

Unit 3**Exception Handling****3.1.1 Fundamentals of Exception Handling****Exception**

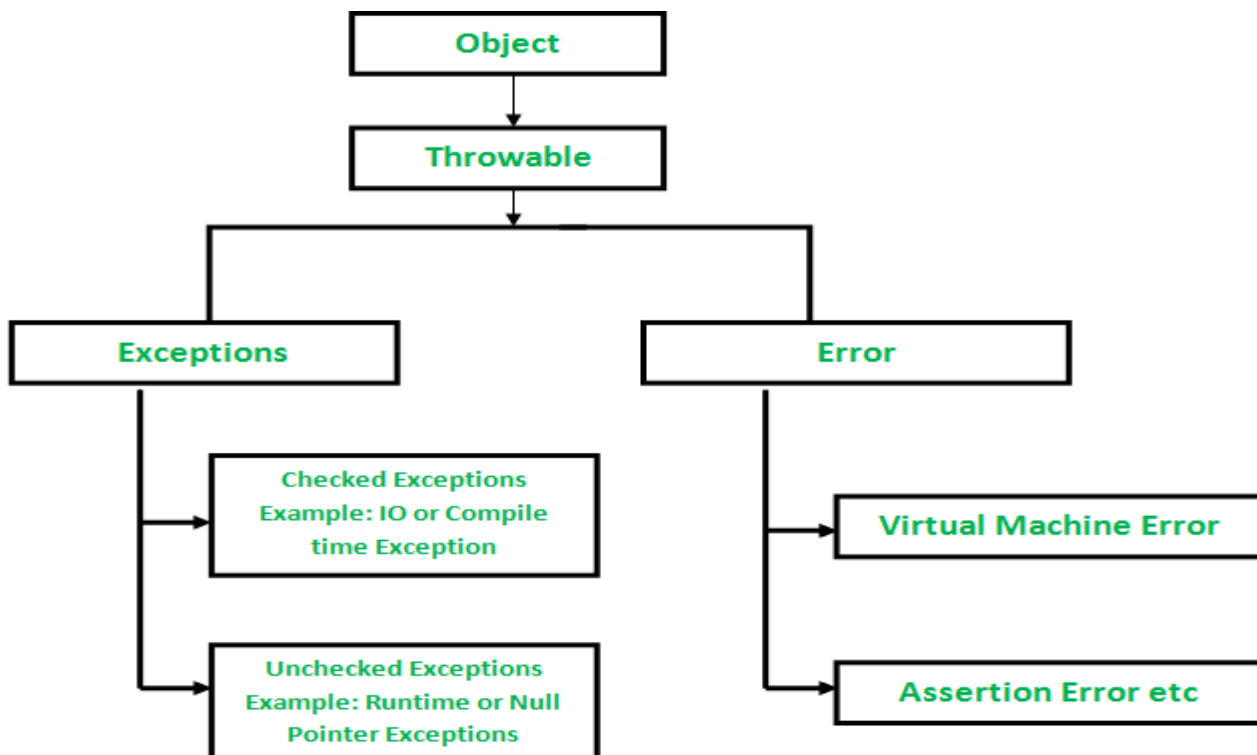
A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an exception handler.

Exception Handling

Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of program. The term exception means exceptional condition, it is a problem that may arise during the execution of program. A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

Exception class Hierarchy

All exception types are subclasses of class Throwable, which is at the top of exception class hierarchy.



- Exception class is for exceptional conditions that program should catch. This class is extended to create user specific exception classes.
- RuntimeException is a subclass of Exception. Exceptions under this class are automatically defined for programs.

3.1.2 Exception Types

- **Checked Exception**

The exception that can be predicted by the programmer. The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions

e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

- **Unchecked Exception**

Unchecked exceptions are the class that extends RuntimeException. Unchecked exceptions are ignored at compile time.

Example : ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

- **Error**

Errors are typically ignored in code because you can rarely do anything about an error. For example if stack overflow occurs, an error will arise. This type of error is not possible to handle in code. Error is irrecoverable

e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios of Exception Handling where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs a NullPointerException.

1. `String s=null;`
2. `System.out.println(s.length());//NullPointerException`

3) Scenario where `NumberFormatException` occurs

The wrong formatting of any value, may occur `NumberFormatException`. Suppose I have a string variable that have characters, converting this variable into digit will occur `NumberFormatException`.

1. `String s="abc";`
2. `int i=Integer.parseInt(s);//NumberFormatException`

4) Scenario where `ArrayIndexOutOfBoundsException` occurs

If you are inserting any value in the wrong index, it would result `ArrayIndexOutOfBoundsException` as shown below:

1. `int a[]=new int[5];`
2. `a[10]=50; //ArrayIndexOutOfBoundsException`

3.1.3 Termination & Resumptive models

There are two basic models in exception handling theory.

Introduction:

When an exception is thrown ,control is transferred to the catch block that handles the error. The programmer now has two choices.

Programmer can print the error message and exit from the program. This technique is called as **Termination Model**.

The Programmer can also continue execution by calling some other function after printing the error message This technique is called as **Resumptive model**

If exception handling is not provided,then in a java application the program terminates automatically after printing the default exception message.

Example for Termination Model

```

class terminationtest

{

    public static void main(String[] args)
    {
        int i=50;
        int j=0;
        int data;
        try
        {
            data=i/j; //may throw exception
        }
        // handling the exception
        catch(Exception e)
        {
            // resolving the exception in catch block
            System.out.println(e);
            System.exit(0);
        }

    }
}

```

Explanation:

Once the exception is ariased repective catch will be executed.In catch we can do two things

i) we can just display error message and exit from program without executing the code after the catch block by using System.exit(0) method as show in above example.[Termination Model]

ii)we can call another method which changes the value of b (or) changes the value of b in catch block so that execution continues after catching the exception.[Resumptive Model]

Resumptive Model

“Resumption means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time.If you want this, try placing your try-catch in a while loop that keeps reentering the try block until the result is satisfactory”

Example for Resumptive Model

```

public class ResumeModel {

```

```

int change()
{
return 10;
}

public static void main(String[] args) {
    int i=50;
    int j=0;
    int data;
    try
    {
        data=i/j; //may throw exception
    }
    // handling the exception
    catch(Exception e)
    {
        // resolving the exception in catch block
        System.out.println(e);
        ResumeModel obj =new ResumeModel();
        j=obj.change();
    }
    data=i/j;
    System.out.println("data value is"+data);

}
}

```

Output:

```

java.lang.ArithmeticException: / by zero
data value is 5

```

3.1.4 Uncaught Exceptions

When we don't handle the exceptions, they lead to unexpected program termination. Lets take an example for better understanding.

```

class UncaughtException
{
public static void main(String args[])
{
    int a = 0;
    int b = 7/a;    // Divide by zero, will lead to exception
}
}

```

This will lead to an exception at runtime, hence the Java run-time system will construct an exception and then throw it. As we don't have any mechanism for handling exception in the above program, hence the default handler will handle the exception and will print the details of the exception on the terminal.

The diagram shows a Java exception message: `java.lang.ArithmeticException: / by zero` on the first line and `at UncaughtException.main(UncaughtException.java:4)` on the second line. Annotations with arrows point to specific parts: 'name and description of Exception' points to the first line; 'class name' points to `java.lang`; 'file name' points to `UncaughtException.java`; and 'Stack Trace (line at which exception occurred)' points to `:4`.

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

name and description of Exception

class name

file name

Stack Trace
(line at which exception occurred)

3.1.5 Using try and catch statements

Exception Handling Mechanism

In java, exception handling is done using five keywords,

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

Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs.

Using try and catch

Try is used to guard a block of code in which exception may occur. This block of code is called guarded region. A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in guarded code, the catch block that follows the try is checked, if the type of exception that occurred is listed in the catch block then the exception is handed over to the catch block which then handles it.

Example using Try and catch

```
class Excp {  
  
    public static void main(String args[])  
  
    {  
  
        int a,b,c;  
  
        try {  
  
            a=0;  
  
            b=10;  
  
            c=b/a;  
  
            System.out.println("This line will not be executed");  
  
        }  
  
        catch(ArithmeticException e) {  
  
            System.out.println("Divided by zero");  
  
        }  
  
        System.out.println("After exception is handled");  
  
    }  
  
}
```

Output :

Divided by zero

After exception is handled

An exception will be thrown by this program as we are trying to divide a number by zero inside **try** block. The program control is transferred outside **try** block. Thus the line *"This line will not be executed"* is never parsed by the compiler. The exception thrown is handled in **catch** block. Once the exception is handled the program continues with the next line in the program. Thus the line *"After exception is handled"* is printed.

3.1.6 Multiple catch clauses

Multiple catch blocks:

A try block can be followed by multiple catch blocks. You can have any number of catch blocks after a single try block. If an exception occurs in the guarded code the exception is passed to the first catch block in the list. If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continues until the exception is caught or falls through all catches.

Example for Multiple Catch blocks

```
class Excep {  
  
    public static void main(String[] args)  
  
    {  
  
        try {  
  
            int arr[]={ 1,2};  
  
            arr[2]=3/0;  
  
        }  
  
        catch(ArithmeticExceptionae)  
  
        {  
  
            System.out.println("divide by zero");  
  
        }  
  
        catch(ArrayIndexOutOfBoundsException e)  
  
        {  
  
            System.out.println("array index out of bound exception");  
  
        }  
  
    }  
  
}
```


Output :

divide by zero

Example for Unreachable Catch block

While using multiple **catch** statements, it is important to remember that exception sub classes inside **catch** must come before any of their super classes otherwise it will lead to compile time error.

```
class Excep {  
  
    public static void main(String[] args)  
  
    {  
  
        try  
  
        {  
  
            int arr[]={ 1,2};  
  
            arr[2]=3/0;  
  
        }  
  
        catch(Exception e) //This block handles all Exception  
  
        {  
  
            System.out.println("Generic exception");  
  
        }  
  
        catch(ArrayIndexOutOfBoundsException e) //This block is unreachable  
  
        {  
  
            System.out.println("array index out of bound exception");  
  
        }  
  
    }  
  
}
```

3.1.7 Nested try statements

try statement can be **nested** inside another block of **try**. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner **try** block does not have a **catch** handler for a particular exception then the outer **try** is checked for match.

```
class Excep {  
  
    public static void main(String[] args) {  
  
        try  
  
        {  
  
            int arr[]={ 5,0,1,2};  
  
            try  
  
            {  
  
                int x=arr[3]/arr[1];  
  
                }  
  
            catch(ArithmeticExceptionae)  
  
            {  
  
                System.out.println("divide by zero");  
  
                }  
  
            arr[4]=3;  
  
            }  
  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("array index out of bound exception");  
        }  
    }  
}
```

Important points to Remember

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
2. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
3. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.
4. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.
5. Only the object of Throwable class or its subclasses can be thrown.

3.1.8 throw, throws and finally Statements

throw Keyword

throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

Syntax :

throw*ThrowableInstance*

Creating Instance of Throwable class

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with **new** operator.

Ex:`new NullPointerException("test");`

This constructs an instance of NullPointerException with name test.

Example demonstrating throw Keyword

```
class Test {  
  
    static void show() {  
  
        try {  
  
            throw new ArithmeticException("demo");  
  
        }  
  
        catch(ArithmeticException e) {  
  
            System.out.println("Exception caught");  
  
        }  
  
    }  
  
    public static void main(String args[])  
    {  
        show();  
    }  
}
```

In the above example the `avg()` method throw an instance of `ArithmeticException`, which is successfully handled using the catch statement.

throws Keyword

Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws** keyword.

Syntax :

```
typemethod_name(parameter_list)throwsexception_list  
  
{  
  
    //definition of method  
  
}
```

NOTE : It is necessary for all exceptions, except the exceptions of type **Error** and **RuntimeException**, or any of their subclass.

Example demonstrating throws Keyword

```
import java.io.*;

public class ThrowExample {

    void mymethod(int num)throws IOException, ClassNotFoundException{

        if(num==1)

            throw new IOException("Exception Message1");

        else

            throw new ClassNotFoundException("Exception Message2");

    }

}

class Demo

{

    public static void main(String args[]){

        try{

            ThrowExample obj=new ThrowExample();

            obj.mymethod(1);

        }

        catch(Exception ex)

        {

            System.out.println(ex);

        }

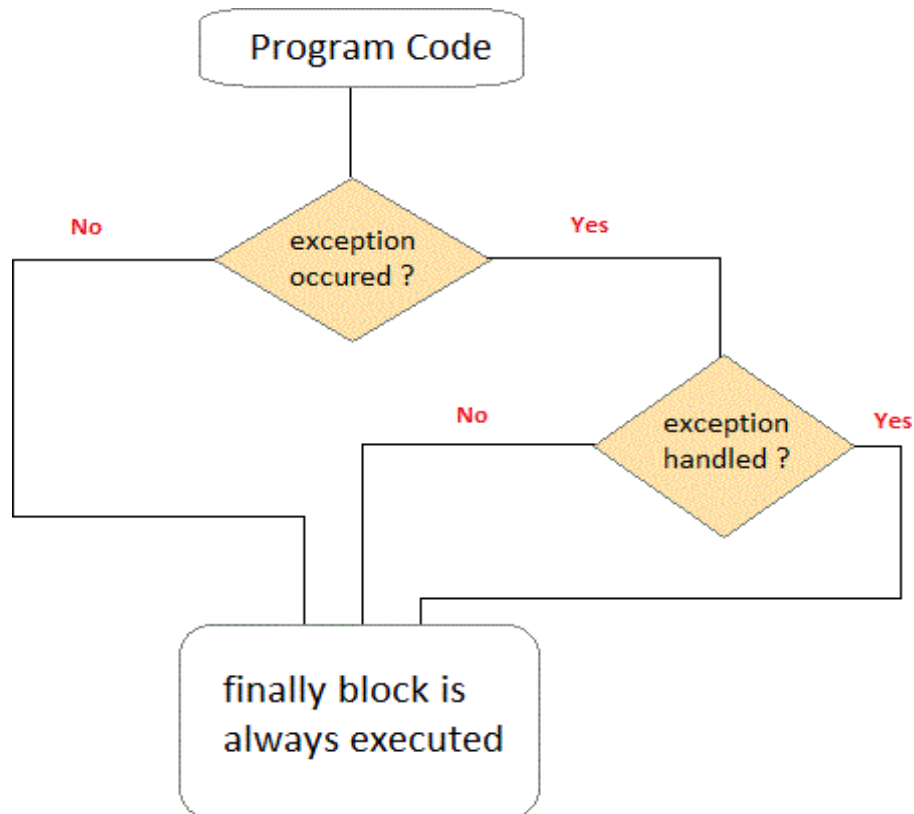
    }

}
```

```
}
```

Finally:

A 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 exception has occurred. Using a finally block, lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

**Example demonstrating finally Clause**

```
class ExceptionTest {  
  
    public static void main(String[] args)  
  
    {  
  
        int a[]= new int[2];  
  
        System.out.println("out of try");  
  
        try {
```

```

System.out.println("Access invalid element"+ a[3]);

/* the above statement will throw ArrayIndexOutOfBoundsException */

}

catch(ArrayIndexOutOfBoundsException ae)

{

System.out.println("array exception");

}

finally {

System.out.println("finally is always executed.");

}

}

}

```

Output :

Out of try

Array exception

finally is always executed.

Exception in thread main java. Lang. exception array Index out of bound exception.

3.1.9 finalize() method

finalize()**Description**

The **java.lang.Object.finalize()** is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

Declaration

Following is the declaration for **java.lang.Object.finalize()** method

```
Protected void finalize()
```

Finalize method does not have Parameters and return value

Exception

- **Throwable** -- the Exception raised by this method

Example on finalize method

```
class FinalizeExample{

public void finalize()

{

System.out.println("finalize called");

}

public static void main(String[] args){

FinalizeExample f1=new FinalizeExample();

FinalizeExample f2=new FinalizeExample();

f1=null;

f2=null;

System.gc();

}

}
```

Output:

finalize called

3.1.10 Built – in Exceptions

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

The most general of these exceptions are subclasses of the standard type `RuntimeException`.

Since `java.lang` is implicitly imported into all Java programs, most exceptions derived from `RuntimeException` are automatically available.

Java defines several other types of exceptions that relate to its various class libraries.

Following is the list of Java Unchecked `RuntimeException`.

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 the current thread state.

IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
StringIndexOutOfBoundsException	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.

3.1.11 User –Defined Exceptions

User defined Exception subclass

You can also create your own exception sub class simply by extending java **Exception** class. You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** function to display your customized message on catch.

```
class MyException extends Exception {
private int ex;
MyException(int a) //constructor
{
ex=a;
}
public String toString()
{
return "MyException[" + ex + "] is less than zero";
}}
class UserException {
static void sum(inta,int b) throws MyException
{
if(a<0) {
throw new MyException(a);
}
else {
System.out.println(a+b);
}
}
public static void main(String[] args)
{
try {
sum(-10, 10);
}
catch(MyException me) {
System.out.println(me);
}
}
}
```

Points to Remember

1. Extend the Exception class to create your own exception class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

COLLECTIONS FRAMEWORK (java.util)

In order to handle group of objects we can use array of objects. If we have a class called Employ with members name and id, if we want to store details of 10 Employees, create an array of object to hold 10 Employ details.

```
Employ ob [] = new Employ [10];
```

- We cannot store different class objects into same array.
- Inserting element at the end of array is easy but at the middle is difficult.
- After retrieving the elements from the array, in order to process the elements we don't have
- any methods

Collections

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

A Collection represents a single unit of objects, i.e., a group.

Collection Object:

- A collection object is an object which can store group of other objects.
- A collection object has a class called Collection class or Container class.
- All the collection classes are available in the package called 'java.util' (util stands for utility).
- Group of collection classes is called a Collection Framework.
- A collection object does not store the physical copies of other objects; it stores references of other objects.
- All the collection classes in java.util package are the implementation classes of different interfaces.

Collection Interface

The Collection interface is the foundation on which Collection Framework is built.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

The collections framework was designed to meet several goals, such as —

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly

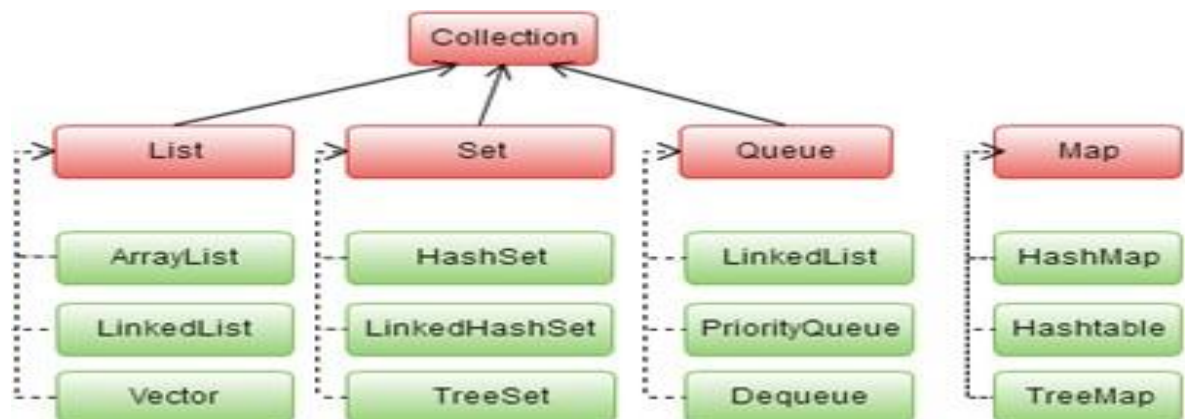
efficient.

- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

A collections framework is a unified architecture for representing and manipulating collections.

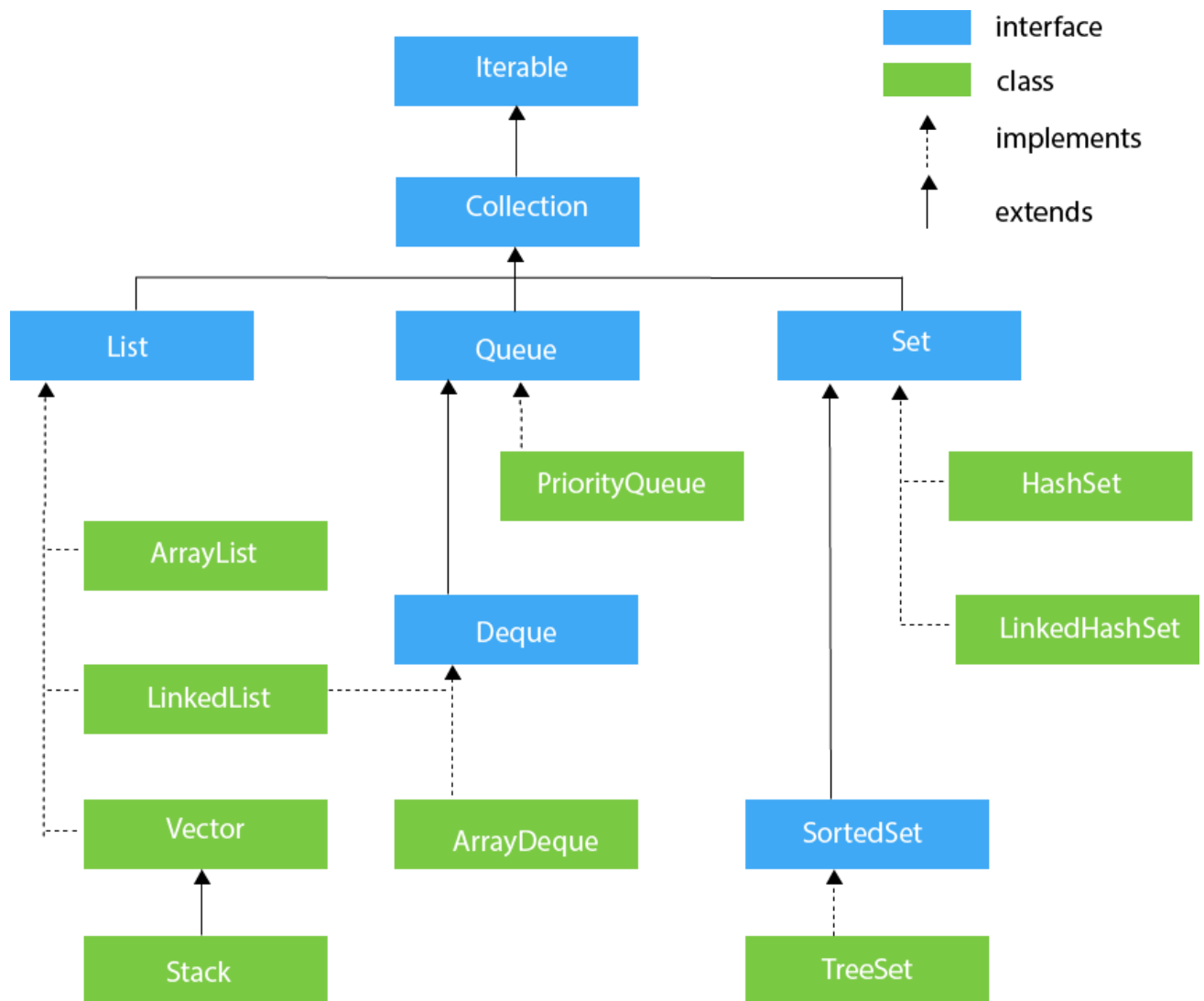
All collections frameworks contain the following –

- **Interfaces** – These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes** – These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms** – These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.



Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



- Collection is an object representing a group of objects.
- Collection framework contains a set of classes and interfaces which are used for representing and manipulating collections.
- Collection interface is the root interface from which the interfaces List, Set, Queue are extended.

List Interface:

- List interface is an ordered collection in which duplicate elements are also allowed.
 - Elements are stored in a sequential way and hence its elements can be accessed using the index value.
1. ArrayList
 2. LinkedList
 3. Vector

Set Interface:

Set is an unordered collection. It does not maintain any order while storing the elements. It does not allow duplicate elements.

Thus if one requires to store a group of unique elements, set can be used

1. HashSet
2. LinkedHashSet
3. TreeSet

Map Interface: Map is an interface which maps keys to values in which each key has to be unique.

1. HashMap
2. Hashtable
3. TreeMap

Collection interface

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. Several of these methods can throw an **UnsupportedOperationException**. These **methods** are summarized in the following.

1. **boolean add(Object obj)** : Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
2. **boolean addAll(Collection c)** : Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.
3. **void clear()** : Removes all elements from the invoking collection.
4. **boolean contains(Object obj)** : Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5. **boolean containsAll(Collection c)** : Returns true if the invoking collection contains all elements of c. Otherwise, returns false.
6. **boolean equals(Object obj)** : Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7. **int hashCode()** : Returns the hash code for the invoking collection.
8. **boolean isEmpty()** : Returns true if the invoking collection is empty. Otherwise, returns false.
9. **Iterator iterator()** : Returns an iterator for the invoking collection.
10. **boolean remove(Object obj)** : Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11. **boolean removeAll(Collection c)** : Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

12. **boolean retainAll(Collection c)** : Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
13. **int size()** : Returns the number of elements held in the invoking collection.
14. **Object[] toArray()** : Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
15. **Object[] toArray(Object array[])** : Returns an array containing only those collection elements whose type matches that of array.

Example

Following is an example to explain few methods from various class implementations of the above collection methods –

```
import java.util.*;
public class CollectionsDemo {

    public static void main(String[] args) {
        // ArrayList
        List a1 = new ArrayList();
        a1.add("Zara");
        a1.add("Mahnaz");
        a1.add("Ayan");
        System.out.println(" ArrayList Elements");
        System.out.print("\t" + a1);

        // LinkedList
        List l1 = new LinkedList();
        l1.add("Zara");
        l1.add("Mahnaz");
        l1.add("Ayan");
        System.out.println();
        System.out.println(" LinkedList Elements");
        System.out.print("\t" + l1);

        // HashSet
        Set s1 = new HashSet();
        s1.add("Zara");
        s1.add("Mahnaz");
        s1.add("Ayan");
        System.out.println();
        System.out.println(" Set Elements");
        System.out.print("\t" + s1);

        // HashMap
        Map m1 = new HashMap();
        m1.put("Zara", "8");
        m1.put("Mahnaz", "31");
        m1.put("Ayan", "12");
        m1.put("Daisy", "14");
        System.out.println();
    }
}
```



```

        System.out.println(" Map Elements");
        System.out.print("\t" + m1);
    }
}

```

OUTPUT

```

ArrayList Elements
    [Zara, Mahnaz, Ayan]
LinkedList Elements
    [Zara, Mahnaz, Ayan]
Set Elements
    [Ayan, Zara, Mahnaz]
Map Elements
    {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

```

List interface

The List interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.

- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- In addition to the methods defined by **Collection**, List defines some of its own, which are summarized in the following table.
- Several of the list methods will throw an `UnsupportedOperationException` if the collection cannot be modified, and a `ClassCastException` is generated when one object is incompatible with another.

Methods of List are :

1. **Object get(int index)** : Returns the object stored at the specified index within the invoking collection.
2. **int indexOf(Object obj)** : Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
3. **int lastIndexOf(Object obj)** : Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
4. **ListIterator listIterator()** : Returns an iterator to the start of the invoking list.
5. **ListIterator listIterator(int index)** : Returns an iterator to the invoking list that begins at the specified index.
6. **Object remove(int index)** : Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
7. **Object set(int index, Object obj)** : Assigns obj to the location specified by index within the invoking list.
8. **List subList(int start, int end)** : Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Example

The above interface has been implemented in various classes like ArrayList or LinkedList, etc. Following is the example to explain few methods from various class implementation of the above collection methods

```
import java.util.*;
public class CollectionsDemo
{
    public static void main(String[] args)
    {
        List a1 = new ArrayList();
        a1.add("Zara");
        a1.add("Mahnaz");
        a1.add("Ayan");
        System.out.println(" ArrayList Elements");
        System.out.print("\t" + a1);

        List l1 = new LinkedList();
        l1.add("Zara");
        l1.add("Mahnaz");
        l1.add("Ayan");
        System.out.println();
        System.out.println(" LinkedList Elements");
        System.out.print("\t" + l1);
    }
}
```

OUTPUT

```
ArrayList Elements
    [Zara, Mahnaz, Ayan]
LinkedList Elements
    [Zara, Mahnaz, Ayan]
```

Set

A Set is a Collection that **cannot contain duplicate** elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

The methods declared by Set are

1. **add()** : Adds an object to the collection.
2. **clear()** : Removes all objects from the collection.
3. **contains()** : Returns true if a specified object is an element within the collection.
4. **isEmpty()** : Returns true if the collection has no elements.
5. **iterator()** : Returns an Iterator object for the collection, which may be used to retrieve an object.
6. **remove()** : Removes a specified object from the collection.
7. **size()** : Returns the number of elements in the collection.

Example

Set has its implementation in various classes like HashSet, TreeSet, LinkedHashSet. Following

is an example to explain Set functionality –

```
import java.util.*;
public class SetDemo
{
    public static void main(String args[])
    {
        int count[] = {34, 22,10,60,30,22};
        Set<Integer> set = new HashSet<Integer>();
        try {
            for(int i = 0; i < 5; i++)
            {
                set.add(count[i]);
            }
            System.out.println(set);
            TreeSet sortedSet = new TreeSet<Integer>(set);
            System.out.println("The sorted list is:");
            System.out.println(sortedSet);
            System.out.println("The First element of the set is: " + (Integer)sortedSet.first());
            System.out.println("The last element of the set is: " + (Integer)sortedSet.last());
        }
        catch(Exception e) {}
    }
}
```

OUTPUT:

[34, 22, 10, 60, 30]

The sorted list is:

[10, 22, 30, 34, 60]

The First element of the set is: 10

The last element of the set is: 60

SortedSet interface

- The SortedSet interface extends Set and declares the behavior of a set sorted in an ascending order.
- Several methods throw a NoSuchElementException when no items are contained in the invoking set.
- A ClassCastException is thrown when an object is incompatible with the elements in a set.
- A NullPointerException is thrown if an attempt is made to use a null object and **null is not allowed in the set**.

In addition to those methods defined by Set, the SortedSet interface declares the methods summarized as following -

1. **Comparator comparator()** : Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
2. **Object first()** : Returns the first element in the invoking sorted set.

3. **SortedSet headSet(Object end)** : Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
4. **Object last()** : Returns the last element in the invoking sorted set.
5. **SortedSet subSet(Object start, Object end)** : Returns a SortedSet that includes those elements between start and end. Elements in the returned collection are also referenced by the invoking object.
6. **SortedSet tailSet(Object start)** : Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Example

SortedSet have its implementation in various classes like TreeSet. Following is an example of a TreeSet class –

```
import java.util.*;
public class SortedSetTest {

    public static void main(String[] args) {
        // Create the sorted set
        SortedSet set = new TreeSet();

        // Add elements to the set
        set.add("b");
        set.add("c");
        set.add("a");

        // Iterating over the elements in the set
        Iterator it = set.iterator();

        while (it.hasNext()) {
            // Get element
            Object element = it.next();
            System.out.println(element.toString());
        }
    }
}
```

Output

```
a
b
c
```

Map interface

The **Map interface maps unique keys to values**. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

- Several methods throw a `NoSuchElementException` when no items exist in the invoking map.
- A `ClassCastException` is thrown when an object is incompatible with the elements in a map.
- A `NullPointerException` is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An `UnsupportedOperationException` is thrown when an attempt is made to change an unmodifiable map.

The methods declared by Map are :

1. **`void clear()`** : Removes all key/value pairs from the invoking map.
2. **`boolean containsKey(Object k)`** : Returns true if the invoking map contains **k** as a key. Otherwise, returns false.
3. **`boolean containsValue(Object v)`** : Returns true if the map contains **v** as a value. Otherwise, returns false.
4. **`Set entrySet()`** : Returns a Set that contains the entries in the map. The set contains objects of type `Map.Entry`. This method provides a set-view of the invoking map.
5. **`boolean equals(Object obj)`** : Returns true if `obj` is a Map and contains the same entries. Otherwise, returns false.
6. **`Object get(Object k)`** : Returns the value associated with the key **k**.
7. **`int hashCode()`** : Returns the hash code for the invoking map.
8. **`boolean isEmpty()`** : Returns true if the invoking map is empty. Otherwise, returns false.
9. **`Set keySet()`** : Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10. **`Object put(Object k, Object v)`** : Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are **k** and **v**, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11. **`void putAll(Map m)`** : Puts all the entries from **m** into this map.
12. **`Object remove(Object k)`** : Removes the entry whose key equals **k**.
13. **`int size()`** : Returns the number of key/value pairs in the map.
14. **`Collection values()`** : Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Example : Map has its implementation in various classes like `HashMap`. Following is an example to explain map functionality –

```
import java.util.*;
public class CollectionsDemo {

    public static void main(String[] args) {
        Map m1 = new HashMap();
        m1.put("Zara", "8");
```

```

m1.put("Mahnaz", "31");
m1.put("Ayan", "12");
m1.put("Daisy", "14");

System.out.println();
System.out.println(" Map Elements");
System.out.print("\t" + m1);
}
}

```

Output:

```

Map Elements
    {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}
Map Elements
    {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

```

The Collection Classes

- Java provides a set of standard collection classes that implement Collection interfaces.
- Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

ArrayList class

- The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.
- Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

constructors provided by the ArrayList class are:

1. **ArrayList()**: This constructor builds an empty array list.
2. **ArrayList(Collection c)**: This constructor builds an array list that is initialized with the elements of the collection **c**.
3. **ArrayList(int capacity)**: This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

ArrayList defines the following methods –

1. **void add(int index, Object element)**: Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2. **boolean add(Object o)**: Appends the specified element to the end of this list.

3. **boolean addAll(Collection c)** : Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws `NullPointerException`, if the specified collection is null.
4. **boolean addAll(int index, Collection c)** : Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws `NullPointerException` if the specified collection is null.
5. **void clear()** : Removes all of the elements from this list.
6. **Object clone()** : Returns a shallow copy of this `ArrayList`.
7. **boolean contains(Object o)** : Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.
8. **void ensureCapacity(int minCapacity)** : Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
9. **Object get(int index)** : Returns the element at the specified position in this list. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).
10. **int indexOf(Object o)** : Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
11. **int lastIndexOf(Object o)** : Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
12. **Object remove(int index)** : Removes the element at the specified position in this list. Throws `IndexOutOfBoundsException` if the index out is of range (`index < 0 || index >= size()`).
13. **protected void removeRange(int fromIndex, int toIndex)** : Removes from this List all of the elements whose index is between `fromIndex`, inclusive and `toIndex`, exclusive.
14. **Object set(int index, Object element)** : Replaces the element at the specified position in this list with the specified element. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).
15. **int size()** : Returns the number of elements in this list.
16. **Object[] toArray()** : Returns an array containing all of the elements in this list in the correct order. Throws `NullPointerException` if the specified array is null.
17. **Object[] toArray(Object[] a)** : Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
18. **void trimToSize()** : Trims the capacity of this `ArrayList` instance to be the list's current size.

Example

// Java program to demonstrate ArrayList class –

```
import java.util.*;
public class ArrayListDemo {

    public static void main(String args[]) {
        // create an array list
        ArrayList al = new ArrayList();
        System.out.println("Initial size of al: " + al.size());
    }
}
```

```

// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " + al.size());

// display the array list
System.out.println("Contents of al: " + al);

// Remove elements from the array list
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " + al.size());
System.out.println("Contents of al: " + al);
}
}

```

Output

Initial size of al: 0

Size of al after additions: 7

Contents of al: [C, A2, A, E, B, D, F]

Size of al after deletions: 5

Contents of al: [C, A2, E, B, D]

LinkedList Class

- The LinkedList class extends AbstractSequentialList and implements the List interface.
- It provides a linked-list data structure.

Constructors supported by the **LinkedList** class are

1. **LinkedList()** : This constructor builds an empty linked list.
2. **LinkedList(Collection c)** : This constructor builds a linked list that is initialized with the elements of the collection **c**.

Apart from the methods inherited from its parent classes, **LinkedList** defines following methods –

1. **void add(int index, Object element)** : Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2. **boolean add(Object o)** : Appends the specified element to the end of this list.
3. **boolean addAll(Collection c)** : Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.
4. **boolean addAll(int index, Collection c)** : Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.

5. **void addFirst(Object o)** : Inserts the given element at the beginning of this list.
6. **void addLast(Object o)** : Appends the given element to the end of this list.
7. **void clear()** : Removes all of the elements from this list.
8. **Object clone()** : Returns a shallow copy of this LinkedList.
9. **boolean contains(Object o)** : Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
10. **Object get(int index)** : Returns the element at the specified position in this list. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).
11. **Object getFirst()** : Returns the first element in this list. Throws `NoSuchElementException` if this list is empty.
12. **Object getLast()** : Returns the last element in this list. Throws `NoSuchElementException` if this list is empty.
13. **int indexOf(Object o)** : Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element.
14. **int lastIndexOf(Object o)** : Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
15. **ListIterator listIterator(int index)** : Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).
16. **Object remove(int index)** : Removes the element at the specified position in this list. Throws `NoSuchElementException` if this list is empty.
17. **boolean remove(Object o)** : Removes the first occurrence of the specified element in this list. Throws `NoSuchElementException` if this list is empty. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).
18. **Object removeFirst()** : Removes and returns the first element from this list. Throws `NoSuchElementException` if this list is empty.
19. **Object removeLast()** : Removes and returns the last element from this list. Throws `NoSuchElementException` if this list is empty.
20. **Object set(int index, Object element)** : Replaces the element at the specified position in this list with the specified element. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).
21. **int size()** : Returns the number of elements in this list.
22. **Object[] toArray()** : Returns an array containing all of the elements in this list in the correct order. Throws `NullPointerException` if the specified array is null.
23. **Object[] toArray(Object[] a)** : Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Example

// Java Program to demonstrate LinkedList class –

```
import java.util.*;
public class LinkedListDemo {

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

```

// create a linked list
LinkedList ll = new LinkedList();

// add elements to the linked list
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);

// remove elements from the linked list
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: " + ll);

// remove first and last elements
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: " + ll);

// get and set a value
Object val = ll.get(2);
ll.set(2, (String) val + " Changed");
System.out.println("ll after change: " + ll);
}
}

```

Output

Original contents of ll: [A, A2, F, B, D, E, C, Z]

Contents of ll after deletion: [A, A2, D, E, C, Z]

ll after deleting first and last: [A2, D, E, C]

ll after change: [A2, D, E Changed, C]

The HashSet Class

- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.
- A hash table stores information by using a mechanism called **hashing**. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
- The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Following is the list of **constructors** provided by the HashSet class.

1. **HashSet()** : This constructor constructs a default HashSet.
2. **HashSet(Collection c)** : This constructor initializes the hash set by using the elements

of the collection **c**.

3. **HashSet(int capacity)** : This constructor initializes the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4. **HashSet(int capacity, float fillRatio)** : This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

Apart from the methods inherited from its parent classes, HashSet defines following methods–

1. **boolean add(Object o)** : Adds the specified element to this set if it is not already present.
2. **void clear()** : Removes all of the elements from this set.
3. **Object clone()** : Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
4. **boolean contains(Object o)** : Returns true if this set contains the specified element.
5. **boolean isEmpty()** : Returns true if this set contains no elements.
6. **Iterator iterator()** : Returns an iterator over the elements in this set.
7. **boolean remove(Object o)** : Removes the specified element from this set if it is present.
8. **int size()** : Returns the number of elements in this set (its cardinality).

Example

// Java program to demonstrate HashSet class –

```
import java.util.*;
public class HashSetDemo {

    public static void main(String args[]) {
        // create a hash set
        HashSet hs = new HashSet();

        // add elements to the hash set
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");
        System.out.println(hs);
    }
}
```

Output

[A, B, C, D, E, F]

TreeSet Class

- TreeSet provides an implementation of the Set interface that uses a tree for storage.

Objects are stored in a sorted and ascending order.

- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

The Constructors Supported By The TreeSet Class

1. **TreeSet()** : This constructor constructs an empty tree set that will be sorted in an ascending order according to the natural order of its elements.
2. **TreeSet(Collection c)** : This constructor builds a tree set that contains the elements of the collection **c**.
3. **TreeSet(Comparator comp)** : This constructor constructs an empty tree set that will be sorted according to the given comparator.
4. **TreeSet(SortedSet ss)** : This constructor builds a TreeSet that contains the elements of the given SortedSet.

Example

The methods supported by this collection are–

// Java program to demonstrate TreeSet class –

```
import java.util.*;

public class TreeSetDemo {

    public static void main(String args[]) {
        // Create a tree set
        TreeSet ts = new TreeSet();

        // Add elements to the tree set
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
        System.out.println(ts);
    }
}
```

Output

[A, B, C, D, E, F]

PriorityQueue class

- A PriorityQueue is used when the objects are supposed to be processed based on the priority.
- It is known that a queue follows First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play.
- The PriorityQueue is based on the priority heap.

- The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- PriorityQueue doesn't permit null.
- We can't create PriorityQueue of Objects that are non-comparable
- PriorityQueue are unbound queues.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

Constructors of PriorityQueue class

1. **PriorityQueue():** Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
2. **PriorityQueue(Collection<E> c):** Creates a PriorityQueue containing the elements in the specified collection.
3. **PriorityQueue(int initialCapacity):** Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
4. **PriorityQueue(int initialCapacity, Comparator<E> comparator):** Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
5. **PriorityQueue(PriorityQueue<E> c):** Creates a PriorityQueue containing the elements in the specified priority queue.
6. **PriorityQueue(SortedSet<E> c):** Creates a PriorityQueue containing the elements in the specified sorted set.

Methods of PriorityQueue are:

1. **boolean add(E element):** This method inserts the specified element into this priority queue.
2. **public peek():** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
3. **public poll():** This method retrieves and removes the head of this queue, or returns null if this queue is empty.

// Java program to demonstrate working of PriorityQueue in Java

```
import java.util.*;
class GfG {
    public static void main(String args[])
    {
        // Creating empty priority queue
        PriorityQueue pQueue = new PriorityQueue();
```

```

// Adding items to the pQueue using add()
pQueue.add(10);
pQueue.add(20);
pQueue.add(15);
// Printing the top element of PriorityQueue
System.out.println(pQueue.peek());

// Printing the top element and removing it
// from the PriorityQueue container
System.out.println(pQueue.poll());

// Printing the top element again
System.out.println(pQueue.peek());
}
}

```

Output:

10
10
15

ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

Constructors in ArrayDeque:

1. **ArrayDeque():** Used to create an empty ArrayDeque and by default holds an initial capacity to hold 16 elements.
2. **ArrayDeque(Collection c):** Used to create an ArrayDeque containing all the elements same as that of the specified collection.
3. **ArrayDeque(int numofElements):** Used to create an empty ArrayDeque and holds the capacity to contain a specified number of elements.

// Java program to demonstrate few functions of ArrayDeque in Java

```

import java.util.*;
public class ArrayDequeExample
{
    public static void main(String[] args)
    {
        //Creating Deque and adding elements
        Deque<String> deque = new ArrayDeque<String>();

```

```

deque.add("Ravi");
deque.add("Vijay");
deque.add("Ajay");
//Traversing elements
for (String str : deque)
{
    System.out.println(str);
}
}

```

Output:

```

Ravi
Vijay
Ajay

```

Accessing a Collection via an Iterator.

Iterator Interface

- Iterator is an interface that contains methods to retrieve the elements one by one from a collection object.
- It retrieves elements only in **forward direction**.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

1. **boolean hasNext()** : It returns true if the iterator has more elements otherwise it returns false.
2. **Object next()**:It returns the element and moves the cursor pointer to the next element.
3. **void remove()**:It removes the last elements returned by the iterator. It is less used.

For-Each alternative

For-each is another array traversing technique like for loop, while loop, do-while loop.

- It starts with the keyword **for** like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.
- It's commonly used to iterate over an array or a Collections class (eg, ArrayList).

Syntax:

```
for (type var : array)
```

```

{
    statements using var;
}

```

is equivalent to:

```

for (int i=0; i<arr.length; i++)
{
    type var = arr[i];
    statements using var;
}

```

// Java program to illustrate for-each loop

```

class For_Each
{
    public static void main(String[] arg)
    {
        {
            int[] marks = { 125, 132, 95, 116, 110 };

            int highest_marks = maximum(marks);
            System.out.println("The highest score is " + highest_marks);
        }
    }
    public static int maximum(int[] numbers)
    {
        int maxSoFar = numbers[0];

        // for each loop
        for (int num : numbers)
        {
            if (num > maxSoFar)
            {
                maxSoFar = num;
            }
        }
        return maxSoFar;
    }
}

```

Output:

The highest score is 132

Stack Class

- A stack represents a group of elements stored in LIFO (Last In First Out) order.
- This means that the element which is stored as a last element into the stack will be the first element to be removed from the stack.

- Inserting the elements (Objects) into the stack is called push operation and removing the elements from stack is called pop operation.
- Searching for an element in stack is called peep operation.
- Insertion and deletion of elements take place only from one side of the stack, called top of the stack.

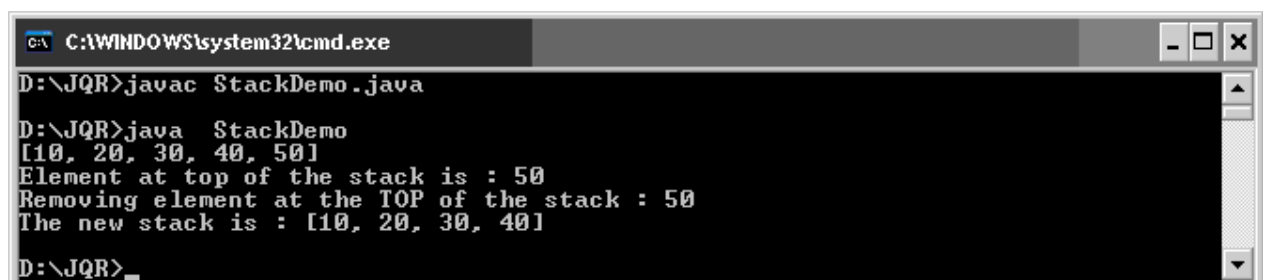
Stack Class Methods:

1. **boolean empty()** : this method tests whether the stack is empty or not. If the stack is empty then true is returned otherwise false.
2. **element peek()** : This method returns the top most object from the stack without removing it.
3. **element pop()** : This method pops the top-most element from the stack and returns it.
4. **element push(element obj)** : This method pushes an element obj onto the top of the stack and returns that element.
5. **int search(Object obj)** : This method returns the position of an element obj from the top of the stack. If the element (object) is not found in the stack then it returns -1.

// Program to perform different operations on a stack.

```
import java.util.*;
class StackDemo
{
    public static void main(String args[])
    {
        //create an empty stack to contain Integer objects
        Stack<Integer> st = new Stack<Integer>();
        st.push (new Integer(10) );
        st.push (new Integer(20) );
        st.push (new Integer(30) );
        st.push (new Integer(40) );
        st.push (new Integer(50) );
        System.out.println (st);
        System.out.println ("Element at top of the stack is : " + st.peek() );
        System.out.println ("Removing element at the TOP of the stack : " + st.pop());
        System.out.println ("The new stack is : " + st);
    }
}
```

OUTPUT :



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac StackDemo.java
D:\JQR>java StackDemo
[10, 20, 30, 40, 50]
Element at top of the stack is : 50
Removing element at the TOP of the stack : 50
The new stack is : [10, 20, 30, 40]
D:\JQR>
```

Vector Class

- Similar to ArrayList, but Vector is synchronized.
- It means even if several threads act on Vector object simultaneously, the results will be reliable.

Vector Class Methods:

1. **boolean add(element obj)** : This method appends the specified element to the end of the Vector.
2. If the element is added successfully then the method returns true.
3. **void add (int position,element obj)** : This method inserts the specified element at the specified position in the Vector.
4. **element remove (int position)** : This method removes the element at the specified position in the Vector and returns it.
5. **boolean remove (Object obj)** : This method removes the first occurrence of the specified element obj from the Vector, if it is present.
6. **void clear ()** This method removes all the elements from the Vector.
7. **element set (int position, element obj)** : This method replaces an element at the specified position in the Vector with the specified element obj.
8. **boolean contains (Object obj)** : This method returns true if the Vector contains the specified element obj.
9. **element get (int position)** : This method returns the element available at the specified position in the Vector.
10. **int size ()** : Returns number of elements in the Vector.
11. **int indexOf (Object obj)** : This method returns the index of the first occurrence of the specified element in the Vector, or -1 if the Vector does not contain the element.
12. **int lastIndexOf (Object obj)** : This method returns the index of the last occurrence of the specified element in the Vector, or -1 if the Vector does not contain the element.
13. **Object[] toArray ()** : This method converts the Vector into an array of Object class type. All the elements of the Vector will be stored into the array in the same sequence.
14. **int capacity ()** : This method returns the current capacity of the Vector.

// Program that shows the use of Vector class.

```
import java.util.*;
class VectorDemo
{
    public static void main(String args[])
    {
```

```

Vector <Integer> v = new Vector<Integer> ();
int x[] = {10,20,30,40,50};
//When x[i] is stored into v below, x[i] values are converted into Integer Objects
//and stored into v. This is auto boxing.
for (int i = 0; i<x.length; i++)
    v.add(x[i]);
System.out.println ("Getting Vector elements using get () method: ");
for (int i = 0; i<v.size(); i++)
    System.out.print (v.get (i) + "\t");
System.out.println ("\nRetrieving elements in Vector using ListIterator :");
ListIterator lit = v.listIterator ();
while (lit.hasNext () )
    System.out.print (lit.next () + "\t");
System.out.println ("\nRetrieving elements in reverse order using ListIterator :");
while (lit.hasPrevious () )
    System.out.print (lit.previous () + "\t");
    }
}

```

More Utility classes, String Tokenizer, Date,Random,Scanner.

StringTokenizer class

- The StringTokenizer class is useful to break a String into small pieces called tokens.
- We can create an object to StringTokenizer as:

```
StringTokenizer st = new StringTokenizer (str, "delimiter");
```

StringTokenizer Class Methods:

1. **String nextToken()** : Returns the next token from the StringTokenizer
2. **boolean hasMoreTokens()** : Returns true if token is available and returns false if not available
3. **int countTokens()** : Returns the number of tokens available.

// Program that shows the use of Vector class.

```

import java.util.*;
class STDemo
{
    public static void main(String args[])
    { //take a String
        String str = "Java is an OOP Language";
    }
}

```

```

//brake wherever a space is found
StringTokenizer st = new StringTokenizer (str," ");
//retrieve tokens and display
System.out.println ("The tokens are: ");
while ( st.hasMoreTokens () )
{
    String s = st.nextToken ();
    System.out.println (s );
}
}
}

```

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac STDemo.java
D:\JQR>java STDemo
The tokens are:
Java
is
an
OOP
Language
D:\JQR>

```

Date Class

- Date Class is also useful to handle date and time.
- Once Date class object is created, it should be formatted using the following methods of DateFormat class of java.text package.
- We can create an object to Date class as: Date dd = new Date ();
- Once Date class object is created, it should be formatted using the methods of DateFormat class of java.text package.

DateFormat class Methods :

1. **DateFormat fmt = DateFormat.getDateInstance(formatconst, region)** : This method is useful to store format information for date value into DateFormat object fmt.
2. **DateFormat fmt = DateFormat.getTimeInstance(formatconst, region)** : This method is useful to store format information for time value into DateFormat object fmt.
3. **DateFormat fmt = DateFormat.getDateTimeInstance(formatconst, formatconst, region)** : This method is useful to store format information for date value into DateFormat object fmt.

Formatconst

DateFormat.FULL

DateFormat.LONG

DateFormat.MEDIUM

Example (region=Locale.UK)

03 september 2007 19:43:14 O'Clock GMT + 05:30

03 september 2007 19:43:14 GMT + 05:30

03-sep-07 19:43:14

// Java program to display System time and date.

```
import java.util.*;
import java.text.*;
class MyDate
{
    public static void main(String args[])
    {
        Date d = new Date ();
        DateFormat fmt = DateFormat.getDateInstance (DateFormat.MEDIUM,
        DateFormat.SHORT, Locale.UK);
        String str = fmt.format (d);
        System.out.println (str);
    }
}
```

OUTPUT:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac MyDate.java
D:\JQR>java MyDate
09-Oct-2011 16:45
D:\JQR>
```

Random class

- The **java.util.Random** class instance is used to generate a stream of pseudorandom numbers.
- The class uses a 48-bit seed, which is modified using a linear congruential formula.
- The algorithms implemented by class Random use a protected utility method that on each invocation can supply up to 32 pseudorandomly generated bits.
- Following is the declaration for **java.util.Random** class –

```
public class Random extends Object implements Serializable
```

doubles() : Returns an unlimited stream of pseudorandom double values.

ints() : Returns an unlimited stream of pseudorandom int values.

longs() : Returns an unlimited stream of pseudorandom long values.

next() : Generates the next pseudorandom number.

nextBoolean() : Returns the next uniformly distributed pseudorandom boolean value from the random number generator's sequence.

nextByte() : Generates random bytes and puts them into a specified byte array.

[nextDouble\(\)](#) : Returns the next pseudorandom Double value between 0.0 and 1.0 from the random number generator's sequence

[nextFloat\(\)](#) : Returns the next uniformly distributed pseudorandom Float value between 0.0 and 1.0 from this random number generator's sequence

[nextGaussian\(\)](#) : Returns the next pseudorandom Gaussian double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.

[nextInt\(\)](#) : Returns a uniformly distributed pseudorandom int value generated from this random number generator's sequence

[nextLong\(\)](#) : Returns the next uniformly distributed pseudorandom long value from the random number generator's sequence.

[setSeed\(\)](#) : Sets the seed of this random number generator using a single long seed.

// Java Program to demonstrate Random class.

```
import java.util.Random;
public class JavaRandomExample2
{
    public static void main(String[] args)
    {
        Random random = new Random();
        //return the next pseudorandom integer value
        System.out.println("Random Integer value : "+random.nextInt());
        // setting seed
        long seed =20;
        random.setSeed(seed);
        //value after setting seed
        System.out.println("Seed value : "+random.nextInt());
        //return the next pseudorandom long value
        Long val = random.nextLong();
        System.out.println("Random Long value : "+val);
    }
}
```

OUTPUT :

Random Integer value : 1294094433

Seed value : -1150867590

Random Long value : -7322354119883315205

Scanner Class

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program.

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

// Example program for Scanner class

```
import java.util.*;
public class ScannerDemo
{
    public static void main(String args[])
    {
        System.out.println("-----Enter Your Details----- ");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");String name
        = in.next(); System.out.println("Name: " + name);
        System.out.print("Enter your age: "); int i =
        in.nextInt(); System.out.println("Age: " + i);
        System.out.print("Enter your salary: ");double d =
        in.nextDouble(); System.out.println("Salary: " +
        d); in.close();
    }
}
```

OUTPUT

```
-----Enter Your Details-----
Enter your name: AbhishekName: Abhishek
Enter your age: 23Age: 23
Enter your salary: 25000Salary: 25000.0
```

Revision

4 Interfaces

- List -> Stack , LinkedList, ArrayList, Vector
- Set -> HashSet , LinkedHashSet
- Map->HashMap , Hashtable
- Queue -> LinkedList <T>

4 Interfaces

- Sets: A set represents a group of elements arranged just like an array. The set will grow dynamically

when the elements are stored into it.

A set will not allow duplicate elements. If we try to pass for same element that is already available in the set, then it is not stored into the set.

- List: Lists are like sets.

They store a group of elements. But lists allow duplicate values to be stored.

- Queues: A Queue represents arrangement of elements in FIFO (First In First Out) order. This means that an element that is stored as a first element into the queue will be removed first from the queue.

- Maps: Maps store elements in the form of key and value pairs. If the key is provided then it's correspond value can be obtained. Of course, the keys should have unique values.

Cursor objects are retrieving objects •

Enumeration

- Iterator
- ListIterator
- For-each loop

Using collections various operations can be performed on the data like

- searching,
- sorting,
- insertion,
- manipulation,
- deletion etc

3.2 MultiThreading

Definition of Process

An executing program itself is called a process. In a process-based multitasking, more than two processes can run simultaneously on the computer. A process can contain multiple threads, where every thread is responding different request of the user.

Process are also called heavyweight task. Process-based multitasking leads to more overhead. Inter-process communication is very expensive is also limited. Switching from one process to also expensive. Java does not control the process based multitasking

Definition of Thread

A thread is a part of a process. A process can contain multiple threads. These multiple threads in a process share the same address space of the process. Each thread has its own stack and register to operate on.

<i>BASIS FOR COMPARISON</i>	PROCESS	THREAD
<i>Basic</i>	An executing program is called a process.	A thread is a small part of a process.
<i>Address Space</i>	Every process has its separate address space.	All the threads of a process share the same address space cooperatively as that of a process.
<i>Multitasking</i>	Process-based multitasking allows a computer to run two or more than two programs concurrently.	Thread-based multitasking allows a single program to run two or more threads concurrently.
<i>Communication</i>	Communication between two processes is expensive and limited.	Communication between two threads is less expensive as compared to process.
<i>Switching</i>	Context switching from one process to another process is expensive.	Context switching from one thread to another thread is less expensive as compared to process.
<i>Components</i>	A process has its own address space,	A thread has its own register, state,

<i>BASIS FOR COMPARISON</i>	PROCESS	THREAD
	global variables, signal handlers, open files, child processes, accounting information.	stack, program counter.
<i>Substitute</i>	Process are also called heavyweight task.	Thread are also called lightweight task.
<i>Control</i>	Process-based multitasking is not under the control of Java.	Thread-based multitasking is under the control of Java.
<i>Example</i>	You are working on text editor it refers to the execution of a process.	You are printing a file from text editor while working on it that resembles the execution of a thread in the process.

Multithreading:

Multithreading is a process of executing multiple threads simultaneously. Thread is basically a lightweight subprocess, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so save memory, and context-switching between the threads takes less time than processes.

Multitasking:

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. The processor time is divided among the tasks that are executed

Example : Windows

Context Switching: The process of loading and unloading the process into the memory.

Time Slicing: The amount of processor time that is given to a particular task for execution.

Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Execution of different processes simultaneously
- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

Ex: painting, printing, listening to music

2) Thread-based Multitasking (Multithreading)

- Executing the different parts(dependent or independent) of the same process simultaneously
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Example : games, webapplications

Note:At a time only one thread is executed.

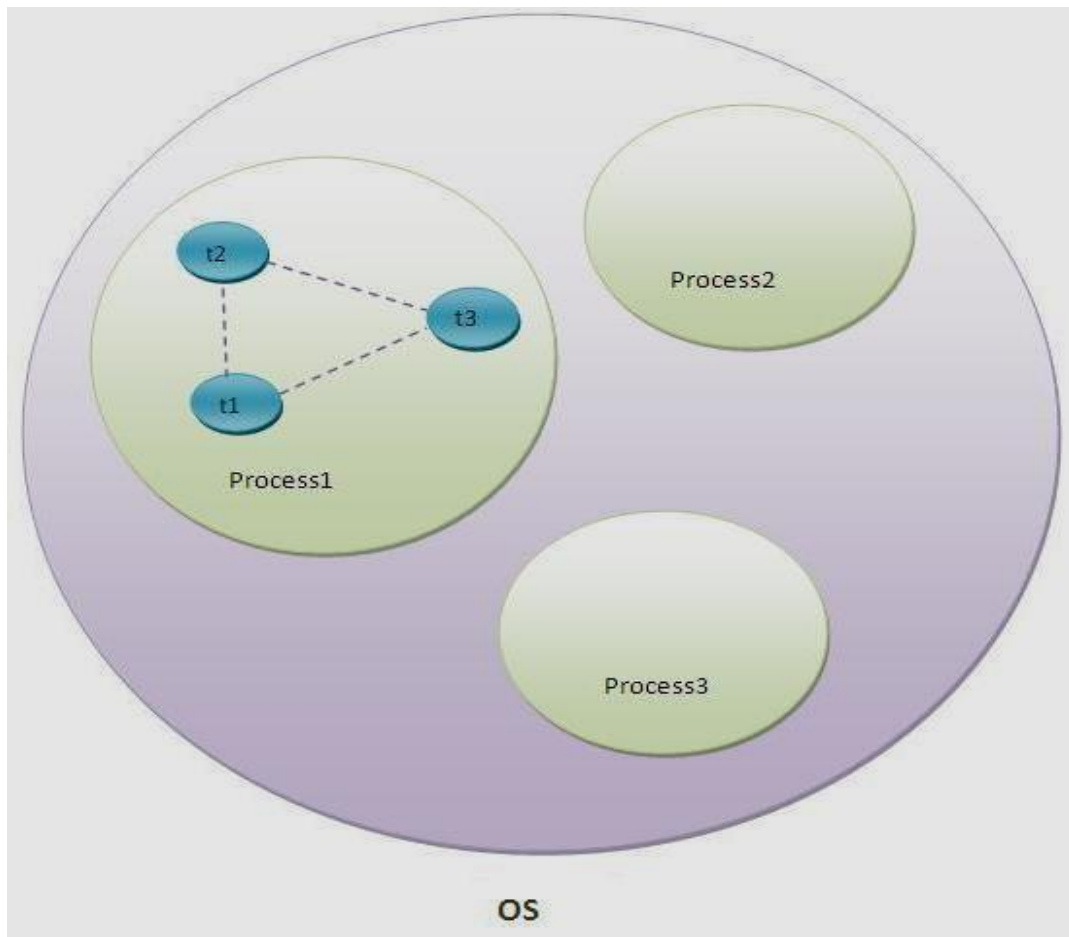
Note:At least one process is required for each thread.

Process Based Multi-tasking	Thread Based multi-tasking
A process is essence of program that is executing which running parallel	Thread is a such part of multithreaded program which defines separate path for execution
It allow us to run java compiler and text editor at a same time	It does not support java compiler and text editor run simultaneously
Process multitasking has more overhead than thread multitasking	Thread multitasking has less overhead than process multitasking
It is unable to gain access over idle time of CPU	It allow taking gain access over idle time taken by CPU
It is not under control of java	It is totally under control of java
It is heavyweight process comparing to thread based multitasking	It is a lightweight process
Inter process communication is expensive and limited	Inter thread communication is inexpensive and context switching from one thread to other is easy
It has slower data rate multitasking	It has faster data rate multithreading

3.2.2 Java Thread model

Thread :

A thread is a part of a process. A process can contain multiple threads. These multiple threads in a process share the same address space of the process. Each thread has its own stack and register to operate on.

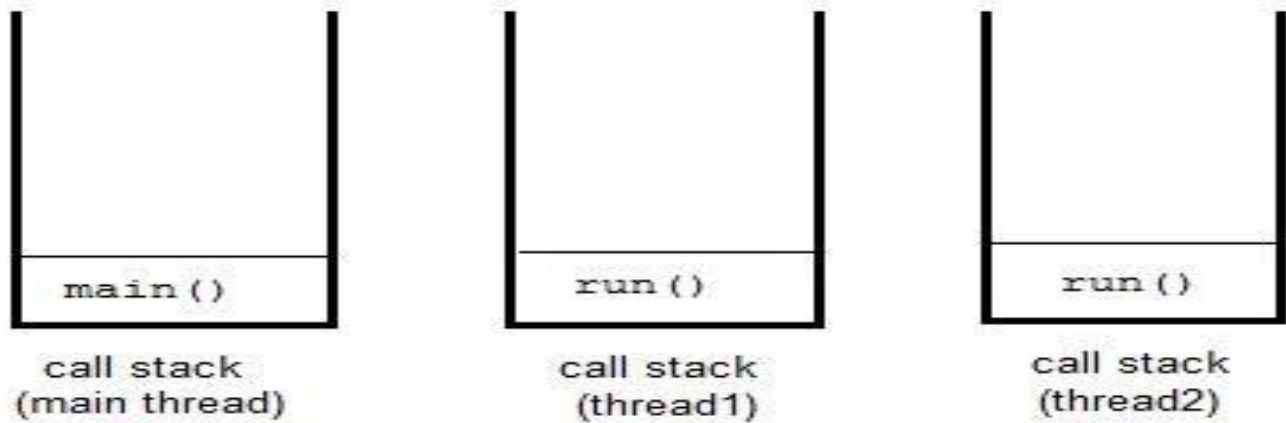


There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

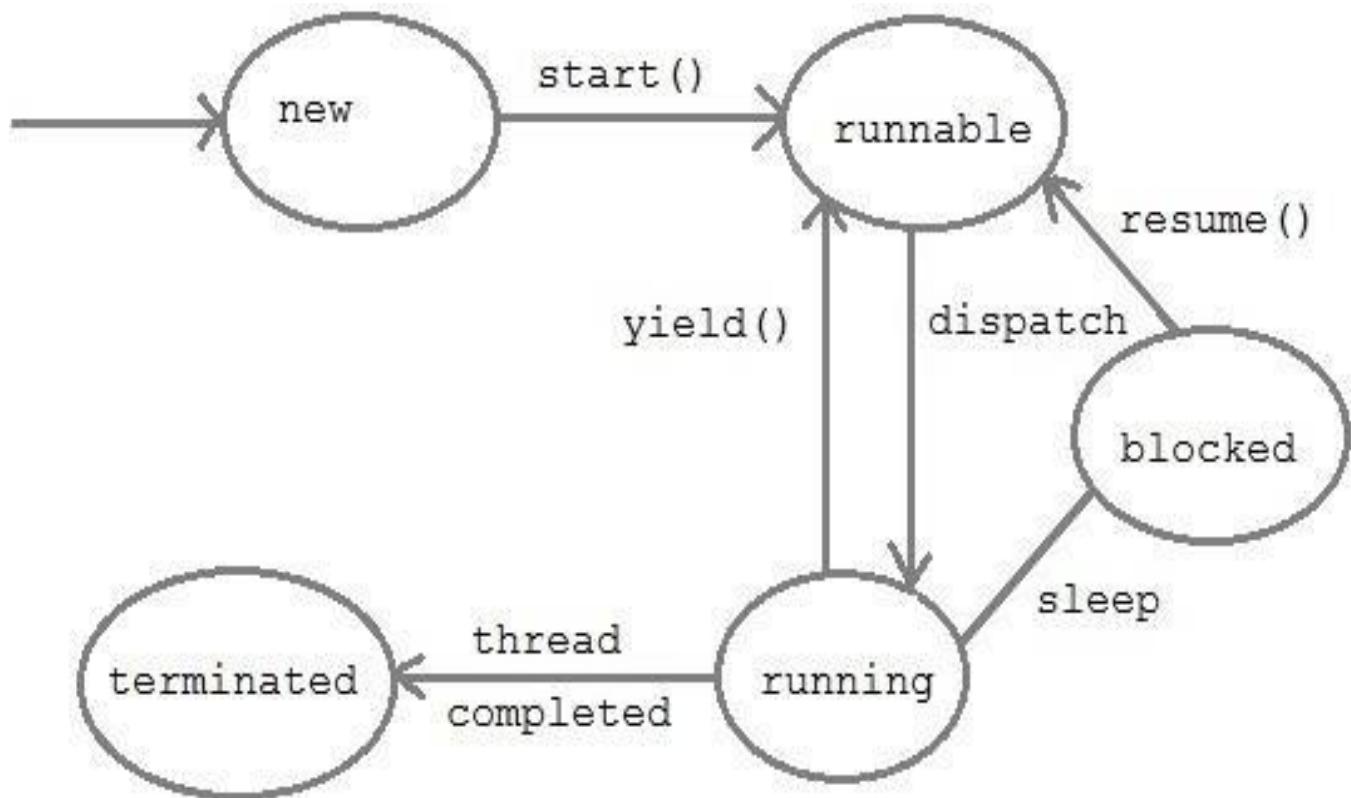
In Java, the word **thread** means two different things.

- An instance of **Thread** class.
- or, A thread of execution.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.



Life cycle of a Thread



1. **New** : A thread begins its life cycle in the new state. It remains in this state until the `start()` method is called on it.
2. **Runnable** : After invocation of `start()` method on new thread, the thread becomes runnable.
3. **Running** : A method is in running thread if the thread scheduler has selected it.
4. **Blocking** : A thread is waiting for another thread to perform a task (such as `sleep`, I/O operations, block suspend, wait). In this stage the thread is still alive.
5. **Terminated** : A thread enters the terminated state when it completes its task.

3.2.3 Creating a Threads

We can a thread by instantiating an object of type Thread. This can be accomplish by in two ways

1. By extending Thread class.
2. By implementing 'Runnable' interface.

1. Steps for Creation of Thread by extending Thread class:

a) Create a class as subclass to Thread Class

Ex: class ThreadDemo extends Thread

b) Write the functionality of user thread with in run method

Ex: public void run()

{

Functionality;

}

c) Create the object of the class that is extending Thread class

Ex: ThreadDemo td=new ThreadDemo()

d) Attach the above created object to the Thread class

Ex: Thread t=new Thread(td);

e) Execute the user thread by invoking start()

t.start()

package threads;

public class ThreadDemo extends Thread

{

public void run()

```

{
    for(int i=1;i<=3;i++)
    {
        System.out.println("user thread:"+i);
    }
}

public static void main(String args[])
{
    ThreadDemo td=new ThreadDemo();

    Thread t=new Thread(td);

    t.start();
}
}

```

Output:

user thread:1

user thread:2

user thread:3

2. Steps for Creation of Thread by implementing 'Runnable' interface:

- a) Create a class implementing Runnable interface

Ex: class RunnableDemo implements Runnable

- b) Write the functionality of user thread with in run method

Ex:public void run() {

Functionality;

```
}
```

- c) Create the object class that is implementing Runnable interface

```
Ex:RunnableDemo rd=new RunnableDemo();
```

- d) Attach the above created object to the Thread class

```
Ex:Thread t=new Thread(rd);
```

- e) Execute the user thread by invoking start()

```
t.start()
```

```
public class RunnableDemo implements Runnable
```

```
{
```

```
    public void run()
```

```
    {
```

```
        for(int i=1;i<=10;i++)
```

```
        {
```

```
            System.out.println("user thread:"+i);
```

```
        }
```

```
    }
```

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

```
{
```

```
    RunnableDemo rd=new RunnableDemo();
```

```
    Thread t=new Thread(rd);
```

```
    t.start();
```

```
}
```



```
}
```

Output:

```
user thread:1
```

```
user thread:2
```

```
user thread:3
```

```
user thread:4
```

```
user thread:5
```

```
user thread:6
```

```
user thread:7
```

```
user thread:8
```

```
user thread:9
```

```
user thread:10
```

The *main* thread :

Even if you don't create any thread in your program, a thread called **main** thread is still created. Although the **main** thread is automatically created, you can control it by obtaining a reference to it by calling **currentThread()** method.

Two important things to know about **main** thread are,

- It is the thread from which other threads will be produced.
- **main** thread must be always the last thread to finish execution.

```
class MainThread
```

```
{
```

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

```
{
```

```
Thread t=Thread.currentThread();
```

```

t.setName("MainThread");

System.out.println("Name of thread is "+t);

}

}

```

3.2.4 Thread Priorities

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In java thread priority ranges between,

- MIN-PRIORITY (a constant of 1)
- MAX-PRIORITY (a constant of 10)

By default every thread is given a NORM-PRIORITY(5). The **main**thread always have NORM-PRIORITY.

Example on Thread Priority

```

class Multi implements Runnable{

public void run() {

    System.out.println("running thread name is:"+Thread.currentThread().getName());

    System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

}

public static void main(String args[]) {

    Multi m1=new Multi ();

    Multi m2=new Multi ();

    m1.setPriority(Thread.MIN_PRIORITY);

```

```
m2.setPriority(Thread.MAX_PRIORITY);

m1.start();

m2.start();

}

}
```

Output:

running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

3.2.5 Thread Synchronizing

Synchronization is the capability of control the access of multiple threads to any shared resource. Synchronization is better in case we want only one thread can access the shared resource at a time.

Use of Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

- Mutual Exclusive
 1. Synchronized method.
 2. Synchronized block.
 3. static synchronization.
- Cooperation (Inter-thread communication)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java by using:

1. synchronized method
2. synchronized block
3. static synchronization

Understanding the concept of Lock

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Understanding the problem without Synchronization

```
class First
{
    public void display(String msg)
    {
        System.out.print "["+msg);

        try
        {

            Thread.sleep(2000);
```

```

    }

    catch(InterruptedException e)
    {
        System.out.println("error is"+e);
    }
    System.out.print("]");
}
}
class Second extends Thread
{
    String msg;
    First fobj;
    Second(First fp,String str)
    {

        fobj=fp;
        msg=str;
        start();
    }
    public void run()

    {

        fobj.display(msg);
    }
}
public class UnSynchroMethod

{

    public static void main(String args[])

    {

        First fnew=new First();

        Second s1=new Second(fnew,"welcome");

        Second s2=new Second(fnew,"new");

        Second s3=new Second(fnew,"program");

    }

}

```

Output:

[welcome[program[new]]]

1.Thread synchronization using synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the method returns

```
class First
{
    synchronized public void display(String msg)
    {
        System.out.print("[ "+msg);
        try {
            Thread.sleep(2000);
        }
        catch(InterruptedException e)
        {
            System.out.println("error is"+e);
        }
        System.out.print("]");
    }
}
```

```
class Second extends Thread
{
    String msg;
    First fobj;
    Second(First fp,String str)
    {
        fobj=fp;
        msg=str;
        start();
    }
    public void run()
    {
        fobj.display(msg);
    }
}
```

```
public class SynchroBlock

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

        First fnew=new First();
```

```

        Second s1= new Second(fnew,"welcome");
        Second s2= new Second(fnew,"new");
        Second s3= new Second(fnew,"program");
    }

}

```

Output:

[welcome][new][program]

2.Thread synchronization using Synchronized block

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```

synchronized (object reference expression) {

    //code block

}

```

Example on Thread synchronization using synchronized block

```

class First

{

    public void display(String msg)

    {

        System.out.print "["+msg);
        try
        {

            Thread.sleep(2000);

```

```

    }
    catch(InterruptedException e)
    {
        System.out.println("error is"+e);

    }
    System.out.print("]");

}
}
class Second extends Thread
{
    String msg;
    First fobj;
    Second(First fp,String str)
    {

        fobj=fp;
        msg=str;
        start();

    }

    public void run()
    {
        synchronized(fobj)

        {
            fobj.display(msg);

        }

    }
}
public class SynchroBlock

{
    public static void main(String args[])

    {
        First fnew=new First();

        Second s1=new Second(fnew,"welcome");
        Second s2=new Second(fnew,"new");
        Second s3=new Second(fnew,"program");

    }
}

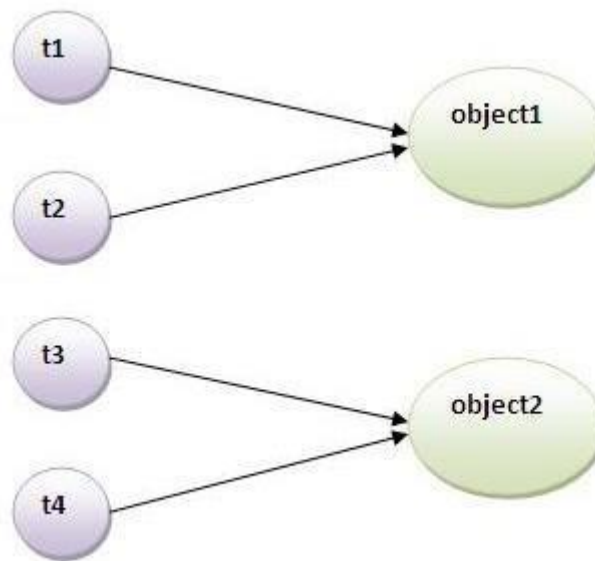
```

Output:

[welcome][new][program]

3. Thread synchronization using Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refer to a common object that has a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of static synchronization

In this example we are applying the synchronized keyword on the static method to perform static synchronization.

```

class Table{

    synchronized static void printTable(int n){

        for(int i=1;i<=10;i++){

            System.out.println(n*i);

            try{

```

```
        Thread.sleep(400);

    }catch(Exception e){ }

}

}

}

class MyThread1 extends Thread{

    public void run(){

        Table.printTable(1);

    }

}

class MyThread2 extends Thread{

    public void run(){

        Table.printTable(10);

    }

}

class MyThread3 extends Thread{

    public void run(){

        Table.printTable(100);

    }

}

class MyThread4 extends Thread{

    public void run(){
```

```
Table.printTable(1000);

}

}

class Use{

public static void main(String t[]){

MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

MyThread4 t4=new MyThread4();

t1.start();

t2.start();

t3.start();

t4.start();

}

}
```

3.2.6 Inter Thread Communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
<code>public final void wait()throws InterruptedException</code>	waits until object is notified.
<code>public final void wait(long timeout)throws InterruptedException</code>	waits for the specified amount of time.

2) `notify()` method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

```
public final void notify()
```

3) `notifyAll()` method

Wakes up all threads that are waiting on this object's monitor. Syntax:

```
public final void notifyAll()
```

Why `wait()`, `notify()` and `notifyAll()` methods are defined in `Object` class not `Thread` class?

It is because they are related to lock and object has a lock.

Difference between `wait` and `sleep`?

Let's see the important differences between `wait` and `sleep` methods. **package** `interthreadcommunication`;

<code>wait()</code>	<code>sleep()</code>
<code>wait()</code> method releases the lock	<code>sleep()</code> method doesn't release the lock.
is the method of <code>Object</code> class	is the method of <code>Thread</code> class
is the non-static method	is the static method
should be notified by <code>notify()</code> or <code>notifyAll()</code> methods	after the specified amount of time, <code>sleep</code> is completed.

Example1: Inter Thread Communication on Banking application

```
class Customer {  
  
    int amount=10000;  
  
    synchronized void withdraw(int amount)    {  
  
        System.out.println("going to withdraw");  
  
        if(this.amount<amount) {  
  
            System.out.println("insuffient funds,waiting for deposit");  
  
            try {  
  
                wait();  
  
            }  
  
            catch(Exception e)  
  
            {  
  
                System.out.println("error"+e);  
  
            }  
  
        }  
  
        this.amount=this.amount-amount;  
  
        System.out.println("withdraw is completed");  
  
        System.out.println("remaining balance amount:"+this.amount);  
  
    }  
  
    synchronized void deposit(int amount) {  
  
        System.out.println("going to deposit amount");  
  
        this.amount=this.amount+amount;  
  
        System.out.println("deposite completed");  
    }  
}
```

```
        System.out.println("balance after deposit:"+this.amount);

        notify();

    }

}

public class interthread {

    public static void main(String args[])

    {

        final Customer c=new Customer();

        new Thread() {

            public void run() {

                c.withdraw(15000);

            }

        }.start();

        new Thread() {

            public void run() {

                c.deposit(10000);

            }

        }.start();

    }

}
```

Output :

going to withdraw

insuffient funds,waiting for deposit

going to deposit amount

deposite completed

balance after deposit:20000

withdraw is completed

remaining balance amount:5000

Example:2 Producer-Consumer Problem Using Inter-thread Communication

Aim: Write a Java program that correctly implements the producer – consumer problem using the concept of inter thread communication.

THEORY: Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Program:

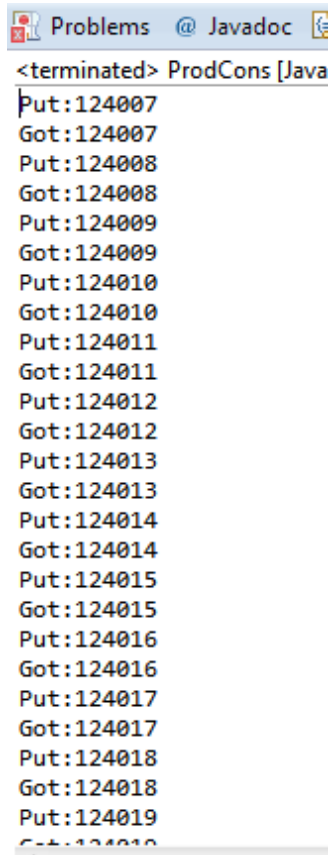
```
class Q
{
    int n;
    boolean valueSet=false;
    synchronized int get()
    {
        if(!valueSet) try
        {
            wait();
        }
        catch(InterruptedException e)
        {
            System.out.println("InterruptedException caught");
        }
        System.out.println("Got:"+n); valueSet=false;
        notify();
        return n;
    }
    synchronized void put(int n)
    {
        if(valueSet) try
        {
            wait();
        }
        catch(InterruptedException e)
        {
            System.out.println("InterruptedException caught");
        }
    }
}
```

```

    }
    this.n=n; valueSet=true;
    System.out.println("Put:"+n); notify();
    }
    }
    class Producer implements Runnable
    {
    Q q; Producer(Q q)
    {
    this.q=q;
    new Thread(this,"Producer").start();
    }
    public void run()
    {
    int i=0;
    while(true)
    {
    q.put(i++);
    }
    }
    }
    class Consumer implements Runnable
    {
    Q q; Consumer(Q q)
    {
    this.q=q;
    new Thread(this,"Consumer").start();
    }
    public void run()
    {
    while(true)
    {
    q.get();
    }
    }
    }
    class ProdCons
    {
    public static void main(String[] args)
    {
    Q q=new Q();
    new Producer(q);

    new Consumer(q);
    System.out.println("Press Control-c to stop");
    }
    }
    Output:

```

Problems @ Javadoc

<terminated> ProdCons [Java]

Put:124007
Got:124007
Put:124008
Got:124008
Put:124009
Got:124009
Put:124010
Got:124010
Put:124011
Got:124011
Put:124012
Got:124012
Put:124013
Got:124013
Put:124014
Got:124014
Put:124015
Got:124015
Put:124016
Got:124016
Put:124017
Got:124017
Put:124018
Got:124018
Put:124019
Got:124019

Multithreaded Program

Aim: Write a Java program that implements a multithreaded program has three threads. First thread generates a random integer every 1 second and if the value is even, second thread computes the square of the number and prints. If the value is odd the third thread will print the value of cube of the number.





Program:

```
import java.io.*;
import java.util.*;
class First1 extends Thread
{
    public void run()
    {
        for(;;)
        {
            int roll;
            Random d = new Random();
            roll = d.nextInt(200) + 1;
            System.out.println(roll);
            Thread t2=new Second2(roll);
            Thread t3=new Third3(roll);
            try
            {
                Thread.sleep(1000);
                if(roll%2==0)
                    t2.start();
                else
                    t3.start();
            }
            catch(InterruptedException e){}
        }
    }
}
class Second2 extends Thread
{
    int r1;
    Second2(int r)
    {
        r1=r;
    }
    public void run()
    {
        System.out.println("The square of
        number"+r1+"is:"+r1*r1);
    }
}
```

```

class Third3 extends Thread
{
int r1;
Third3(int r)
{
r1=r;
}
public void run()
{
System.out.println("The Cube of the Number"+r1+"is: "+r1*r1*r1);
}
}
class Mthread
{
public static void main(String[] args)
{
Thread t1=new First1();
System.out.println("press Ctrl+c to stop .....");
t1.start();
}
}

```

 Problems
  Javadoc
  Declaration
  C

Mthread [Java Application] C:\Program Files\Java\

|press Ctrl+c to stop.....

9

31

The Cube of the Number9is: 729

89

The Cube of the Number31is: 29791

196

The Cube of the Number89is: 704969

26

The square of number196is:38416

113

The square of number26is:676

92

The Cube of the Number113is: 1442897