Java Programming

Unit – 4
Multithreading and Collections

TABLE OF CONTENTS

Creating Threads

Multi Tasking

- There are two distinct types of multitasking:

  1. Process-based
  2. Thread-based.

- **Process-based** multitasking is the feature that allows our computer to run two or more programs concurrently.
- In a **Thread-based** multitasking environment, a single program can perform two or more tasks simultaneously.

Thread

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Multithreading is a specialized form of multitasking.
- Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Difference between Process and Thread

Process	Thread
A Process is a program under execution. A Process contains many threads	A Thread is a single flow of execution and is a segment of a Program
Creation of new Process requires a new address space and resources	Threads can be created in the same address space. It saves memory space and OS resources
Processes have different code and data segments.	Threads share the code and data segments i.e., if one thread modifies a variable, all threads see the new value of that variable
Processes are heavy weight components	Threads are light weight components.
Communication between processes can be established using Inter Process Communication mechanism. Ex: Sockets and pipes	Communication between threads are very simple and efficient. Ex: Inter Thread Communication

Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program, because it is the one that is executed when our program begins.
- The main thread is important for two reasons:
    - It is the thread from which other “child” threads will be spawned.
    - Often it must be the last thread to finish execution because it performs various shutdown actions.

Main Thread Program

```
class CurrentThreadDemo
{               public static void main(String args[])
                {               Thread t = Thread.currentThread();
                                System.out.println("Current thread: " + t);
                                // change the name of the thread
                                t.setName("My Thread");
                                System.out.println("After name change: " + t);
                                try
                                {               for(int n = 5; n > 0; n--)
                                                {               System.out.println(n);
                                                                t.sleep(1000);
                                                }
                                }
                                catch (InterruptedException e)
                                {               System.out.println("Main thread interrupted");
                                }
                }
}
```

- A reference to the current thread (the main thread, in this case) is obtained by calling currentThread( ), and this reference is stored in the local variable t.
- setName( ) to change the internal name of the thread.
- The argument to sleep( ) specifies the delay period in milliseconds.
- Notice the try/catch block around this loop. The sleep( ) method in Thread might throw an InterruptedException.
- The name of the thread, its priority, and the name of its group.

Creating a Thread

- In the most general sense, we create a thread by instantiating an object of type Thread.
- Java defines two ways
  - We can implement the Runnable interface.
  - We can extend the Thread class.

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- To implement Runnable, a class need only implement a single method called run( ), which is declared like this: public void run( )
- Inside run( ), we will define the code that constitutes the new thread.
- After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class.

Implementing Runnable

- Thread defines several constructors.

  Thread(Runnable threadOb, String threadName)

- In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its start( ) Method
- void start( )

Runnable Interface Program

```
class NewThread implements Runnable
{               Thread t;
                NewThread()
                {t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                                t.start();
                }
                public void run()
                {  try
                   { for(int i = 5; i > 0; i--)
                      {
                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                      }
                   }
                catch (InterruptedException e)
                {System.out.println("Child interrupted.");}
                System.out.println("Exiting child thread.");
                }
}
```

```
class ThreadDemo
{       public static void main(String args[])
        {       new NewThread();
                try
                { for(int i = 5; i > 0; i--)
                { System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
                }
                }
                catch (InterruptedException e)
                {
System.out.println("Main thread interrupted.");
                }
        System.out.println("Main thread exiting.");
        }
}
```

Extending Thread

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run() method, which is the entry point for the new thread.
- It must also call start() to begin execution of the new thread.

Extending Thread Class Program

```
class NewThread1 extends Thread
{
        NewThread1()
        {
        System.out.println("Child thread: " + this);
        start();
        }
        public void run()
        {  try
           { for(int i = 5; i > 0; i--)
             {
        System.out.println("Child Thread: " + i);
                        Thread.sleep(500);
             }
           }
        catch (InterruptedException e)
        {System.out.println("Child interrupted.");}
        System.out.println("Exiting child thread.");
        }
}
```

```
class ThreadDemo
{     public static void main(String args[])
      {       new NewThread1();
              try
              {
              for(int i = 5; i > 0; i--)
              {
      System.out.println("Main Thread: " + i);
              Thread.sleep(1000);
              }
              }
              catch (InterruptedException e)
              {
System.out.println("Main thread interrupted.");
              } System.out.println("Main thread exiting.");
      }
}
```

Creating Multiple Threads

- So far, we have been using only two threads: the main thread and one child thread.
- However, our program can spawn as many threads as it needs.

Creating Multiple Threads Program

```
class NewThread2 implements Runnable
{
        String name;
        Thread t;
        NewThread2(String threadname)
        {
                name = threadname;
                t = new Thread(this, name);
        System.out.println("New thread: " + t);
                t.start();
        }
        public void run()
        {       try
                {for(int i = 5; i > 0; i--)
                {
        System.out.println(name + ": " + i);
                Thread.sleep(1000);
                }
                }
        catch (InterruptedException e){
                System.out.println(name +"Interrupted");
                }
        System.out.println(name + " exiting.");
        }
        }
        class MultiThreadDemo{
                public static void main(String args[])
                {
                        new NewThread2("One");
                        new NewThread2("Two");
                        new NewThread2("Three");
        try{
        Thread.sleep(10000);}
        catch (InterruptedException e){
        System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
        }
        }
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higher-priority thread can also preempt a lower-priority one.
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread.
- final void setPriority(int level)

  Here, level specifies the new priority setting for the calling thread.

Thread Priorities

- The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as final variables within Thread.
- We can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

```
final int getPriority( )
```


Thread Priorities Program

```
class A extends Thread
{               public void run()
                {
                                for (int i=1;i<=5; i++)
                                {
System.out.println("\tFrom Thread A : i="+i);
                                }
                System.out.println("Exit From A ");
                }
}
 class B extends Thread
{               public void run()
                {
                                for (int j=1;j<=5 ;j++)
                                {
System.out.println("\tFrom Thread B : j="+j);
                                }
                System.out.println("Exit From B ");
                }
 }
```

```
class C extends Thread
{
                public void run()
                {
                                for (int k=1;k<=5; k++)
                                {
System.out.println("\tFrom Thread C : K="+k);
                                }
                System.out.println("Exit From C ");
                }
}
```

Thread Priorities Program

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

                                               c.setPriority(Thread.MAX_PRIORITY);
                                               b.setPriority(Thread.NORM_PRIORITY);
 //b.setPriority(c.getPriority()-2));
                                               a.setPriority(Thread.MIN_PRIORITY);
//a.setPriority(b.getPriority()+1));
                                               System.out.println("Start Thread C ");
                                               c.start();
                                               System.out.println("Start Thread B ");
                                               b.start();
                                               System.out.println("Start Thread A ");
                                               a.start();
                                               System.out.println("End of Main Thread ");
                        }
}
```

Using isAlive() and join

- Two ways exist to determine whether a thread has finished.
- First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:
  - final boolean isAlive( )
- The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

Using isAlive() and join

- While **isAlive( )** is occasionally useful.
- The method that we will more commonly use to wait for a thread to finish is called **join( )**, shown here:
  - final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates.

isAlive() and join() Program

```
class NewThread4 implements Runnable
{                       String name;            Thread t;
                        NewThread4(String threadname)
                        {       name = threadname;
                                t = new Thread(this, name);
                                System.out.println("New thread: " + t);
                                t.start();
                        }
                        public void run()
                        {       try
                                {       for(int i = 5; i > 0; i--)
                                        {  System.out.println(name + ": " + i);
                                        Thread.sleep(1000);     }
                                }
                                catch (InterruptedException e)
                                { System.out.println(name + " interrupted."); }
                          System.out.println(name + " exiting.");
                        }
}
```

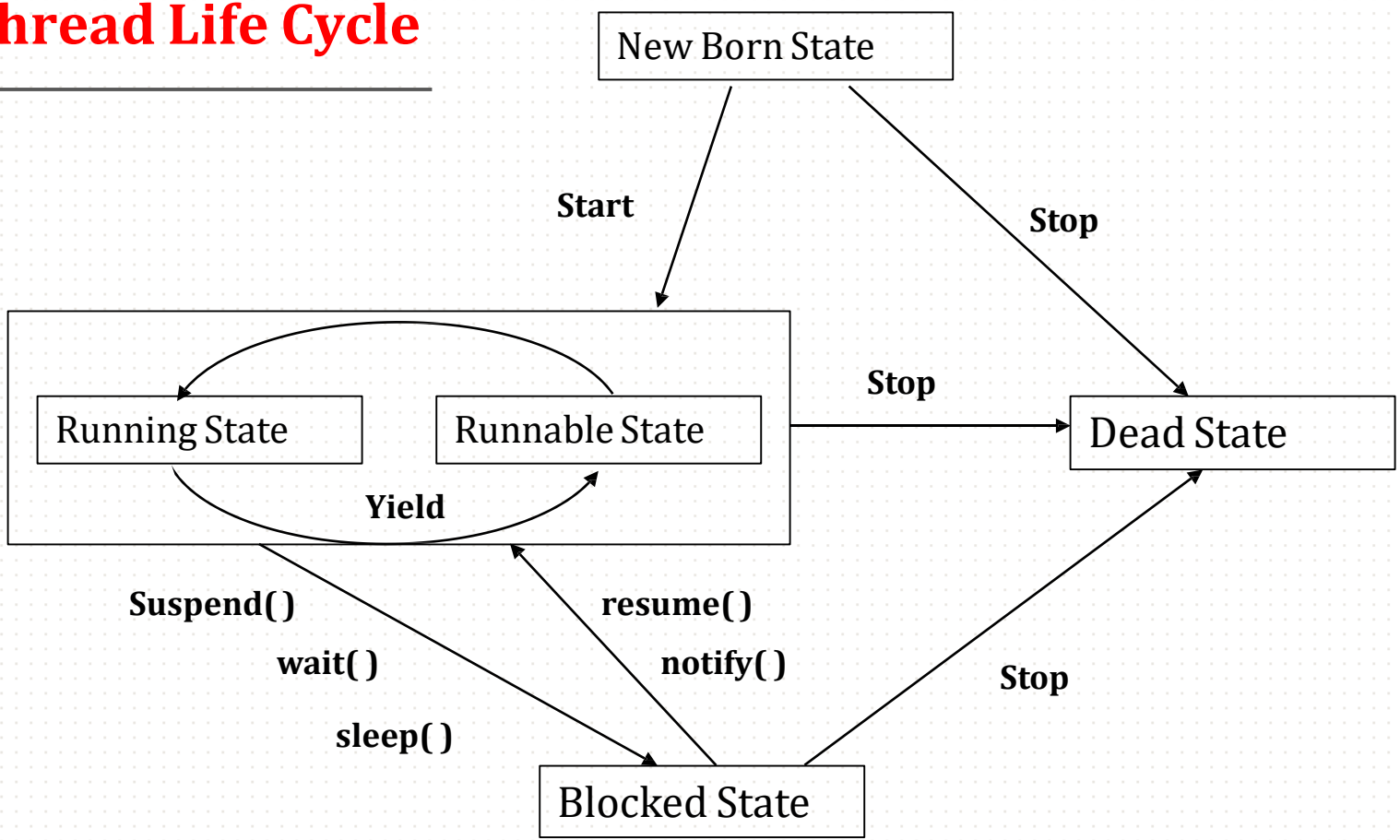
isAlive() and join() Program

```
class isalivejoin
{               public static void main(String args[])
                {               NewThread4 ob1 = new NewThread4("One");
                                NewThread4 ob2 = new NewThread4("Two");
                                NewThread4 ob3 = new NewThread4("Three");
                                System.out.println("Thread One is alive: "+ ob1.t.isAlive());
                                System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
                                System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
                                try
                                {               System.out.println("Waiting for threads to finish.");
                                                ob1.t.join();ob2.t.join();ob3.t.join();
                                }
                                catch (InterruptedException e)
                                {               System.out.println("Main thread Interrupted"); }

                                 System.out.println("Thread One is alive: "+ ob1.t.isAlive());
                                System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
                                System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
                                System.out.println("Main thread exiting.");
                }
}
```

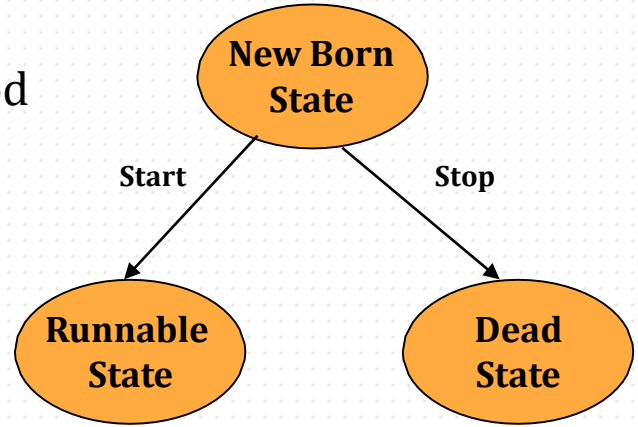
Thread Life Cycle

Thread Life Cycle



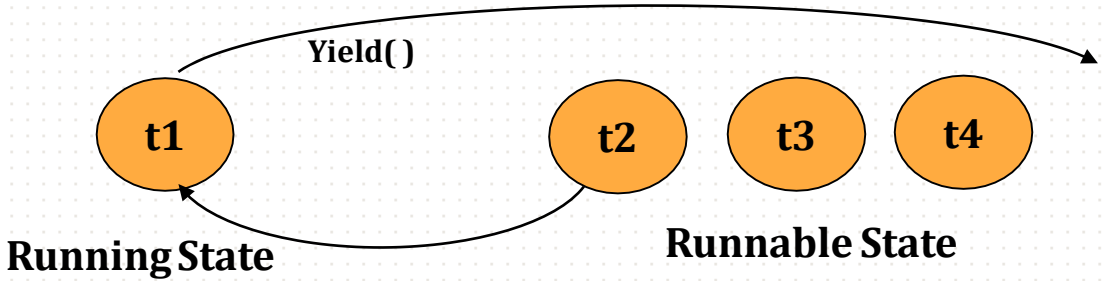
New born State

- When we create a thread object then the thread is in New born state.
- It has two alternatives
  - Schedule it for running using start() method
  - Kill it using stop() method



Runnable State

- It means thread is ready for execution and is waiting for the availability of the processor. If all threads have equal priority then they are given time slots for execution in Round Robin fashion.
- We can move the threads from running state to runnable state by giving yield() command.



Running State

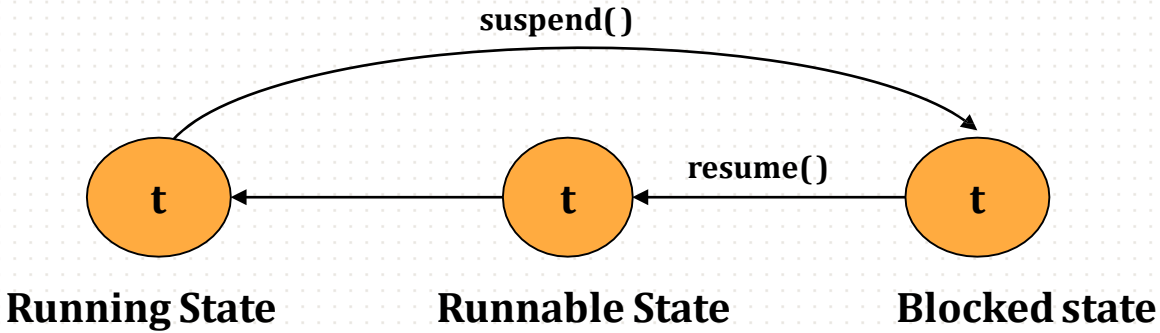
- It means the processor has given its time to the thread for its execution. The thread runs until it relinquishes the control on its own or it is preempted by a higher priority thread

Blocked State

- A thread is said to be blocked when it is prevented from entering into runnable state and subsequently running state. It is also called “Not runnable state”

Suspend()

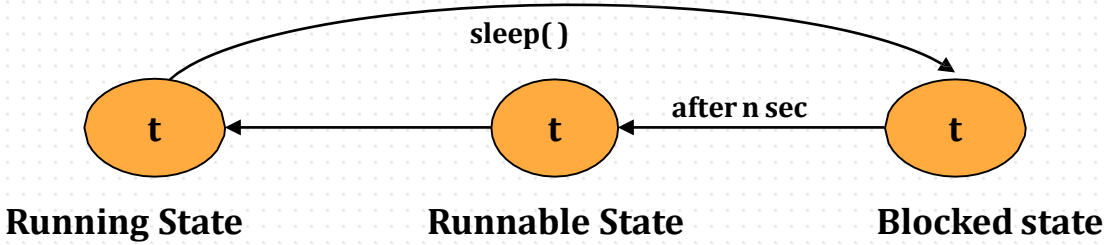
- A thread can be suspended using suspend() method



- A suspended thread can be received by using resume() method

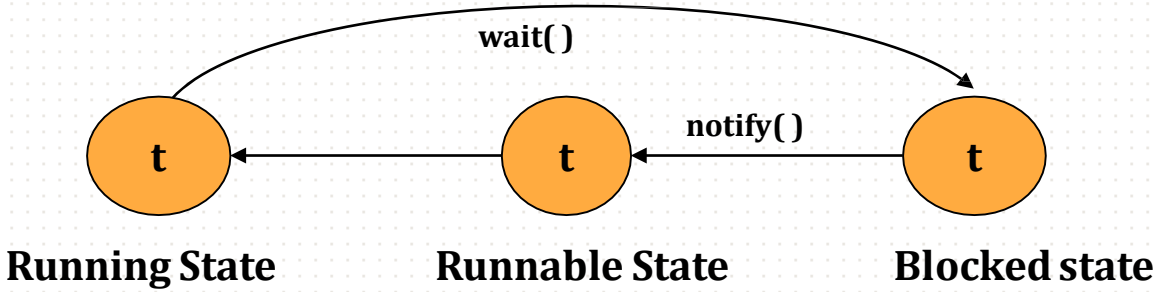
Sleep()

- We can put a thread into sleep mode for a specified time period using sleep() method



wait()

- The thread has to wait until some event occurs. This is done using a wait() method. The thread can be scheduled to run again using notify() command



Dead State

- A running thread ends its life when it is completed its execution is called " Natural Death" otherwise we can kill using a stop() message then it is called "Preemptive death"

Synchronization

Synchronization

- One thread may try to read a record from a file while another is still writing to the same file. This time we may get strange results.
- Java enables us to overcome this problem using a technique known as Synchronization
- When two or more threads need access to a resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization.

Synchronization

- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.

Synchronized Method

- When we declare a method synchronized, Java creates a “monitor” and hands it over to the thread that calls the method first time.
- As long as the thread holds the monitor, no other thread can enter the synchronized section of code.
- A monitor is like a key and the thread that holds the key can only open the lock.
- This is the general form of the synchronized statement:

```
synchronized method()
        {
                // statements to be synchronized
        }
```


Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.
Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.
If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Syntax

```
synchronized (object reference expression) {
  //code block
}
```

Suspend - Resume

- The **suspend()** method of thread class puts the thread from running to waiting state.
- This method is used if you want to stop the thread execution and start it again when a certain event occurs.
- This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Suspend – Resume Program

```
public class JavaResumeExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
         try
         {
            // thread to sleep for 500 milliseconds
             sleep(500);
             System.out.println(Thread.currentThre
ad().getName());
         }catch(InterruptedException e){System.o
ut.println(e);}
         System.out.println(i);
      }
   }
```

```
   public static void main(String args[])
      {
         // creating three threads
         JavaResumeExp t1=new JavaResumeExp ();
         JavaResumeExp t2=new JavaResumeExp ();
         JavaResumeExp t3=new JavaResumeExp ();
         // call run() method
         t1.start();
         t2.start();
         t2.suspend(); // suspend t2 thread
         // call run() method
         t3.start();
System.out.println("Thread going to sleep for 5 seconds");
 // Sleeping thread for specific amount of time
         Thread.sleep(5000);
// Display message
         System.out.println("Thread Resumed");
         t2.resume(); // resume t2 thread
      }
}
```

Inter Thread Communication

Interthread Communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.
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 :
wait()
notify()
notifyAll()

Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object.

public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

Interthread Communication

- wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same CriticalSection and calls notify( ).
- notify( ) wakes up the first thread that called wait( ) on the same object.

  final void notify( )
- notifyAll( ) wakes up all the threads that called wait( ) on the same object.The highest priority thread will run first.

  final void notifyAll( )

Interthread Communication

- One thread is producing some data and another is consuming it.
- The producer before it generates more data, It has to wait until the consumer is finished.
- All three methods can be called only from within a synchronized context.

Producer Consumer Program

```
class Q
{
        int n;
        boolean valueSet = false;

        synchronized int get()
        {       if(!valueSet)
                {       try
                        {  wait();     }
                  catch(InterruptedException e)
                    {
System.out.println(“Exception 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();
     }
}
```

Producer Consumer Program

```
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 PC1
{
            public static void main(String args[])
            {
                        Q q = new Q();

                        new Producer(q);
                        new Consumer(q);

System.out.println("Press Control+C to stop.");
            }
}
```

Daemon Threads

- Daemon thread in Java is a service provider thread that provides services to the user thread.
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.

Daemon Threads

- Daemon threads are designed as low level background threads that perform useful work. One example of Daemon thread is the “Garbage Collector thread”
- To create a Daemon thread call setDaemon method with a boolean true argument value i.e., setDaemon(true)
- To determine whether a thread object is associated with a Daemon thread call isDaemon(). It returns boolean value T/F

Simple example of Daemon thread in java

```
public class TestDaemonThread1 extends Thread{
 public void run(){
 if(Thread.currentThread().isDaemon()){//checking for daemon thread
  System.out.println("daemon thread work");
 }
 else{
 System.out.println("user thread work");
}
}
 public static void main(String[] args){
 TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
 TestDaemonThread1 t2=new TestDaemonThread1();
 TestDaemonThread1 t3=new TestDaemonThread1();

 t1.setDaemon(true);//now t1 is daemon thread

 t1.start();//starting threads
 t2.start();
 t3.start();
}
}
```

Exploring java.io

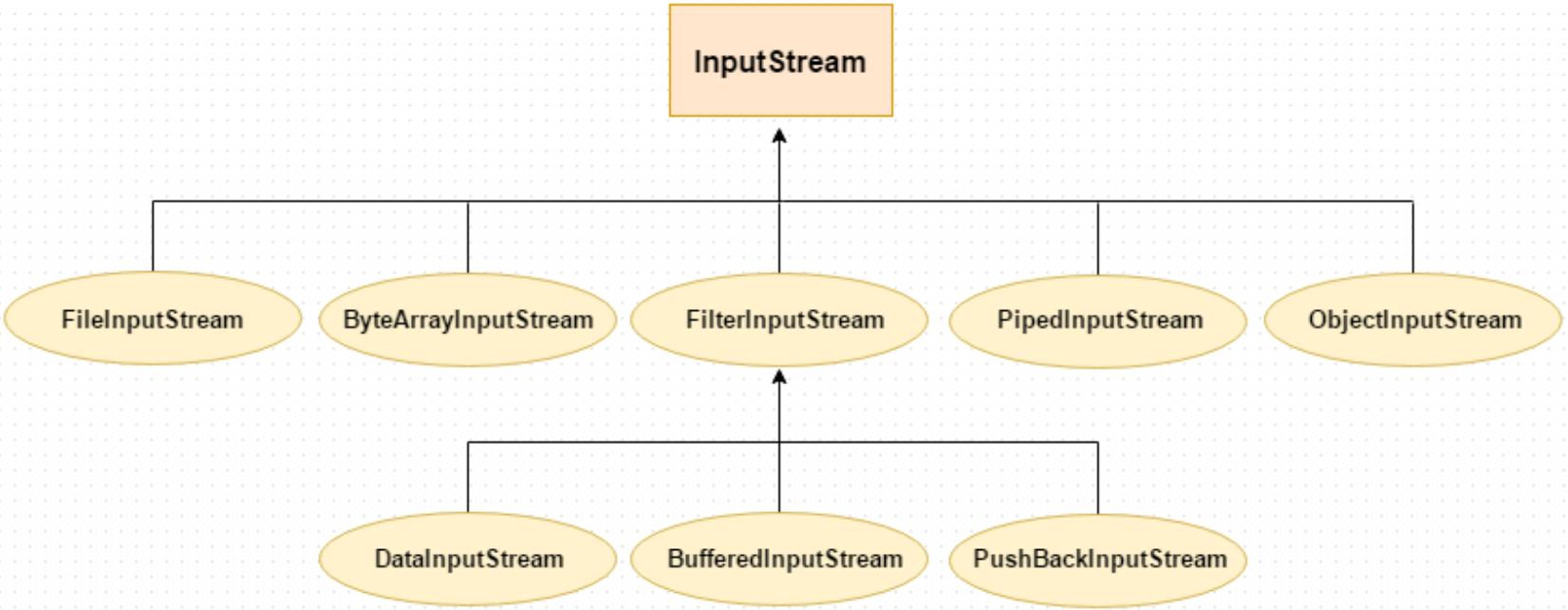
The **java.io** package is used to handle input and output operations. Java IO has various classes that handle input and output sources. A stream is a sequence of data.
Java input stream classes can be used to read data from input sources such as **keyboard** or a **file**. Similarly output stream classes can be used to write data on a **display** or a **file** again.

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader

DataInputStream
DataOutputStream
File
FileInputStream
FileOutputStream
FileReader and FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter
InputStream
InputStreamReader

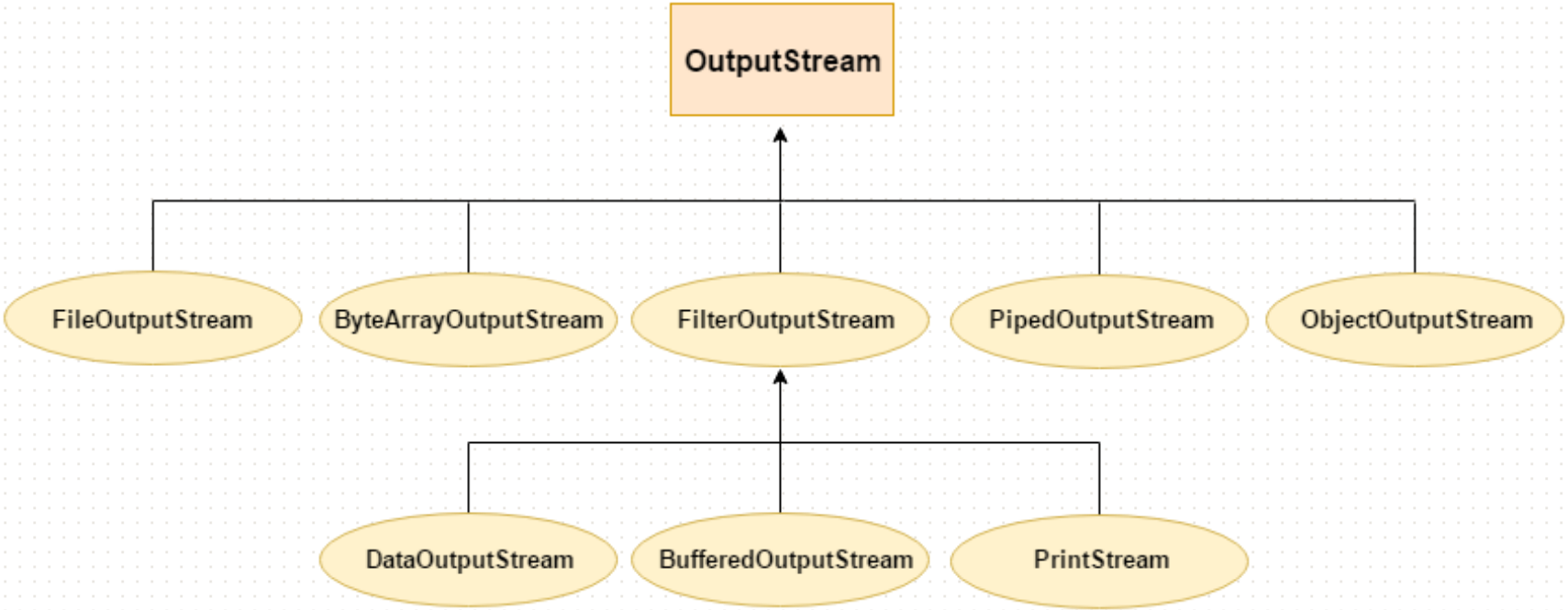
InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket



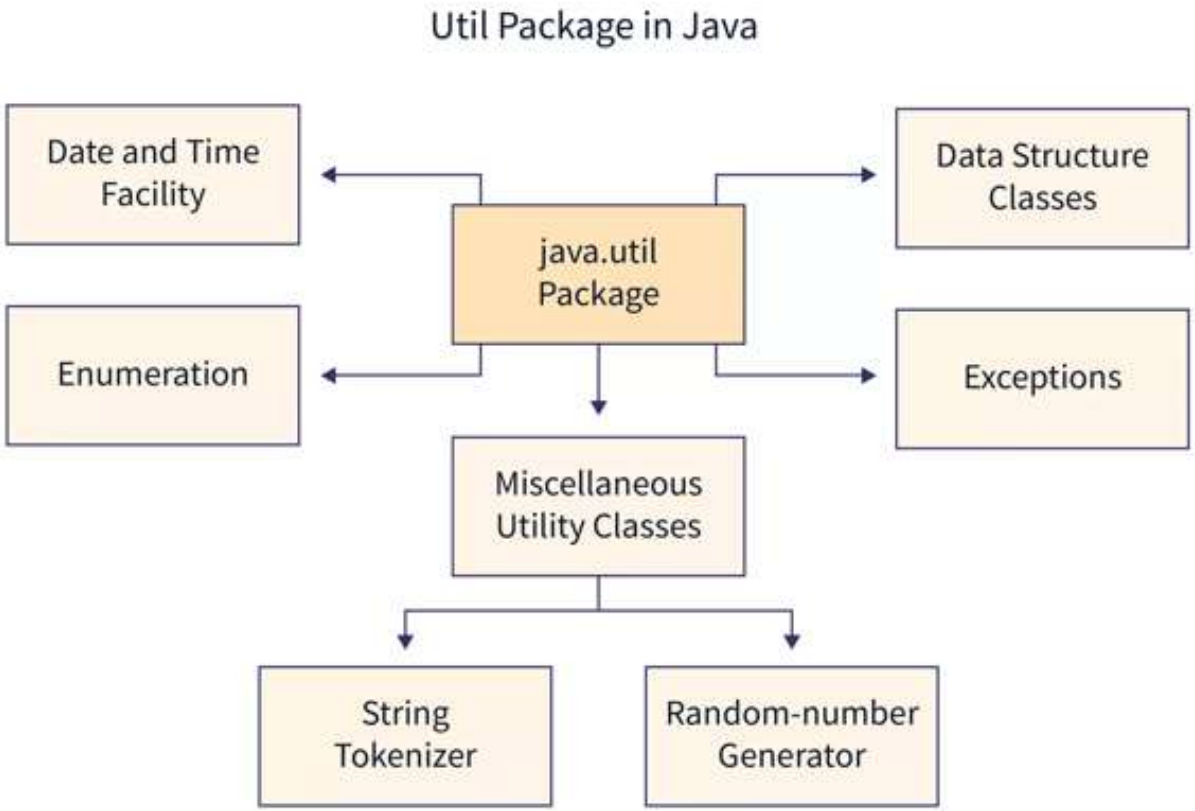
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.



Java.util Package

The java.util package in Java is a built-in package that contains various utility classes and interfaces. It provides basic functionality for commonly occurring use cases. It contains Java's collections framework, date and time utilities, string-tokenizer, event-model utilities, etc.

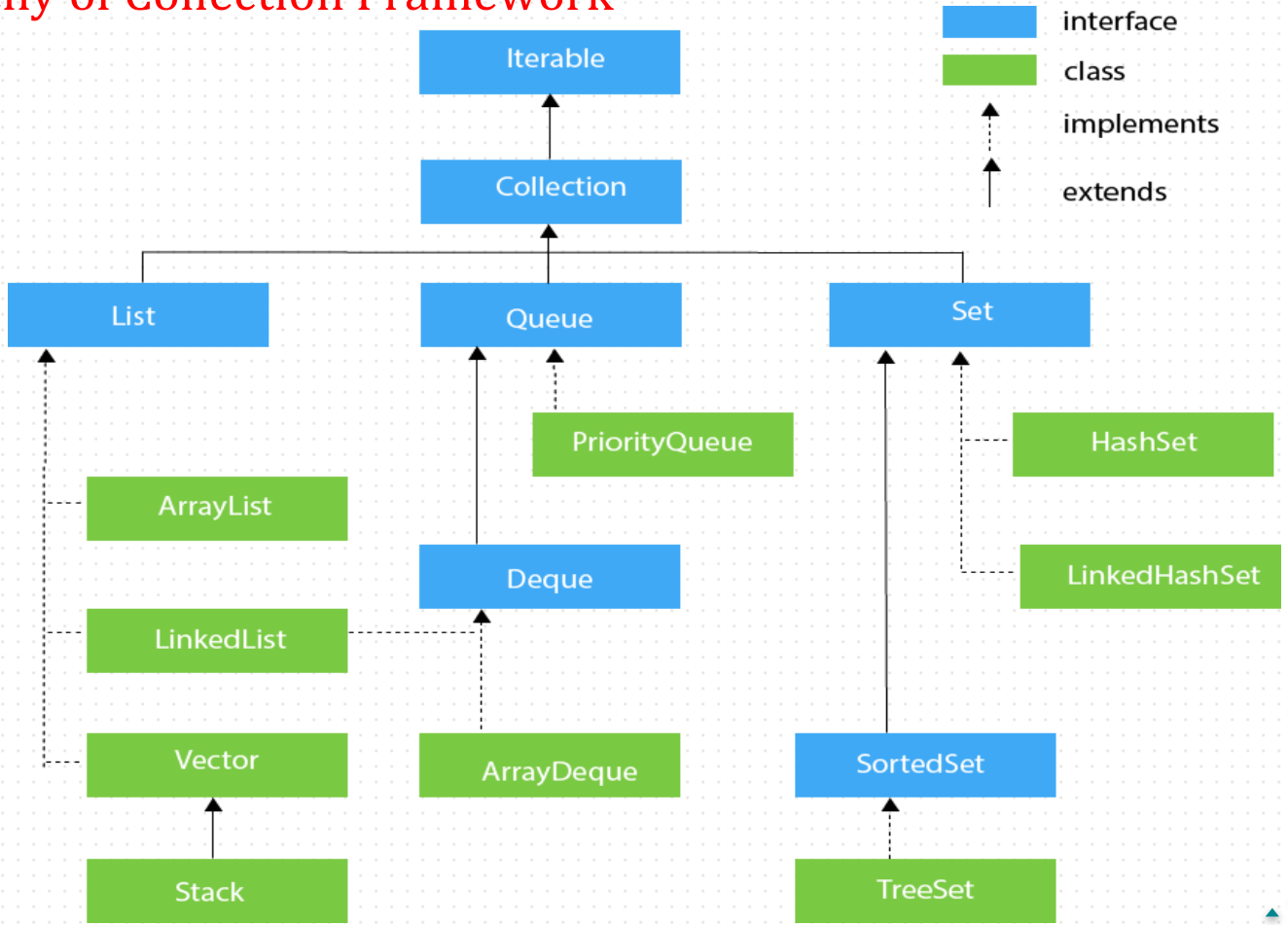

Util Package in Java
Date and Time Facility
Data Structure Classes
java.util Package
Enumeration
Exceptions
Miscellaneous Utility Classes
String Tokenizer
Random-number Generator

Collections: Overview of Collection Framework

What is Collections Framework in Java?

Collections Framework in Java is a special part of the **java.util** package that can be used to represent and manipulate collections. It provides easy storage and organization of a group of objects (or collection). It includes pre-written classes, interfaces, and algorithms under a unified architecture.

Hierarchy of Collection Framework



ArrayList

It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
void clear()	It is used to remove all of the elements from this list.
E remove(int index)	It is used to remove the element present at the specified position in the list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of the specified comparator.
int size()	It is used to return the number of elements present in the list.

ArrayList Program

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList list=new ArrayList(); Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
System.out.println(list);

//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

LinkedList

It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized.

boolean add(E e)	It is used to append the specified element to the end of a list.
void add(int index, E element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
E element()	It is used to retrieve the first element of a list.
E get(int index)	It is used to return the element at the specified position in a list.
E getFirst()	It is used to return the first element in a list.
E getLast()	It is used to return the last element in a list.

Program

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList al=new LinkedList();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

add()	It is used to append the specified element in the given vector.
addElement()	It is used to append the specified component to the end of this vector. It increases the vector size by one.
capacity()	It is used to get the current capacity of this vector.
clear()	It is used to delete all of the elements from this vector.
elementAt()	It is used to get the component at the specified index.
get()	It is used to get an element at the specified position in the vector.
indexOf()	It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
lastElement()	It is used to get the last component of the vector.
remove()	It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.
set()	It is used to replace the element at the specified position in the vector with the specified element.

Vector

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector v=new Vector();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

add(E e)	It is used to add the specified element to this set if it is not already present.
clear()	It is used to remove all of the elements from the set.
contains(Object o)	It is used to return true if this set contains the specified element.
isEmpty()	It is used to return true if this set contains no elements.
iterator()	It is used to return an iterator over the elements in this set.
remove(Object o)	It is used to remove the specified element from this set if it is present.
size()	It is used to return the number of elements in the set.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet set=new HashSet();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

boolean add(E e)	It is used to add the specified element to this set if it is not already present.
E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.
E pollLast()	It is used to retrieve and remove the highest(last) element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void clear()	It is used to remove all of the elements from this set.

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet set=new TreeSet();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

HashMap

It stores the data in (Key, Value) pairs, and you can access them by an index of another type (e.g. an Integer). One object is used as a key (index) to another object (value). If you try to insert the duplicate key, it will replace the element of the corresponding key.

HashMap is similar to HashTable, but it is unsynchronized. It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. This class makes no guarantees as to the order of the map. To use this class and its methods, you need to import **java.util.HashMap** package or its superclass.

void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
V get(Object key)	This method returns the object that contains the value associated with the key.
V replace(K key, V value)	It replaces the specified value for a specified key.

```
import java.util.HashMap;
 public class GFG {
     public static void main(String[] args)
    {        // Create an empty hash map by declaring object     // of string and integer type
        HashMap<String, Integer> map = new HashMap<>();
        // Adding elements to the Map
        // using standard put() method
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);
        // Print size and content of the Map
        System.out.println("Size of map is:- "+ map.size());
         // Printing elements in object of Map
        System.out.println(map);
         // Checking if a key is present and if// present, print value by passing   // random element
        if (map.containsKey("vishal")) {
             // Mapping
            Integer a = map.get("vishal");
             // Printing value for the corresponding key
            System.out.println("value for key"
                               + " \"vishal\" is:- " + a);
        }
    }
}
```

HashTable

The **Hashtable** class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

Features of Hashtable

It is similar to HashMap, but is synchronized.
Hashtable stores key/value pair in hash table.
In Hashtable we specify an object that is used as a key, and the value we want to associate to that key.
The initial default capacity of Hashtable class is 11

```
import java.io.*;
import java.util.*;
class AddElementsToHashtable {
        public static void main(String args[])
        {                       // No need to mention the          // Generic type twice
                Hashtable<Integer, String> ht1 = new Hashtable<>();

                // Initialization of a Hashtable                   // using Generics
Hashtable<Integer, String> ht2= new Hashtable<Integer, String>();

                // Inserting the Elements                          // using put() method
                ht1.put(1, "one");
                ht1.put(2, "two");
                ht1.put(3, "three");
                ht2.put(4, "four");
                ht2.put(5, "five");
                ht2.put(6, "six");
                // Print mappings to the console
                System.out.println("Mappings of ht1 : " + ht1);
                System.out.println("Mappings of ht2 : " + ht2);
        }
}
```

```
Mappings of ht1 : {3=three, 2=two, 1=one}
Mappings of ht2 : {6=six, 5=five, 4=four}
```

TreeMap

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:
Java TreeMap contains values based on the key. It implements the
Java TreeMap contains only unique elements.
Java TreeMap cannot have a null key but can have multiple null values.
Java TreeMap is non synchronized.
Java TreeMap maintains ascending order.

```
import java.util.*;
public class TreeMap2 {
  public static void main(String args[]) {
   TreeMap<Integer,String> map=new TreeMap<Integer,Stri
ng>();
    map.put(100,"Amit");
    map.put(102,"Ravi");
    map.put(101,"Vijay");
    map.put(103,"Rahul");
    System.out.println("Before invoking remove() method");
    for(Map.Entry m:map.entrySet())
    {
      System.out.println(m.getKey()+" "+m.getValue());
    }
    map.remove(102);
    System.out.println("After invoking remove() method");
    for(Map.Entry m:map.entrySet())
    {
      System.out.println(m.getKey()+" "+m.getValue());
    }
    }
}
```

```
Before invoking remove() method
100 Amit
101 Vijay
102 Ravi
103 Rahul
After invoking remove() method
100 Amit
101 Vijay
103 Rahul
```

Iterator

In Java, an **Iterator** is one of the Java cursors. **Java Iterator** is an interface that is practiced in order to iterate over a collection of Java object components entirety one by one.

The Java Iterator is also known as the **universal cursor** of Java as it is appropriate for all the classes of the Collection framework. The Java Iterator also helps in the operations like READ and REMOVE. When we compare the Java Iterator interface with the enumeration iterator interface, we can say that the names of the methods available in Java Iterator are more precise and straightforward to use.

Advantages of Java Iterator

The user can apply these iterators to any of the classes of the Collection framework.

In Java Iterator, we can use both of the read and remove operations.

If a user is working with a for loop, they cannot modernize(add/remove) the Collection, whereas, if they use the Java Iterator, they can simply update the Collection.

The Java Iterator is considered the Universal Cursor for the Collection API.

The method names in the Java Iterator are very easy and are very simple to use.

Java Iterator Methods

The following figure perfectly displays the class diagram of the Java Iterator interface. It contains a total of four methods that are:

hasNext()

next()

remove()

forEachRemaining()

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

public class JavaIteratorExample {
   public static void main(String[] args)
   {
      ArrayList<String> cityNames = new ArrayList<String>();

      cityNames.add("Delhi");
      cityNames.add("Mumbai");
      cityNames.add("Kolkata");
      cityNames.add("Chandigarh");
      cityNames.add("Noida");

      // Iterator to iterate the cityNames
      Iterator iterator = cityNames.iterator();

      System.out.println("CityNames elements : ");

      while (iterator.hasNext())
         System.out.print(iterator.next() + " ");
         System.out.println();
   }
}
```

Comparator

Java Comparator interface is used to order the objects of a user-defined class.
This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).
It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.