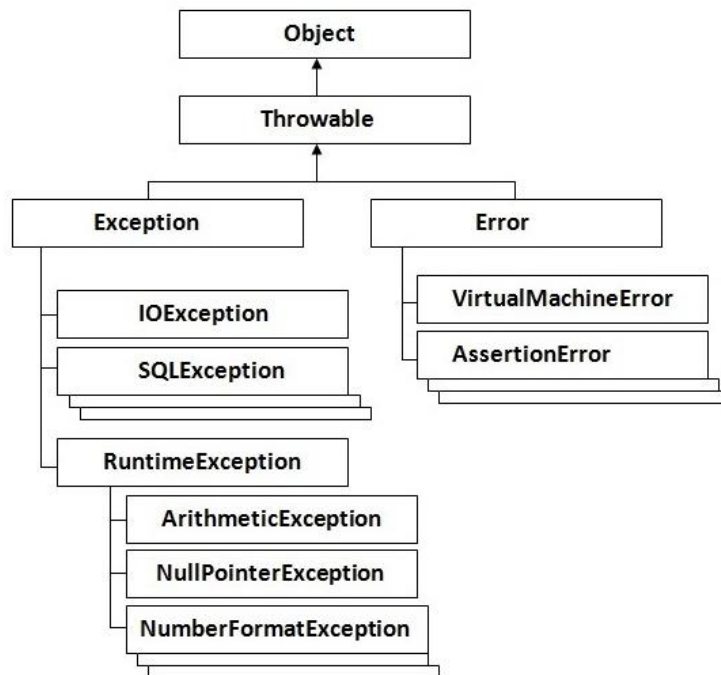


Table of JAVA – Built in Exceptions

Following is the list of Java Unchecked RuntimeException

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with 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.
UnsupportedOperationException	An unsupported operation was encountered.

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

Exception	Description
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.

Handling Exceptions in Java


Following five keywords are used to handle an exception in Java:

1. try
2. catch
3. finally
4. throw
5. throws

try –catch block

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

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



Write block of code here that is likely to cause an error condition and throws an exception

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

Example of Program without Exceptional Handling

```
class excep1
{
    public static void main(String args[])
    {
        int i=100/Integer.parseInt(args[0]);
        System.out.println("Value of i is:"+i);
    }
}
```

This statement can cause error as divide by Zero is an ArithmeticException

Output:

```
C:\Achin Jain>java excep1 12
Value of i is:8

C:\Achin Jain>java excep1 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at excep1.main(excep1.java:5)
```

Same Program with Exception Handling

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("Code after try-catch");
    }
}
```

Now as the statement which can cause error condition is wrapped under try block and catch block is also present to handle the exception object thrown. In this case even if there is an error rest of the program will execute normally

Output

```
C:\Achin Jain>java excep1 12
Value of i is:8
Code after try-catch

C:\Achin Jain>java excep1 0
java.lang.ArithmeticException: / by zero
Code after try-catch
```

Multiple Catch Blocks:

A try block can be followed by multiple catch blocks, but when we use multiple catch statements it is important that exception subclasses must come before any of their superclasses. The reason is “a catch statement with superclass will catch exceptions of that type plus any of its subclass, thus causing a catch statement with subclass exception a non-reachable code which is error in JAVA”.

Example:

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        }
        catch(Exception e1)
        {
            System.out.println(e1);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("Code after try-catch");
    }
}
```

In the example, two catch statement are used but first one is of type Exception which is a superclass of ArithmeticException (used in second catch). So any exception thrown will be caught by first catch block which makes second block unreachable and error is shown during compile time

Output

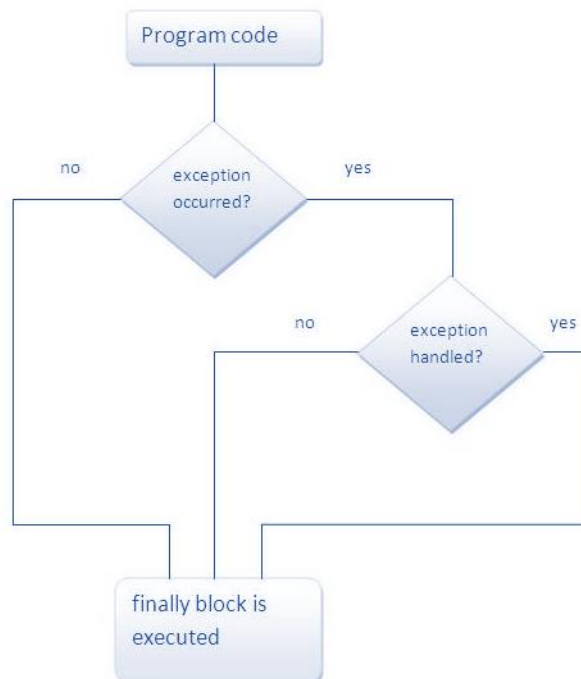
```
C:\Achin Jain>javac excep1.java
excep1.java:14: error: exception ArithmeticException has already been caught
    catch(ArithmeticException e)
    ^
1 error
```

However if the order of the catch blocks is reversed like shown below, then program will execute normally

```
catch(ArithmeticException e)
{
    System.out.println(e);
}
catch(Exception e1)
{
    System.out.println(e1);
}
```

Finally Block

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.



Example of Finally Statement

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        }
        catch(ArithmeticException e1)
        {
            System.out.println(e1);
        }
        finally
        {
            System.out.println("Finally Code Executed");
        }
        System.out.println("Code after try-catch");
    }
}
```

Output 1

In the first case no command line arguments are passed which will throw `ArrayIndexOutOfBoundsException` and in the above code we are handling only `ArithmeticException` which will cause the system to terminate and remaining program will not run. But in this case also the statement written in the finally block will get executed as shown below:

```
C:\Achin Jain>java excep1
Finally Code Executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at excep1.main(excep1.java:7)
```

In second case '0' is passed as command line argument to let program throw `ArithmeticException` which will eventually be handled by catch block. See the output below which clearly shows that remaining part of the code will also run along with finally statement.

```
C:\Achin Jain>java excep1 0
java.lang.ArithmeticException: / by zero
Finally Code Executed
Code after try-catch
```

In third case '5' is passed as command line argument which is perfectly fine and in this case no exception will be thrown. Now see the output below, in this case also finally statement will get executed.

```
C:\Achin Jain>java excep1 5
Value of i is:20
Finally Code Executed
Code after try-catch
```

Throw Keyword

The throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception. The throw keyword is normally used to throw custom exception.

Example

In the example shown below a method `validage(int i)` is used which will check the value of passed parameter *i* and if the value is less than 18 then a `ArithmeticException` is thrown. Now as you can see when we have called the method no try catch block is used which results in termination of the program and message is displayed as "not valid" which is passed during throwing of `ArithmeticException` object.


```

class excep1
{
    static void validage(int i)
    {
        if(i<18)
        {
            throw new ArithmeticException("not valid");
        }
        else
        {
            System.out.println("Welcome");
        }
    }
    public static void main(String args[])
    {
        validage(12);
        System.out.println("Code after try-catch");
    }
}

```

Output

```

C:\Achin Jain>javac excep1.java
C:\Achin Jain>java excep1
Exception in thread "main" java.lang.ArithmeticException: not valid
    at excep1.validage(excep1.java:7)
    at excep1.main(excep1.java:16)

```

However if during call of validage method try-catch block has been used then the program will run normally

```

try
{
    validage(12);
}
catch(ArithmeticException e)
{
    System.out.println(e);
}

```

Output

```

C:\Achin Jain>javac excep1.java
C:\Achin Jain>java excep1
java.lang.ArithmeticException: not valid
Code after try-catch

```

Throws Keyword

The throws keyword is used to declare the exception, it provide information to the programmer that there may occur an exception so during call of that method, and programmer must use exceptional handling mechanism. Throws keyword is also used to propagate checked exception.

Example

In this example, exception is created by extending Exception class and the custom exception is declared in the method validage(int i)

```
class ajexception extends Exception
{
    ajexception(String s)
    {
        super(s);
    }
}

class excep2
{
    static void validage(int i) throws ajexception
    {
        if(i<18)
        {
            throw new ajexception("not valid");
        }
    }
    public static void main(String args[])
    {
        validage(12);
        System.out.println("Code after try-catch");
    }
}
```

Code to create custom exception with name "ajexception"

Case 1: During call of validage method exceptional handling is not used and code looks like this and error is displayed in the compilation of the code.

```
public static void main(String args[])
{
    validage(12);
    System.out.println("Code after try-catch");
}
```

Output

```
C:\Achin Jain>javac excep2.java
excep2.java:19: error: unreported exception ajexception; must be caught or declared to be thrown
    validage(12);
    ^
```

Case 2: During call of method validage exceptional handling is used with try-catch keyword like this and the program runs as expected.

```
try
{
    validage(12);
}
catch(ajexception aj)
{
    System.out.println(aj);
}
```

Output:

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
ajexception: not valid
Code after try-catch
```

Case 3: During call of method validage exceptional handling is used without try-catch keyword and throws keyword is used in main method as shown below

```
public static void main(String args[]) throws ajexception
{
    validage(12);
    System.out.println("Code after try-catch");
}
```

There will be no error now during compile time, but program will gets terminated when exception event takes place.

Output

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
Exception in thread "main" ajexception: not valid
    at excep2.validage(excep2.java:14)
    at excep2.main(excep2.java:19)
```

Case 4: Try to propagate custom exception not of type RuntimeException without declaring in method using throws keyword. This will give compile time error

```
static void validage(int i) //throws ajexception
{
    if(i<18)
    {
        throw new ajexception("not valid");
    }
}
```

Output:

```
C:\Achin Jain>javac excep2.java
excep2.java:14: error: unreported exception ajexception; must be caught or declared to be thrown
        throw new ajexception("not valid");
              ^
1 error
```

Case 5: Make custom exception by extending RuntimeException Class and try the same method as use for Case 4. There will be no error now and the program runs as expected

```
class ajexception extends RuntimeException
{
    ajexception(String s)
    {
        super(s);
    }
}
```

Output

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
ajexception: not valid
Code after try-catch
```

Important Points in Exceptional Handling

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

Declaring your Own Exceptions

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

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

[Example to create custom exception](#) is shown in the section above.

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A **process** consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process.

There are two distinct types of **Multitasking** i.e. Processor-Based and Thread-Based multitasking.

Q: What is the difference between thread-based and process-based multitasking?

Ans: As both are types of multitasking there is very basic difference between the two. **Process-Based multitasking** is a feature that allows your computer to run two or more programs concurrently. For example you can listen to music and at the same time chat with your friends on Facebook using browser. In Thread-based multitasking, thread is the smallest unit of code, which means a single program can perform two or more tasks simultaneously. For example a text editor can print and at the same time you can edit text provided that those two tasks are performed by separate threads.

Q: Why multitasking thread requires less overhead than multitasking processor?

Ans: A multitasking thread requires less overhead than multitasking processor because of the following reasons:

- Processes are heavyweight tasks where threads are lightweight
- Processes require their own separate address space where threads share the address space
- Interprocess communication is expensive and limited where Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

Benefits of Multithreading

1. Enables programmers to do multiple things at one time

2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution
3. Improved performance and concurrency
4. Simultaneous access to multiple applications

Life Cycle of Thread

A thread can be in any of the five following states

1. **Newborn State:** When a thread object is created a new thread is born and said to be in Newborn state.
2. **Runnable State:** If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion
3. **Running State:** It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs
 - a. Thread give up its control on its own and it can happen in the following situations
 - i. A thread gets suspended using **suspend()** method which can only be revived with **resume()** method
 - ii. A thread is made to sleep for a specified period of time using **sleep(time)** method, where time in milliseconds
 - iii. A thread is made to wait for some event to occur using **wait ()** method. In this case a thread can be scheduled to run again using **notify ()** method.
 - b. A thread is pre-empted by a higher priority thread
4. **Blocked State:** If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.
5. **Dead State:** A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.

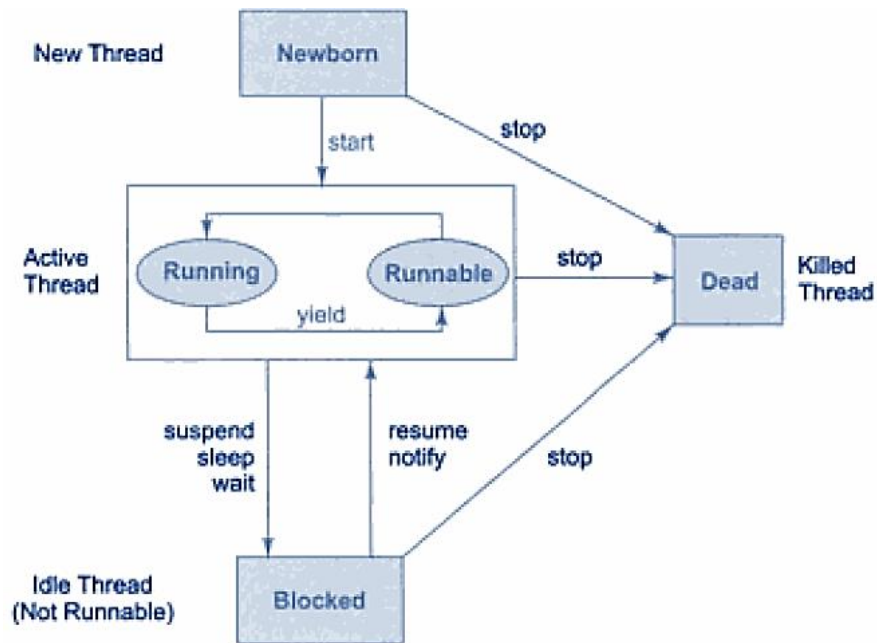


Fig: Life Cycle of Thread

Main Thread

Every time a Java program starts up, one thread begins running which is called as the main thread of the program because it is the one that is executed when your program begins.

- Child threads are produced from main thread
- Often it is the last thread to finish execution as it performs various shut down operations

Creating a Thread

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Create Thread by Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called **run()**, which is declared like this:

```
public void run( )
```

You will define the code that constitutes the new thread inside **run()** method. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName. After the new thread is created, it will not start running until you call its **start()** method, which is declared within Thread. The start() method is shown here:

```
void start();
```

Example to Create a Thread using Runnable Interface

```
class t1 implements Runnable
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t1.java
C:\NIEC Java>java t1
Thread is Running
C:\NIEC Java>
```

Create Thread by Extending Thread

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The **extending class must override the run() method**,

which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

Example to Create a Thread by Extending Thread Class

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
C:\NIEC Java>
```

Thread Methods

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread

	to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

Q: Can we start a thread twice?

Ans: No, if a thread is started it can never be started again, if you do so, an `IllegalThreadStateException` is thrown. Example is shown below in which a same thread is coded to start again

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
        obj1.start();
    }
}
```

As you can see two statements to start a same thread is written in the code which will not give error during compilation but when you run it you can see an Exception as shown in the Output Screenshot.

Output:

```
C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Unknown Source)
    at t2.main(t2.java:11)
```

Use of Yield() Method

Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled

Example

```
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) yield();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
}
class B extends Thread
{
    public void run()
    {
        for(int j=1; j<=5; j++)
        {
            System.out.println("B:" +j);
        }
        System.out.println("Exit from B");
    }
}
class yieldtest
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        a.start();
        b.start();
    }
}
```

Condition is checked and when i==2
yield() method is evoked taking
control to thread B

As you can see in the output below, thread A gets started and when condition if(i==2) gets satisfied yield() method gets evoked and the control is relinquished from thread A to thread B which run to its completion and only after that thread a regain the control back.

Output

```
C:\NIEC Java>javac yieldtest.java
C:\NIEC Java>java yieldtest
A:1
B:1
B:2
B:3
B:4
B:5
Exit from B
A:2
A:3
A:4
A:5
Exit from A
```

Use of stop() Method

The stop() method kills the thread on execution

Example

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) stop();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 stop() method is evoked causing termination of thread execution

Output

```
C:\NIEC Java>java C
A:1
```

Use of sleep() Method

Causes the currently running thread to block for at least the specified number of milliseconds.

You need to handle exception while using sleep() method.

Example

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            try
            {
                if(i==2) sleep(1000);
            }
            catch(Exception e)
            {
            }
            System.out.println("A: " +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 sleep() method is evoked which halts the execution of the thread for 1000 milliseconds. When you see output there is no change but there is delay in execution.

Output

```
C:\NIEC Java>javac C.java
C:\NIEC Java>java C
A:1
A:2
A:3
A:4
A:5
Exit from A
```

Use of suspend() and resume() method

A suspended thread can be revived by using the resume() method. This approach is useful when we want to suspend a thread for some time due to certain reason but do not want to kill it.

Following is the example in which two threads C and A are created. Thread C is started ahead of Thread A, but C is suspended using suspend() method causing Thread A to get hold of the processor allowing it to run and when Thread C is resumed using resume() method it runs to its completion.

Example

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("C:" +i);
        }
        System.out.println("Exit from C");
    }
}
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
}
class suspendtest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        c.start();
        a.start();
        c.suspend();
        c.resume();
    }
}
```

Although Thread 'C' is started earlier than Thread 'A' but due to suspend method Thread 'A' gets completed ahead of Thread 'C'

Output

```
C:\Achin Jain>java suspendtest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C
```

Thread Priority

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between **MIN_PRIORITY (a constant of 1)** and **MAX_PRIORITY (a constant of 10)**. By default, every thread is given priority **NORM_PRIORITY (a constant of 5)**.

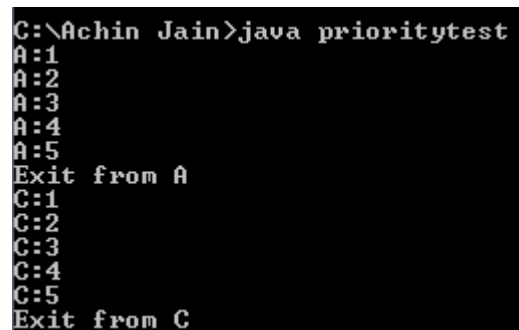
Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Example

```
class prioritytest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        a.setPriority(10);
        c.setPriority(1);
        c.start();
        a.start();
    }
}
```

In the above code, you can see Priorities of Thread is set to maximum for Thread A which lets it to run to completion ahead of C which is set to minimum priority.

Output:



```
C:\Achin Jain>java prioritytest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C
```

Use of isAlive() and join() method

The java.lang.Thread.isAlive() method tests if this thread is alive. A thread is alive if it has been started and has not yet died. Following is the declaration for java.lang.Thread.isAlive() method

public final boolean isAlive()

This method returns true if this thread is alive, false otherwise.

join() method waits for a thread to die. It causes the currently thread to stop executing until the thread it joins with completes its task.

Example

```
class A extends Thread
{
    public void run()
    {
```

```
        System.out.println("Status:" + isAlive());
    }
```

At this point Thread A is alive so the value gets printed by **isAlive()** method is "**true**"

```
class alivetest
{
```

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

```
        A a = new A();
```

```
        a.start();
```

```
        try
```

```
        {
            a.join();
```

join() method is called from Thread A which stops executing of further statement until A is Dead

```
        }
        catch (InterruptedException e)
```

```
        {}
```

```
        System.out.println("Status:" + a.isAlive());
```

```
    }
```

Now isAlive() method returns the value false as the Thread A is complete

Output

```
C:\Achin Jain>java alivetest
Status:true
Status:false
```


Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization. The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

synchronized(object)

```
{  
    // statements to be synchronized  
}
```

Problem without using Synchronization

In the following example method updatesum() is not synchronized and access by both the threads simultaneously which results in inconsistent output. Making a method synchronized, Java creates a “monitor” and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of the code. Writing the method as synchronized will make one thread enter the method and till execution is not complete no other thread can get access to the method.

```
synchronized void updatesum(int i)  
{  
    Thread t = Thread.currentThread();  
    for(int n=1; n<=5; n++)  
    {  
        System.out.println(t.getName()+" : "+(i+n));  
    }  
}
```

```

class update
{
    void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n));
        }
    }
}

class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}

class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}

```

Diagram illustrating the synchronization of the `updatesum` method:

- The original method signature `void updatesum(int i)` is shown in a red box.
- An arrow points to the modified signature `synchronized void updatesum(int i)`, also in a red box.
- A red arrow points from the modified signature to the output window.

Output when
method is declared
as synchronized

```

C:\Achin Jain>java syntest
Thread A : 11
Thread A : 12
Thread A : 13
Thread A : 14
Thread A : 15
Thread B : 11
Thread B : 12
Thread B : 13
Thread B : 14
Thread B : 15

```

Output

```

C:\Achin Jain>java syntest
Thread A : 11
Thread B : 11
Thread A : 12
Thread B : 12
Thread A : 13
Thread B : 13
Thread A : 14
Thread B : 14
Thread A : 15
Thread B : 15

```

Interthread Communication

It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methods of Object Class:

- **wait():** This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- **notify():** This method wakes up the first thread that called wait() on the same object.
- **notifyAll():** This method wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

Example

```
class customer
{
    int amount = 0;
    int flag = 0;
    public synchronized int withdraw(int amount)
    {
        System.out.println(Thread.currentThread().getName()+" : is going to withdraw");
        if(flag==0)
        {
            try
            {
                System.out.println("Waiting...");
                wait();
            }
            catch(Exception e)
            {
            }
        }
        this.amount-=amount;
        System.out.println("Withdraw Complete");
        return amount;
    }
    public synchronized void deposit(int amount)
    {
        System.out.println(Thread.currentThread().getName()+" : is going to Deposit");
        this.amount+=amount;
        notifyAll();
        System.out.println("Deposit Complete");
        flag=1;
    }
}
```

The diagram consists of two red arrows. One arrow originates from the `wait();` line in the `withdraw` method and points to a text box. The other arrow originates from the `notifyAll();` line in the `deposit` method and points to the same text box.

If both these methods are commented which means there is no communication, output will be inconsistent. See [Output 2](#)

```

class threadcomm
{
    public static void main(String args[])
    {
        final customer c = new customer();
        Thread t1 = new Thread()
        {
            public void run()
            {
                c.withdraw(5000);
                System.out.println("After withdraw Amount is :"+ c.amount);
            }
        };
        Thread t2 = new Thread()
        {
            public void run()
            {
                c.deposit(10000);
                System.out.println("After Deposit Amount is :"+ c.amount);
            }
        };
        t1.start();
        t2.start();
    }
}

```

Output 1:

```

C:\Achin Jain>java threadcomm
Thread-0 : is going to withdraw
Waiting....
Thread-1 : is going to Deposit
Deposit Complete
After Deposit Amount is :10000
Withdraw Complete
After withdraw Amount is :5000

```

Output 2:

```

C:\Achin Jain>java threadcomm
Thread-0 : is going to withdraw
Waiting....
Withdraw Complete
After withdraw Amount is :-5000
Thread-1 : is going to Deposit
Deposit Complete
After Deposit Amount is :5000

```

JAVA - PACKAGES

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

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

Some of the existing packages in Java are::

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

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

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

Creating a package:

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

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

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

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

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

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

Example:

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

Below given package example contains interface named *animals*:

```
/* File name : Animal.java */
package animals;
interface Animal {
    public void eat();
    public void travel();
}
```

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

```
package animals;

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

```

public class MammalInt implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}

```

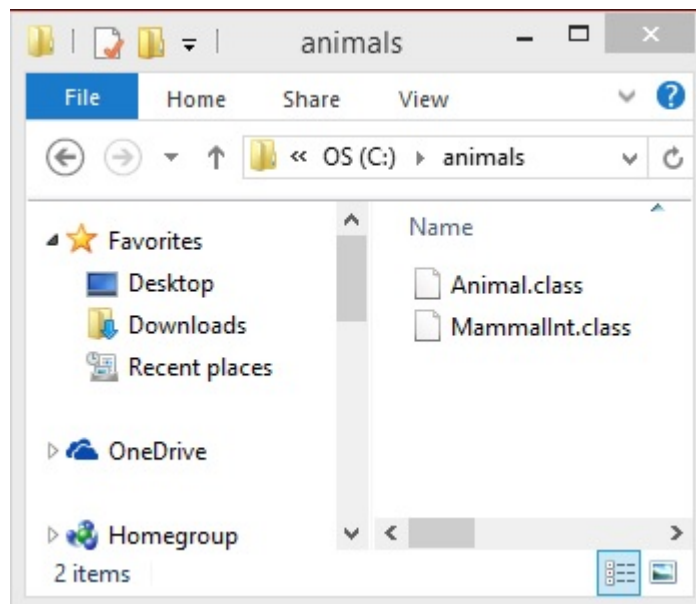
Now compile the java files as shown below:

```

$ javac -d . Animal.java
$ javac -d . MammalInt.java

```

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



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

```

$ java animals.MammalInt
ammal eats
ammal travels

```

The import Keyword:

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

Example:

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

following Boss class.

```
package payroll;

public class Boss
{
    public void payEmployee(Employee e)
    {
        e.mailCheck();
    }
}
```

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

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

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card *. For example:

```
import payroll.*;
```

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

```
import payroll.Employee;
```

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

The Directory Structure of Packages:

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

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

Here is simple way of managing your files in Java:

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

```
// File Name : Car.java

package vehicle;

public class Car {
    // Class implementation.
}
```

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

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

Now, the qualified class name and pathname would be as below:

- Class name -> vehicle.Car
- Path name -> vehicle\Car.java *in windows*

In general, a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

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

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

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

For example:

```
// File Name: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

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

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

This would put compiled files as follows:

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

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

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

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

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

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

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

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

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

Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and UNIX

Bourneshell:

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

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

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

To set the CLASSPATH variable:

- In Windows -> set CLASSPATH=C:\users\jack\java\classes
- In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

Loading [MathJax]/jax/output/HTML-CSS/jax.js