BLOCK 3
MULTITHREADING AND I/O

BLOCK 3 INTRODUCTION

A multi-threaded program contains two or more parts that can run concurrently. Each part is known as a thread and can handle a different task at the same time, this makes optimal use of the available resources specially when a computer has multiple CPUs. Java is a multi-threaded programming language. Using Java you can develop multi-threaded program. Java multithreading provide features of managing thread priorities, interthread communication, and concurrency control etc.

Java provide classes for working with different types of numbers such as int, float, double, handling character, and working with strings.

We use Java I/O (Input and Output) to process the input and produce output. In Java concept of a stream is used to make I/O operation fast. The java.io package contains the classes required for input and output operations.

A collection represents a group of objects, known as its elements. The Java API has a basic set of interfaces like Collection, Set, List, Vector, Map etc and classes those implement these interfaces.

This block is on multithreading, string handling, I/O stream, and Java APIs. This block consists of 4 units and is organised as follows:

Unit 9: This unit explains concept of multithreading , java thread model and thread life cycle. Further in this unit you will learn how to create threads in java. Also, in this unit thread priority, synchronization and use of multithreading in problem solving are explained.

Unit 10: This unit explains how to handle Number, Character and string in Java using respective classes and methods. Also, in this unit concepts of autoboxing and unboxing are explained .

Unit 11: This unit explains Java I/O classes and interfaces. Also, File class and some of its methods are explained in this unit. Further this unit explains use of stream classes and serialization.

Unit 12: This unit explains collection hierarchy and use of Date class, Hashmap class, Vector class, and Stack class. Also, Set interface and Map interface are explained in this unit.

UNIT 9 MULTITHREADING

Structure **Page No.**

9.0 Introduction 219
9.1 Objectives 219
9.2 Multithreading 220
9.3 Java thread Model 222
9.4 Creating Threads in Java 222
9.4.1 The Thread Class 223
9.4.2 The Main Thread 223
9.4.3 Creating Child Threads 224
9.5 Thread Life Cycle 227
9.6 Creating Multiple Threads 228
9.7 Using isAlive( ) and join( ) 229
9.8 Thread Priority 231
9.9 Synchronization 233
9.10 Interthread Communication 236
9.11 Suspending, Resuming, and Stopping Threads 239
9.12 Obtaining a Thread State 240
9.13 Using Multithreading in problemsolving 240
9.14 Summary 242
9.15 Solutions/ Answer to Check Your Progress 242
9.16 References/Further Reading 245

9.0 INTRODUCTION

Multithreading is one of the important features provided by java programming language. A thread represents a single sequence of executionthat can independently execute in an application. Uses of threads in programs are good for maximizing the resources utilization of the system on which applications are running. Multithreaded programming is very useful in developing many types of applications, such as network applications, gaming applications and Internet applications. In this unit, you will learn the concept of multithreading and how they are used in applications development using java. This unit also explains the working of threads, threadproperties, thread synchronization, and interthread communication.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- understand concepts of multithreading,
- describe the java thread model,
- create and use threads in your programs,
- write programs to describe how to set the thread priorities,
- use the concept of thead synchronization in programming, and
- use inter-thread communication in programs.

9.2 MULTITHREADING

Most modern computer systems have the capability of performing more than one job. It is like having more than one computer to perform your jobs. Conceptually, in basic terms, this is called multitasking.

Multitasking can be performed either at the Process level, or it can be executed at Thread level.

- Process based multitasking: A process is a program in execution. This type of multitasking lets your computer run two or more programs simultaneously.

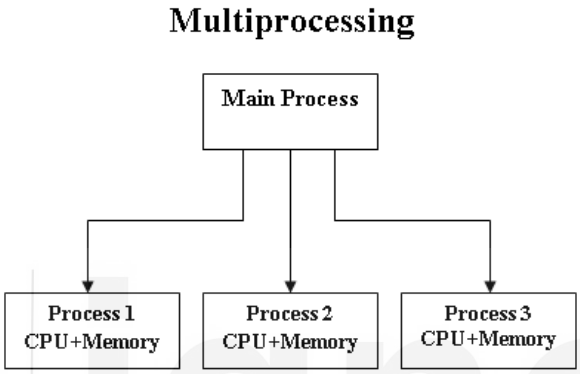


Figure 9.1(a): Multiprocessing

- Thread based multitasking: Thread is a part of a single program that can run concurrently. In thread-based multitasking, one program can perform two or more tasks simultaneously.

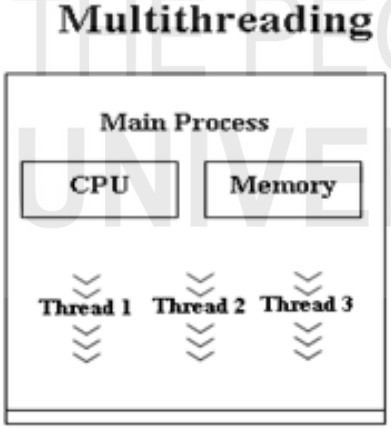


Figure 9. 1(b): Multithreading

Parts of a Program

Any computer program in memory has four different types of spaces. These are stack,heap, variable- space and program code space. These are allocated by the memory management module of the operating system based on the optimization algorithm used by the operating system.

- The stack is used for static memory allocation.
- The heap is used for dynamic memory allocation.
- The variable space is used for storing all the variables declared.
- The code space includes all the instructions written in the code.

Heavy-weight and Light-weight Processes

A program will occupy all four spaces as a unit. In process-based multitasking, if another program resides in the memory, it will occupy all four spaces as a separate unit. That is why process-based multitasking is known as heavy-weight multitasking. On the other hand, a thread is a part of a process. All threads of a process use the same four spaces that the program is allocated by overlaying the space with their variables and code space. Since they occupy the same space, they are called light-weight processes, as can be seen in figure 9.2.

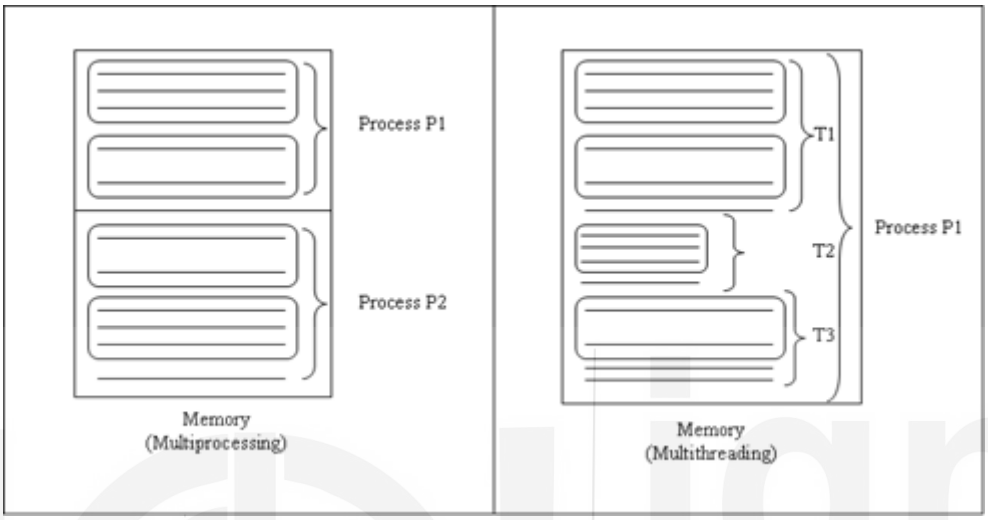


Figure 9. 2: Multiprocessing and Multithreading

As apparent from the figure 9.2, process-based multitasking uses individual memory spaces, whereas the threads use the single memory space of a process, justifying the name light-weight processes.

☞Check your Progress-1

1) What are different memory spaces occupied by a program when it is in execution?

……

……

……

……

2) Why is a thread called a light-weight process?

……

……

……

……

3) What are the advantages of multithreading?

……

……

……

……

9.3 JAVA THREAD MODEL

A thread has the following three components:

- CPU –Thread Class is designed to encapsulate the virtual CPU in it. When a thread is constructed, the code and data that define the thread specified by the object are passed to the constructor of the thread class.
- Data – Data may/may-not be shared by multiple threads
- Code – Shared by multiple threads, independent of data.

Java is a very strong language as far as support of multithreading is concerned. It has a very rich set of methods available for muthithreaded programming. Further in this unit, you will learn how to write programs using concepts of multithreading in java.

In java, a thread can be in various states. These states are identified as :

- New state: The main thread starts when the program starts executing.
- Ready State: It is ready to occupy CPU as soon as possible by the scheduler.
- Running State: The Thread is in running state when it has the CPU. It may leave this state for doing I/O or by executing wait(),sleep() methods, etc.
- Blocked State: A thread moves to blocked state for several reasons. Few notable ones are performing I/O, acquiring locks, resources etc.
- Dead State: A thread moves to dead state when it completes its execution, i.e., the execution of main() method completes.

A thread life cycle can be depicted by combining the thread states, as shown in figure 9.3.

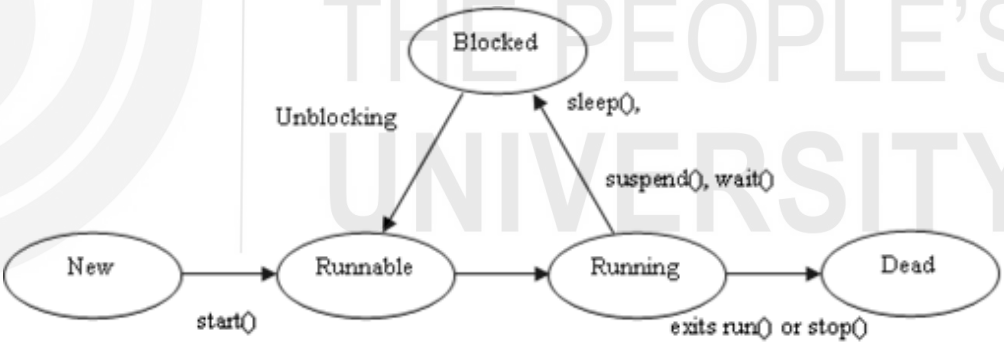


Figure 9.3: Thread Life Cycle

9.4 CREATING THREADS IN JAVA

There are two ways to create a thread in java. The java Multithreading system is built upon the following:

- Thread Class and its methods
- Interface Runnable

The **java.lang.Thread** class allows you to create and control child thread. The Thread class encapsulates the thread of CPU. The thread reference thus obtained helps to know the status of the main thread.

9.4.1 The Thread Class

The Thread class defines several constructors and methods to create and manage threads

The Thread Constructors

A few Thread class constructors are listed below:

Constructor	Description
Thread()	This allocates a new **Thread** object.
Thread(String name)	This constructor takes the name of the child-thread as the parameter and constructs the Thread object/instance
Thread(Runnable target)	This allocates a new **Thread** object
Thread(Runnable target, String name)	This allocates a new **Thread** object.

There are several other constructors with ThreadGroup Parameter to create thread objects, but it is beyond the scope of our discussion in this course.

The Thread Class Methods

Thread Class defines several methods to manage the threads. Some commonly used methods of Thread class are listed below.

Method	Description
getName()	Gets the name of the thread
getPriority()	Gets the priority of the thread
isAlive()	Determining if the Thread is still Alive
join()	Waits for a thread to terminate
run()	Entry point for thread
sleep()	Suspends a thread for a period
start()	Start a thread by calling its run method.

9.4.2 The Main Thread

Whenever a java program runs, one thread begins running immediately. This thread is called the main thread of the program. All child threads are born out of the main thread. Also, all the child threads will finish their execution before the main thread. The main thread is always the last thread to complete its execution. When the main thread finishes, the program terminates.

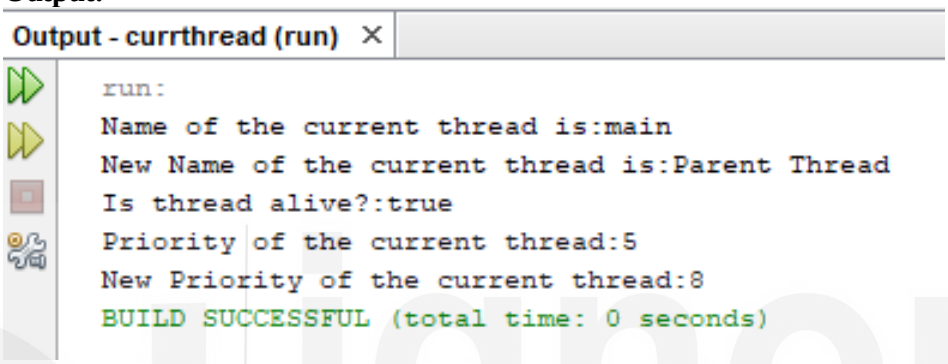
The main thread is created automatically by the system when your program starts running. It could be controlled by the thread reference. The reference is obtained by the currentThread() method, which is a public, static function of the Thread Class.

Now let us see a program that demonstrates the working ofthe main thread.

```
import java. lang.Thread;
public class Currthread
{
```

```
  public static void main(String[] args)
  {
    Thread t=Thread.currentThread();
    System.out.println("Name of the current thread is:"+t.getName());
    t.setName("Parent Thread");
    System.out.println("New Name of the current thread is:"+t.getName());
    System.out.println("Is thread alive?:"+t.isAlive());
    System.out.println("Priority of the current thread:"+t.getPriority());
    t.setPriority(8);
    System.out.println("New Priority of the current thread:"+t.getPriority());
  }
}
```

Output:



In the above program, you can see the use of methods setName(), which is used to set the name of that on which it is called. Also, methods setPriority() and getPriority() are used in this program. For thread priority, we will learn in the later section of the unit.

9.4.3 Creating Child Threads

As mentioned earlier , there are two ways to create thread instances.

- By implementing the Runnable interface
- By extending the Thread class

Now let us see these ways of creating threads one by one.

9.4.3.1 First Way to create thread by implementing Runnable interface

To implement Runnable interface, a class needs to implement a single method called run().

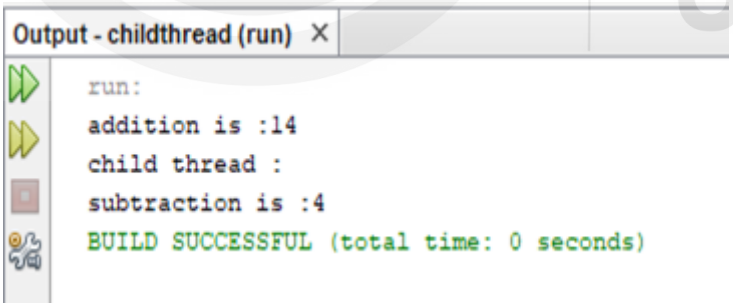
```
public void run()
{
……
}
```

Inside run() method you can define the code that you want from your child thread to execute.The run() method can call other methods, use other classes, declare variables, just like the main thread. Basically the run() method establishes the entry point for another concurrent thread of execution within your program.

After you create a class that implements Runnable interface, you will instantiate an object of Thread from that class. An example of creating a threading by implementing the Runnable interface is given below.

```
import java.lang.Thread;
class mythread implements Runnable
{
    Thread t;
    String name;
    static int a, b;
    mythread(String n,int a1,int b1)
    {
        a=a1;
        b=b1;
        t=new Thread(this,n);//Child Thread created
        t.start();//Child thread now ready  to  run
    }
    public void run()
    {
        int c;
        c=a-b;
        System.out.println("child thread :");
        System.out.println("subtraction is :"+c);
    }
}
public class Childthread
{
    public static void main(String[] args)
    {
        int a=9;int b=5;
        mythread t1=new mythread("child thread",a, b);
        System.out.println("addition is :"+(a+b));
    }
}
```

Output:



9.4.3.2 Second way to create child threads is to extend the Thread class itself.

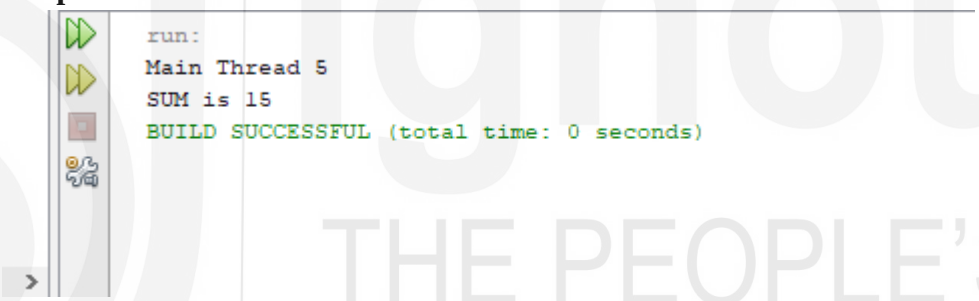
The Thread class defines several constructors.

The constructor **Thread**(String name) uses the child-thread name as the parameter. Using the constructor, the new thread is created. The child thread created does not run till a call to start() method is called.

The start() method makes a call to run() method(This is done automatically). Once the call to run() is made, the child thread is ready to run and is put in the scheduling queue.

```
//A program in java to extend Thread class.
public class myThread extends Thread
{
  static int a,b;
  String n;
  myThread(String n1, int a1, int b1)
  {
    super(n1);
    a=a1;
    b=b1;
    start();
  }
  public void run()
  {
    System.out.println("SUM is " + (a+b));
  }
  public static void main(String args[])
  {
    myThread i1 = new myThread("Thread is ",10,5);
    System.out.println("Main Thread " + (a-b));
  }
}
```

Output:



☞Check your Progress-2

1) Discuss the difference between Runnable and the Running state of the thread.

……

……

……

2) Create a child thread by implementing the Runnable interface wherein the child thread does string concatenation, and the main thread changes the string to uppercase.

……

……

……

3) Repeat the above program by making child thread by extending the thread class.

……

……

……

9.5 CHILD THREAD LIFE CYCLE

A newly born/child thread is created when the constructor is called. The child thread starts running only when a call to start() method is done. Calling start() method places the child thread in a runnable state. This means, it is ready to execute the run() method and is available for scheduling. This does not mean that the thread runs immediately. It executes the run method only after the CPU is available .

- Ready state : After the call to start() method is made, the thread is in ready state. It is ready to occupy CPU as soon as it is possible by the schedular.
- Running State: The Thread is in running state when it has the CPU and is executing its run() method. It may leave this state for doing I/O or by executing wait(),sleep() methods, etc.
- Blocked State : A thread moves to blocked state for several reasons. Few notable ones are performing I/O, acquiring locks, resources etc.
- Dead State : A thread moves to dead state when it completes its execution, i.e., the execution of run() method.

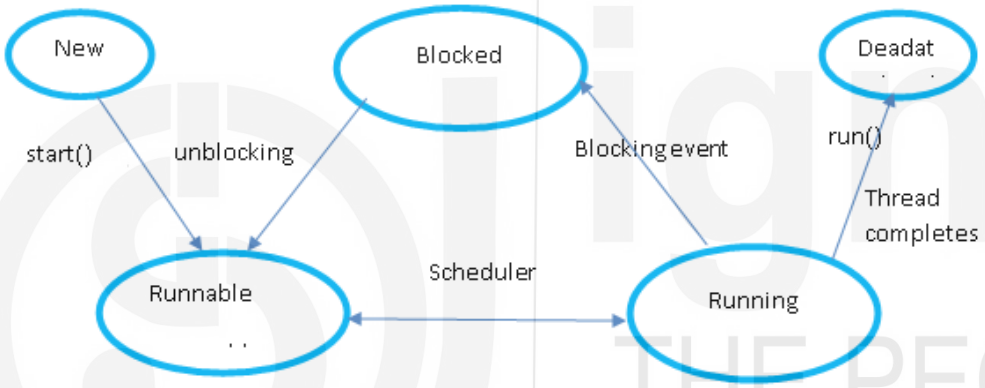


Figure 9.4: Life cycle of Child Thread

Java threads follow the **preemptive scheduling** where a thread with lower priority relinquishes the CPU when preempted by a thread of higher priority. The model of a preemptive schedular is such that many threads might be runnable, but only one is running.

A thread can relinquish the CPU for a variety of other reasons besides being preempted by a higher priority thread:

- The thread code can execute a Thread.sleep() method call deliberately asking the thread to pause for a fixed period.
- The thread might have to perform I/O, access a shared resource,access a network resource, and cannot continue until the resource becomes visible.

All threads that are runnable are kept in a pool according to their priority. When CPU becomes available, the thread with the highest priority is given the CPU.

Since java threads are not time-slicedtherefore you mustensure that the code for your threads gives other threads a chance to execute from time to time.This is achieved by calling the sleep call at various intervals or by code controlled programming.

Thread.sleep() method can pause a thread for a specific period and are interruptible. The sleep() is a static method in the Thread class. Because it operates on the current

thread, it is called Thread. sleep(int milliseconds) where milliseconds is the time for which the thread is made inactive.

When a child thread completes the run() method's execution, it terminates and then cannot run again.

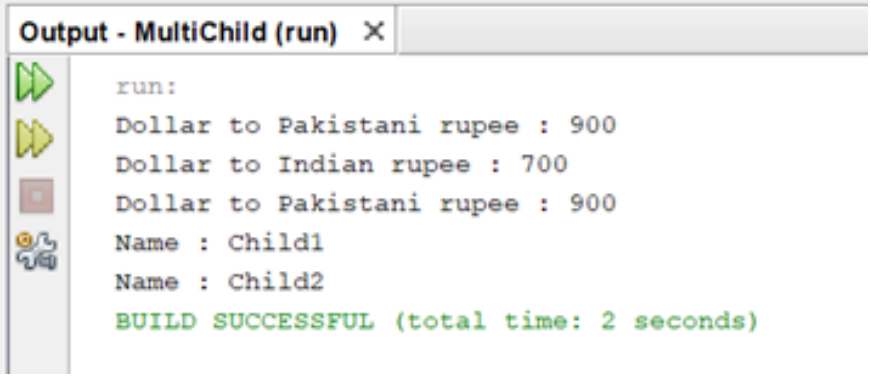
9.6 CREATING MULTIPLE CHILD THREADS

More than one child thread can be created and can be programmed to perform different tasks. In the following example, the main thread converts the dollar to the Indian rupees. The child thread1 converts the dollar to the Pakistani rupee, and childhread2 converts to the Nepali rupee.

Note: The function *getName()* is used to distinguish between the child threads.

```
import java.lang.Thread;
public class MultiChild implements Runnable
{
    String name;
    static int dollar,rupee;
    Thread t;
    MultiChild(String n,int d)
    {
        t=new Thread(this,n);
        dollar=d;
        rupee=0;
        t.start();
    }
    public void run()
    {
        if(t.getName().equals(name))
        {
            System.out.println("Name child1: "+t.getName());
            rupee = dollar*80;
            System.out.println("Dollar to Nepali rupee : "+ rupee);
        }
        else
        {
            System.out.println("Dollar to Pakistani rupee : "+ dollar*90);
            System.out.println("Name : "+t.getName());
        }
    }
    public static void main(String args[])
    {
        MultiChild t1 = new MultiChild("Child1",10);
        MultiChild t2 = new MultiChild("Child2",10);
        System.out.println("Dollar to Indian rupee : "+ dollar*70);
    }
}
```

Output:



9.7 USING ISALIVE() AND JOIN()

There are some methods that help to control the working of a thread.

Testing a ThreadState: A thread can be in an unknown state. Using the method *isAlive()*,you can determine if a thread is still viable(Running/Runnable/Blocked). This method returns true for the thread that has started but has not completed its task(i.e still running).

public final boolean isAlive()

This method returns true if this thread is alive; false otherwise.The following java program shows the use of isAlive() method.

//Program

```
class RunnableClass implements Runnable
{
    public void run()
    {
        for(int i =0; i <3; i++)
        {
            System.out.println(Thread.currentThread().getName()+" i - "+ i);
            try
            {
                Thread.sleep(100);
            }
            catch(InterruptedException e)
            {
                System.out. println(“ Exception Caught”);
                e.printStackTrace();
            }
        }
    }
}
public class Example
{
    public static void main(String[] args)
    {
        Thread t1 =newThread(newRunnableClass(),"t1");
        Thread t2 =newThread(newRunnableClass(),"t2");
        t1.start();
        t2.start();
```

```
        System.out.println("t1 Alive - "+ t1.isAlive());
        System.out.println("t2 Alive - "+ t2.isAlive());

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());
        System.out.println("Processing finished");
}
}
```

Output:

```
t1 Alive-true
t2 Alive-true
Processing finished
t3 i -0
t1 i -0
t2 i -0
t3 i -1
t1 i -1
t2 i -1
t2 i -2
t1 i -2
t3 i -2
```

Joining a Thread: The join() method causes the calling thread to wait until the thread on which the join method is called terminates.

public final void join() throws InterruptedException

This method waits until the thread on which it is called terminates. There are three overloaded join functions.

- **public final void join()** - Waits indefinitely for this thread to die.
- **public final void join(long milliseconds)** - Waits at most milliseconds forthis thread to die. A timeout of 0 means to wait forever.
- **Public final void join(long milliseconds, int nanoseconds)** - Waits at most milliseconds plus nanoseconds for this thread to die.

The following is a java program that shows the use of the join() method.

```
//program
class RunnableClass implements Runnable
{
    public void run()
    {
        for(int i = 0; i < 5 ; i++)
        {
            System.out.println(Thread.currentThread().getName() + " i - " + i);
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class JoinExample
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new RunnableClass(), "t1");
        Thread t2 = new Thread(new RunnableClass(), "t2");
        t1.start();
        t2.start();

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        try
        {
            t1.join();
            t2.join();

        }
        catch (InterruptedException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        System.out.println("Processing finished");
    }
}
```

Output:

```
t1 Alive - true
t2 Alive - true
t2 i - 0
t1 i - 0
t1 i - 1
t2 i - 1
t1 i - 2
t2 i - 2
t1 Alive - false
t2 Alive - false
Processing finished
```

9.8 JAVA PRIORITY ENVIRONMENT

Java threadsoperate in a Preemptive/priority- based scheduling environment. This means that a thread with higher priority forces/preempts a thread with lower priority to release the CPU. This is called context switching. Two threads of equal priority are further scheduled by the secondary scheduling algorithm of the operating system. The default priority assigned to the thread is 5(NORM-PRIORITY). The range lies between MIN-PRIORITY(value=1) and MAX-PRIORITY(value =10). The priority of a thread is an int value . The priority can be set using the method:

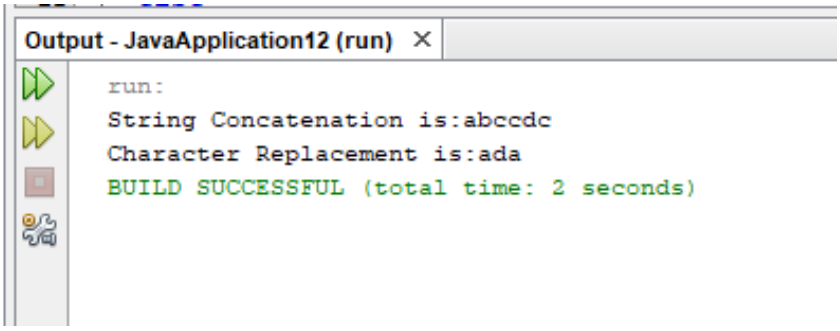
final void setPriority(int level)

where the level specifies the new priority setting for the calling method. The value of the level must be in range 1 to10.

The following program demonstrates the use of setPriority(int level) method.

```
//Program
import java.lang.String;
public class Priority implements Runnable
{
    String name;
    Thread t;
    String s3, s4;
    Priority(String n, String s1, String s2, int prio)
    {
        name = n;
        s3 = new String(s1);
        s4 = new String(s2);
        t = new Thread(this, n);
        t.setPriority(prio);
        t.start();
    }

    public void run()
    {
        if(t.getName().equals("ch1"))
        System.out.println("String Concatenation is:" + (s3+s4));
        else
        System.out.println("Character Replacement is:" + s4.replace('c','a'));
    }
    public static void main(String args[])
    {
        String s1 = new String("abc");
        String s2 = new String("cdc");
        Priority th1 = new Priority("ch1", s1, s2, 7);
        Priority th2 = new Priority("ch2", s1, s2, 2);
    }
}
```

Output:



☞Check your Progress-3

1) Enlist the ways in which a child thread is created. Which one is the preferred method and why?

……

……

2) Discuss the macros for priorities within Java thread environment.

……

……

……

……

3) Discuss the utility of isAlive() method in the Java thread environment

……

……

……

……

9.9 SYNCHRONIZATION OF THREADS

When two or more threads want access to the shared resources, they need some way to ensure that the resource is used by only one thread that has exclusive access to it at a time. The synchronization keyword helps to achieve this. Synchronization in java is achieved by using the concept of object lock flag.

The communication with the lock flag is achieved via the keyword I and allows exclusive access to code with shared data. When the thread reaches the synchronized statement, it examines the object passed as an argument. The object is examined to obtain the lock flag from it before continuing. After taking the lock flag from the object, the thread continues executing the shared code(if not, the thread waits in the object lock flag pool). When all other threads try to execute the same synchronized statement, they also try to obtain the lock flag from the object, which is not present. These threads then join a pool of waiting threads on object's lock pool.When the previous thread finishes executing the synchronized method, the lock flag is returned and then a thread waiting on lock pool is given the flag, and then this thread starts executing the synchronized statement.

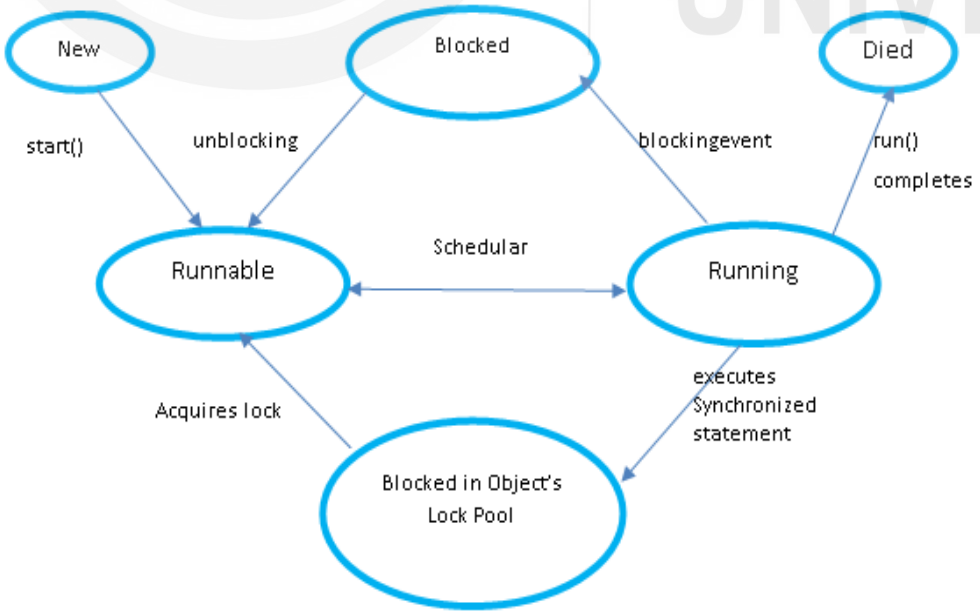
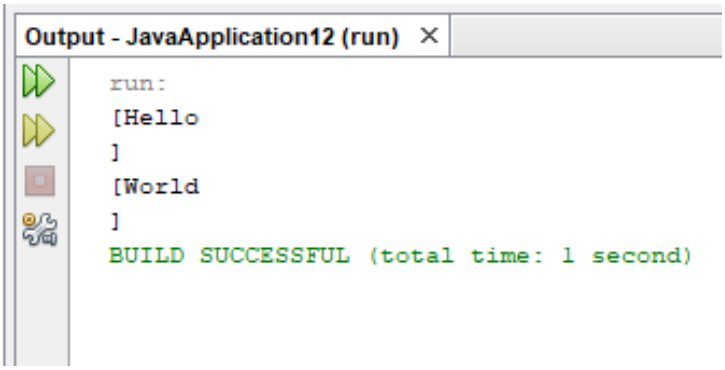


Figure 9.5: States of Thread through its life cycle with synchronized statement

The following java program illustrates the concept of synchronization.

```
//program
class Callme
{
    void call(String msg)
    {
        System.out.println("[" + msg);
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
class Caller implements Runnable
{
    Callme target;
    String msg;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run()
    {
        synchronized(target)
        {
            target.call(msg);
        }
    }
}
class Synch
{
    public static void main(String[] agrs)
    {
        Callme target = new Callme();
        Caller obj1 = new Caller(target, "Hello");
        Caller obj2 = new Caller(target, "World");
    }
}
```

Output:

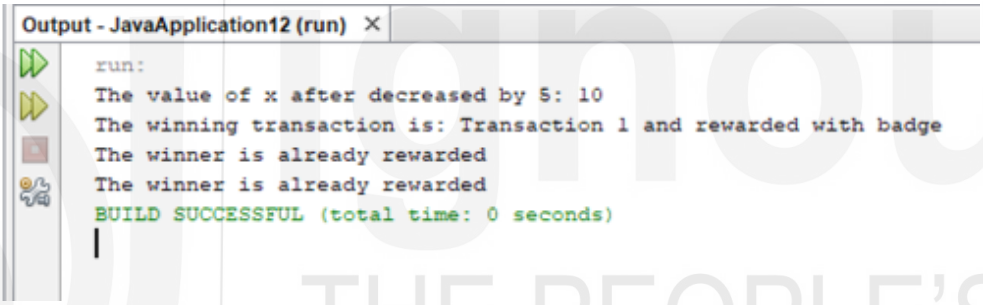


Let us see one more java program in which each transaction decrement the value of given object say x by -5. Here it is assumed 3 transactions t1,t2 and t3 and 3 child threads. They starts simultaneously with the same priority, and each one tries to decrements the value of x. The system tries to return the transaction, which completes its execution first.

```
//Program
class Decrement
{
  public void decrementValue(int x)
  {
    System.out.println("The value of x after decreased by 5: "+(x-5));
  }
 }
class Decrementar implements Runnable
{
  static boolean winner=false;
  String name;
  int x;
  Thread t;
  Decrement d;
  Decrementar(Decrement d1,String n,int num)
  {
    this.d=d1;
    name=n;
    x=num;
    t=new Thread(this,name);
    t.start();
  }
  public void run()
  {
    synchronized(d)
    {
      if(!winner)
      {
       d.decrementValue(x);
       System.out.println("The winning transaction is: "+name+" and rewarded with
      badge");
       winner=true;
      }
```

```
        else
        {
         System.out.println("The winner is already rewarded");
        }
      }
   }
}
public class decrementTransaction
{
   public static void main(String  args[])
   {
      Decrement d=new Decrement();
      new Decrementar(d,"Transaction 1",15);
      new Decrementar(d,"Transaction 2",11);
      new Decrementar(d,"Transaction 3",19);
   }
}
```

Output:



9.10 INTERTHREAD COMMUNICATION

Three final methods namely wait(),notify(), notifyAll() are provided by Java. lang. Object class for inter-thread communication. When a wait call is issued on object X by a thread , it pauses its execution until another thread issues a notify call on the same object X. The prerequisite for a thread to call wait/notify is that the thread must possess lock flag for that particular object. Hence, it is understood that wait and notify can only be called from within a synchronized block of the object.

When a thread executes a synchronized code on a particular object and calls a wait(), the thread is placed in wait-pool for that object. Additionally, the thread that calls wait releases that object's lock flag and goes to sleep until some other thread enters the same synchronized code on the same object and calls notify(). When a notify call is executed on a particular object, the first thread that calls wait on the same object is moved from the object's wait pool to the object's lock pool where threads stay until object's lock flag becomes available.

On the other hand, when notifyAll() call is issued, it moves all thread waiting on that object's wait-pool and puts them into the lock-pool. Only from the lock pool, when the highest priority thread obtains the lock flag, can continue running,where it left off when it was called to wait.

Method	Description
public final void wait() throws InterruptedException	waits until object is notified
public final void wait (long timeout)throws InterruptedException	waits for the specified amount of time
public final void notify()	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.
public final void notifyAll()	Wakes up all threads that are waiting on this object's monitor.

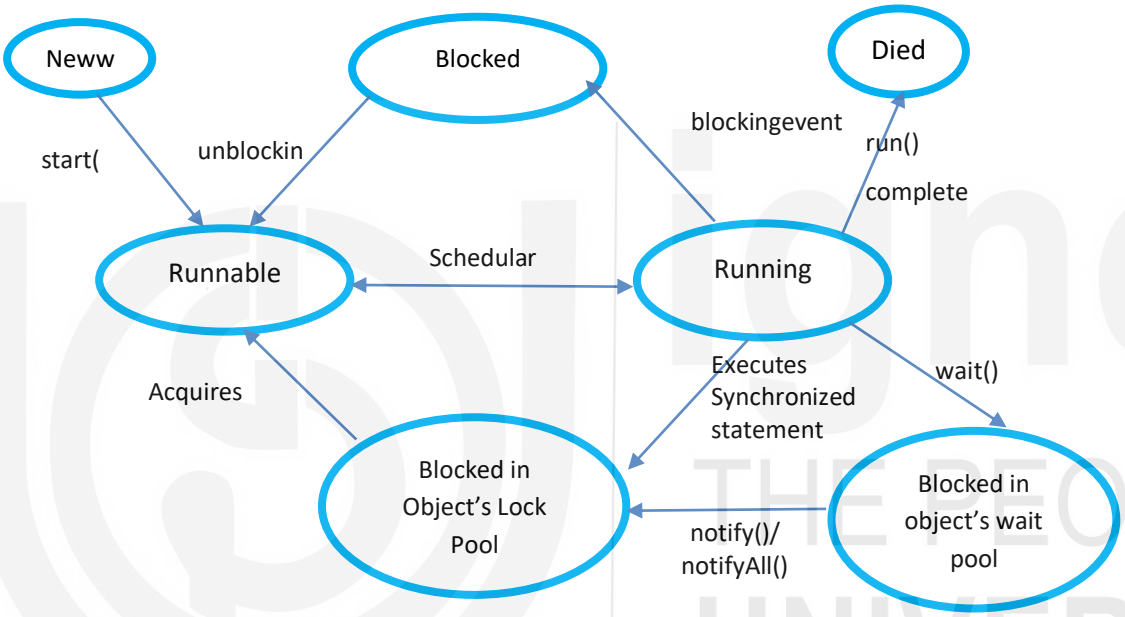


Figure 9.6: States of Thread through its life cycle with synchronized statement with wait/notify

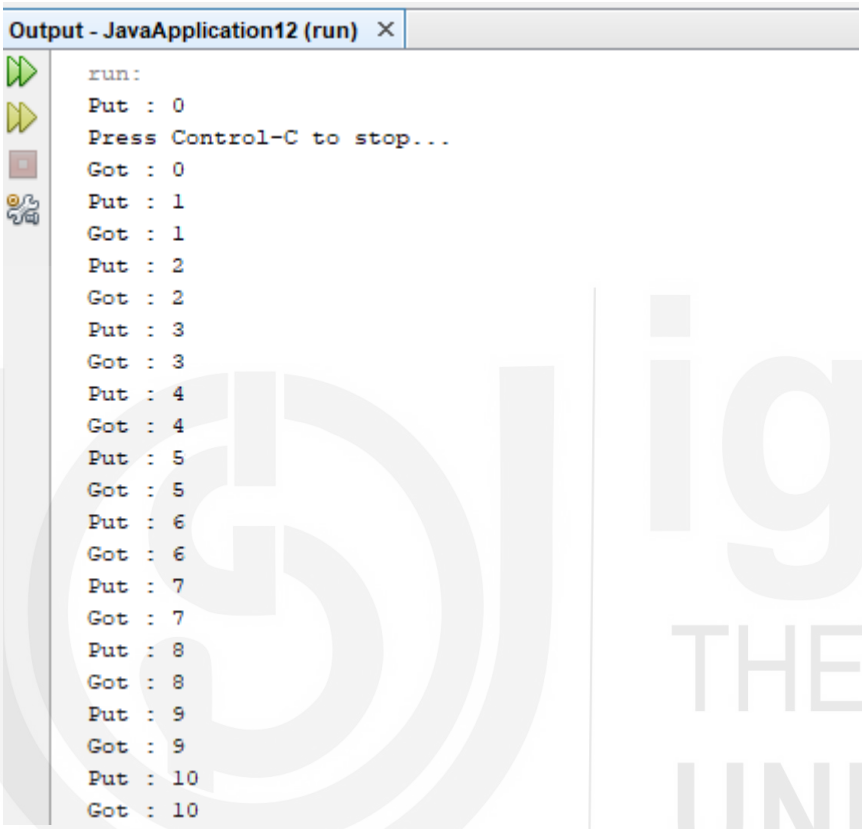
The following java program shows the concept of interthread communication using the producer-consumer problem for one unit of food. You are advised to execute this program and watch the output.

```
//Program
class Q
{
  int num;
  boolean valueSet = false;

  synchronized int get()
  {
    while(!valueSet)
    try
    {
     wait();
    }
    catch(InterruptedException e)
    {
     System.out.println("InterruptedException caught..!!");
    }
```

```
      System.out.println("Got : " + num);
      valueSet = false;
      notify();
      return num;
    }
    synchronized void put(int num)
    {
      while(valueSet)
      try
      {
       wait();
      }
      catch(InterruptedException e)
      {
       System.out.println("InterruptedException caught..!!");
      }
      this.num = num;
      valueSet = true;
      System.out.println("Put : " + num);
      notify();
    }
}
class Producer implements Runnable
{
 Q que;

 Producer(Q que)
 {
  this.que = que;
  new Thread(this, "Producer").start();
 }
 public void run()
 {
    int n = 0;
    while(true)
    {
       que.put(n++);
    }
 }
}

class Consumer implements Runnable
{
 Q que;
 Consumer(Q que)
 {
  this.que = que;
  new Thread(this, "Consumer").start();
 }
 public void run()
 {
    while(true)
    {
       que.get();
    }
 }
}

class PCFixed
```

```
{
   public static void main(String args[])
   {
      Q que = new Q();
      new Producer(que);
      new Consumer(que);
      System.out.println("Press Control-C to stop...");
   }
}
```

Output:



9.11 SUSPENDING,RESUMING AND STOPPING THREAD

These are deprecated methods by Java 2.0. While the suspend(),resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads to pause, restart and stop the execution of the thread.

suspend()

The suspend( ) method of the Thread class was deprecated by Java 2 several years ago. This was done because suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that the thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

resume()

The resume( ) method is also deprecated. It does not cause problems but cannot be used without the suspend( ) method as its counterpart.

stop( )

This method of the Thread class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

9.12 OBTAINING A THREAD STATE

As discussed earlier, a thread can exist in several states. You can get the current state of the thread by calling the getState() method defined by the Thread class.

Thread.State getState()

It returns a value of type Thread. The State indicates the state of thread at the time at which the call was made. State is an enumeration defined by Thread.

The following table enlists the values that can be returned by getState().

State-Value	Meaning
BLOCKED	A thread has suspended execution because it is waiting to acquire a lock
NEW	A thread is created and has not begun execution
RUNNABLE	A thread that is currently executing and has access to cpu
TERMINATED	A thread that has completed execution
TIMED-WAITING	A thread that has suspended execution for a specified period because it is waiting for some action to occur
WAITING	A thread that has suspended execution because it is waiting for some action to occur …e.g. wait() and join()

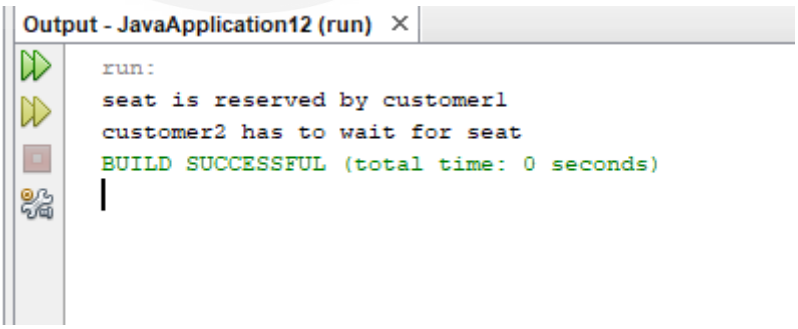
9.13 USING MULTITHREADING IN PROBLEM SOLVING

Before we conclude this unit, let us see a program that books a seat using synchronized statement.

```
// java program fora seat reservation in railway
//package railway;
class reservation
{
  public void reserve(String s)
  {
    System.out.println("seat is reserved by "+s);
  }
}
class customer implements Runnable
{
  reservation r;
  String name;
  Thread t;
  static boolean seat=true;
  public
```

```
  customer(reservation r1,String s)
  {
    r=r1;
    name=s;
    t=new Thread(this,s);
    t.start();
  }
  public void run()
  {
    synchronized(r)
    {
      if(seat)
      {
         r.reserve(name);
        seat=false;
      }
      else
      {
        System.out.println(name+" has to wait for seat");
      }
    }
  }
}
public class Railway1
{
  public static void main(String[] args)
  {
    reservation r11=new reservation();
    customer c1=new customer(r11,"customer1");
    customer c2=new customer(r11,"customer2");
  }
}
```

Output:



☞Check your Progress-4

1) Discuss the concept of synchronization.

……

……

……

……

2) Explain the utility of inter-thread communication.

……

……

……

3) Discuss any two problem scenarios where threads can be utilized.

……

……

……

9.14 SUMMARY

This unit described the concepts of multithreading in Java programming. The unit begins with the concept of the main thread, its creation and control methods. Different states of threads are discussed in detail, from its creation to the death state. This unit also explained the process of creating child threads using Thread class and Runnable interface. Subsequently the concept of preemptive java environment with thread's priority are explained. This unit demonstrated the use of the concept of synchronization, creating synchronous methods and inter-thread communication. The concept of object locks used to control access to shared resources also has been explained.

9.15 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

☞Check Your Progress-1

1) A program in execution is assigned memory by the operating system. The memory assigned is divided into the following spaces: stack, heap, variable space, and code space

- The stack is used for static memory allocation.
- The heap is used for dynamic memory allocation.
- The variable space is used for storing all the variables declared.
- The code space includes all the instructions written in code.

2) The thread process is a light- weight process as a child process continues to share the memory space of the parent thread. No separate memory space is allocated, unlike in fork, where a child process gets a replica of all in a separate memory space.

3) Advantages of Multithreading are they being light weight process. They perform multiple tasks while operating in the same memory space.

☞Check Your Progress 2

1) A thread is said to be in a runnable state when it is ready to run but has not been assigned the CPU and is waiting in the scheduling queue. The thread is said to be runnable when it is allotted the CPU and is executing.

2) Create a child thread by implementing the Runnable interface wherein the child thread does string concatenation, and the main thread changes the string to

uppercase.

```
import java.lang.Thread;
class mythread implements Runnable
{
   Thread t;
   String name;
   static String  a, b;
   mythread(String n, String  a1,String  b1)
   {
      a=a1;
      b=b1;
      t=new Thread(this, n);//Child Thread created
      t.start();//Child thread now ready  to  run
   }
   public void run()
   {
      System.out.println("child thread :");
      a.concat(b)
      System.out.println("After concatenation :"+a);
   }
}
public class Childthread
{
   public static void main(String[] args)
   {
      String one="hello"; String two="world";
      mythread t1=new mythread("child thread", one, two);
      System.out.println("Main Thread " );
      System.out.println("hello in uppercase looks like :"+toUpperCase(one));
   }
}
```

Output:

```
Child thread
helloworld
Main Thread
HELLO
```

3) Repeat the above program by making child thread by extending the thread class.

//Program

```
public class myThread extends Thread
{
  static String a,b;
  String n;
  myThread(String n1, String a1, String b1)
  {
    super(n1);
    a=a1;
    b=b1;
    start();
  }
  public void run()
  {
    System.out.println("child thread :");
    a.concat(b)
```

```
    System.out.println("After concatenation :"+a);
  }
  public static void main(String args[])
  {
    myThread i1 = new myThread("Thread is ","hello","world");
    System.out.println("Main Thread " );
    System.out.println("hello in uppercase looks like :"+toUpperCase("hello"));
  }
}
```

Ouptput:

Child thread
helloworld
Main Thread
HELLO

☞Check Your Progress-3

1) A child thread in Java can be created either by extending the thread class or by implementing the runnable interface. The second method by implementing the Runnable interface is preferred as it is felt that classes should be only extended if added functionality is required in the generalized version of the class. Hence, while making the child thread, generally, the thread class is usually not enhanced. Hence, keeping in the philosophy of the OOPs concepts, the Runnable interface method is added.

2) Java scheduling environment is priority based. A thread with higher priority can take the CPU control from a thread with a lower priority. By default, each thread is associated with a priority called normal priority. The priority value can vary between the MIN-PRORITY to MAX-PRIORITY. The macros associated with Java Priority environment are as follows:

   i) NORM-PRIORITY=Value 5

   ii) MIN-PRIORITY = Value 1

   iii) MAX-PRIORITY=Value 10

3) The isAlive() method helps a thread to know about the status of the called thread. The status returned is true if the called thread is still running else false. The syntax of the method *is boolean isAlive()*. This is generally used by the main thread to see the status of the child threads.

☞Check Your Progress-4

1) Synchronization is a mechanism that ensures exclusive access to the shared code. This is like having a write lock on the sharable data in the database environment. In java ‘s thread environment, it is achieved through the object’s lock flag. Only the thread that has the object’s lock flag can execute the sharable code exclusively. All other threads needing to execute the sharable code will have to wait till the thread with the lock flag finishes execution.

2) Interthread communication enables Threads to communicate or coordinate with each other with the help of methods like wait(), notify() and notifyAll(). These methods enable a thread to help get the waiting threads out of the object’s lock pool and other wait pools to ensure optimum utilization of CPU cycles.

3) The threads can effectively solve the following problems:

   i) Client -Server Simulation

ii) Database transactions

9.16 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" Vol. 2. Pearson Education, 2013.
- Prasanalakshmi, "*Advanced Java Programming*", CBS Publishers 2015
- Sagayaraj, Denis, Karthik and Gajalakshmi ,"*Java Programming – for Core and Advanced Users*", Universities Press 2018 .

ignou THE PEOPLE'S UNIVERSITY

UNIT 10 NUMBER, CHARACTER AND STRINGS

Structure **Page No.**

10.0 Introduction 246
10.1 Objectives 246
10.2 Number Class and its Methods 247
10.3 Character Class and its Methods 252
10.4 String Class and its Methods 254
10.5 StringBuilder Class and its Methods 263
10.6 StringBuffer Class and its Methods 268
10.7 Autoboxing and Unboxing 271
10.8 Summary 274
10.9 Solutions/ Answer to Check Your Progress 275
10.10 References/Further Readings 279

10.0 INTRODUCTION

In programming, we use several data types as per our needs. There are *wrapper* classes for each primitive data types in Java. These are used to create the object of that data type. Each primitive type in Java has a corresponding wrapper class. You will learn about Number Class (an abstract class) and its subclasses in this unit. Also, we use characters and strings frequently in programming. Many programming languages as C and C++, store strings as arrays of characters . However, Java represent strings as a separate object type, called String. You have already used objects of String type in your programs of the previous units of this course. In Java mainly, four classes are there that you can use when working with character data: Character, String StringBuilder, and StringBuffer. In this unit, you will learn different constructors, and operations like concatenation of strings, comparison of strings, insertion in a string etc. Also, you will learn about character extraction from a string, searching in a string, and conversion of different types of data into string form. Towards the end of this unit, you will learn about Autoboxing and Unboxing.

10.1 OBJECTIVES

After going through this unit you will be able to:

- explain fundamentals of characters and strings;
- use different constructors of String, StringBuilder, and StringBuffer clasess,
- use special operations on String objects,
- extract characters from a string,
- perform different operations such as string searching, comparison of strings etc.,
- data conversion using valueOf ( ) methods,
- use StringBuilder class and StringBuffer class and their methods, and
- use Autoboxing and Unboxing.

10.2 NUMBER CLASS AND ITS METHODS

So far you have used primitive data types for making variables and writing programs. Java primitive types are also called **literals which represent** fixed values in memory. However, sometimes you may need to use objects in place of primitives, for example, when you are using Object wrappers is generics where you have to use some Class, such as collections as List<Integer> or Map<Integer, Double> . Each of the primitive data types in Java platform has corresponding *wrapper* classes. The object of the wrapper class wraps (contains) its respective primitive data type which can be used to create an object of that data type. Figure 10.1 shows the Inheritance tree for primitive types.

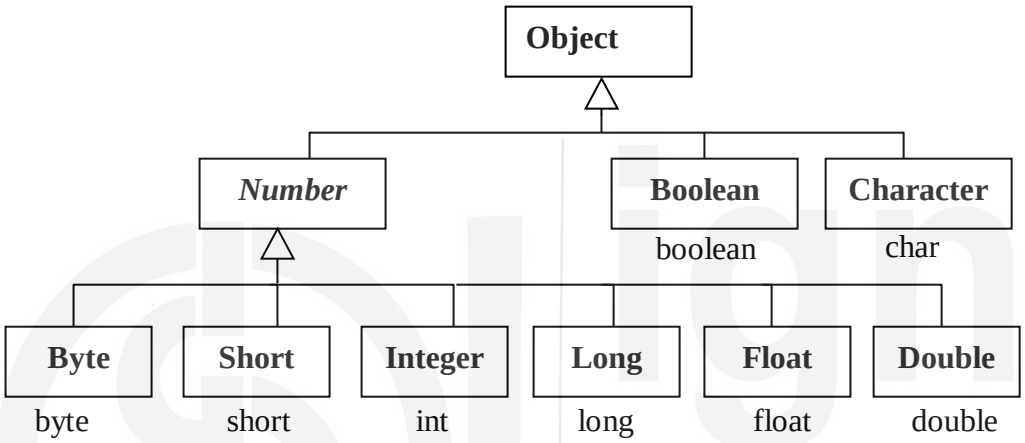


Figure 10.1: Inheritance tree for primitive types

Let us first learn about Number Class (an abstract class) and its subclasses. As you can see in figure 10.1 "each wrapper class has **Object** as a superclass. Byte, Short, Integer, Long Float and Double have **Number** as their direct superclass". For all the primitives there are classes which "wrap" the primitive in an object. This wrapping is often done by the compiler. Here important point to note is that Objects of the wrapper classes will have a default value *null,* because they are reference types, but as you know, primitive types default value can never be null because in case a primitive type is not assigned a value, the Java will assign default value to it. This important behaviour of default value will help you to choose between using a primitive type or a wrapper object. One more feature of primitive wrapper objects is that they are final, therefore, they are immutable.In Java when a wrapper objects value is modified, the Java compiler create a new object and subsequently reassign that object to the original.

Unlike the primitive data types, the non-primitive objects are created by the users in Java. Some examples include arrays, strings, classes, interfaces etc. In Java when the reference variables are stored, the variable is stored in the stack and the original object is stored in the heap. In object data type although two copies are created, but they both l points to the same variable in the heap, hence changes made to any variable reflects the change in both the variables. Unless using a wrapper class is necessary, as a programmer you should generally choose primitive types over wrapper classes. Primitive types are faster than Wrapper Objects, at the same time advantage of Wrapper Objects is that they can be *null.*

The Numbers Classes

When we are working with numbers, most of the time we use the primitive types such as int, float, double in our program. For example:

int age = 20;

float price = 15.5;

In case when you are using a primitive in place where an object is expected, Java compiler *boxes* your primitive in its wrapper class for you. Similarly, when you are using a number object where a primitive is expected, the compiler *unboxes* the object for you. Detail about *Outboxing* and *Inboxing* you will learn in section 10.6 of this unit.

The following are three reasons to use a Number object rather than a primitive:

1. When a method expects an object as an argument.
2. Using constants defined by the class, such as MIN_VALUE which lower bounds of the data type and MAX_VALUE which provide the upper bounds of the data type.
3. When you need the class methods which convert values to and from other primitive types. For example, when you need to convert to and from strings or to convert between number systems.

The following table 10.1 , lists the methods of all the subclasses of the Number class.

Table 10.1: Methods Implemented by all Subclasses of Number

Method	Description
byte byteValue() short shortValue() int intValue() long longValue() float floatValue() double doubleValue()	These are used to convert the value of Number object to the primitive data type returned.
int compareTo(Byte anotherByte) int compareTo(Double anotherDouble) int compareTo(Float anotherFloat)zx int compareTo(Integer anotherInteger) int compareTo(Long anotherLong) int compareTo(Short anotherShort)	It compares this Number object to the argument.
boolean equals(Object obj)	Used to determine whether this number object is equal to the argument. This method returns true if the argument is not null and it is an object of the same type and with the same numeric value.

Conversion Methods

Each Number class contains some methods used for converting numbers to and from strings and for converting between number systems. Table 10.2 lists the methods in

the Integer class for conversion. It is to note that other Number subclasses also have similar classes.

Table 10.2 : Conversion Methods of Integer Class

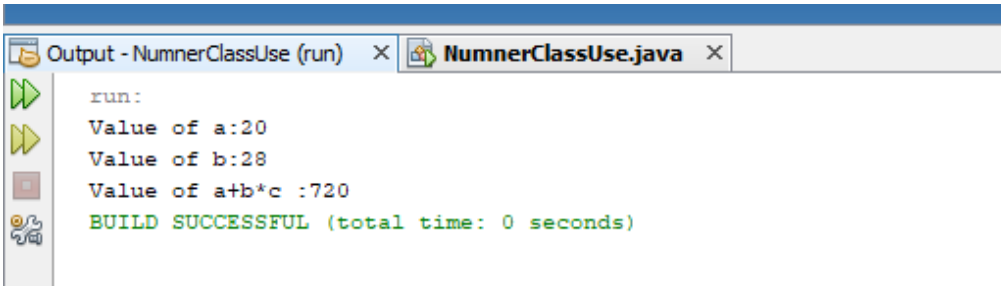
Method	Description
static Integer decode(String s)	Decodes a string into an integer. This method can accept string representations of decimal, octal, or hexadecimal numbers as input.
static int parseInt(String s)	It returns an integer
static int parseInt(String s, int radix)	It returns an integer value, of the given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16, respectively) numbers as input.
String toString()	Returns a String object representing the value of this Integer.
static String toString(int i)	Returns a String object representing the specified integer.
static Integer valueOf(String s)	It returns an Integer object which is holding the value of the specified string parameter(value).
static Integer valueOf (int i)	It returns an Integer object which is holding the value of the specified primitive.
static Integer valueOf(String s, int radix)	It returns an Integer object which is holding the integer value of the specified string representation, parsed with the value of radix as arguments.

Now let us see use of some methods given above in the Java program.

Example Program:

```
package numnerclassuse;
import static java.lang.Integer.parseInt;
import static oracle.jrockit.jfr.events.Bits.intValue;

public class NumnerClassUse
{
   public static void main(String[] args)
   {
    Integer I = 20; // I is object of Integer type
    Integer J = 25;
    String S = "28";
    int a, b, c, k ;
   a = intValue(I);
   b= parseInt(S); // getting integer value from String object
   c = intValue(J);
   System.out.println("Value of a:"+a);
   System.out.println("Value of b:"+b);
   k = a+b*c;
   System.out.println("Value of a+b*c :"+k);
   }
}
```

Output:



Formatting Numeric for Print Output

In the earlier program where you used *println* , you have seen that the use of the *println* methods for printing strings to standard output (*System.out*) where all numbers are converted to strings. Using println you can print out an arbitrary mixture of strings and numbers. To add more control over your print-output options, the Java programming language provides some other methods, especially when numbers are included.

The printf and format Methods

The java.io package includes PrintStream class. This class has two formatting methods which you can use to replace *println* method. These methods are format and printf, and are equivalent to one another.

The System.out that you have been using is a PrintStream object, therefore you can invoke PrintStream methods on System.out. Now you can try *format or printf* method anywhere in your code where you have previously been using println.

The syntax for format method and print methods of java.io.PrintStream is the same:
public PrintStream format(String format, Object... args)

In the above method the argument *format* is a string .This is used to specify the formatting to be used, and the second argument *args* is a list of the variables which are to be printed using that formatting. Here the notation Object... args is called *varargs*, that indicates that the number of arguments may vary. The format string contains the plain text as well as the *format specifiers*, which contains characters that format the arguments of Object... args.

Let us see an example:

```
System.out.format("The value of " + "the  integer variable is " +
    "%d, while the value of the " + "float variable is %f, " +
    "and the string is %s", intVar, floatVar, stringVar);
```

Example code:

```
int  var1= 520;
```

System.out.format("The value of var1 is: %d%n", var1);

Where %d specifies that the single variable is a decimal integer. The %n is a platform-independent newline character. The output is:

The value of var1 is: 520

The *format* method has the following syntax:

public PrintStream format(Locale l, String format, Object... args)

Locale is used for localization. If Locale l is null then no localization is applied. For example, to print numbers in the German system (in which in the English representation of floating point numbers a comma is used in place of the decimal place), you may use:

System.out.format(Locale.GERMAN, "The value of the float variable is: %f, and the " +

"value of the integer variable is: %d %n", floatVar, intVar);

The following table 10.3 has a list of some converters and flags that are used in the program given below

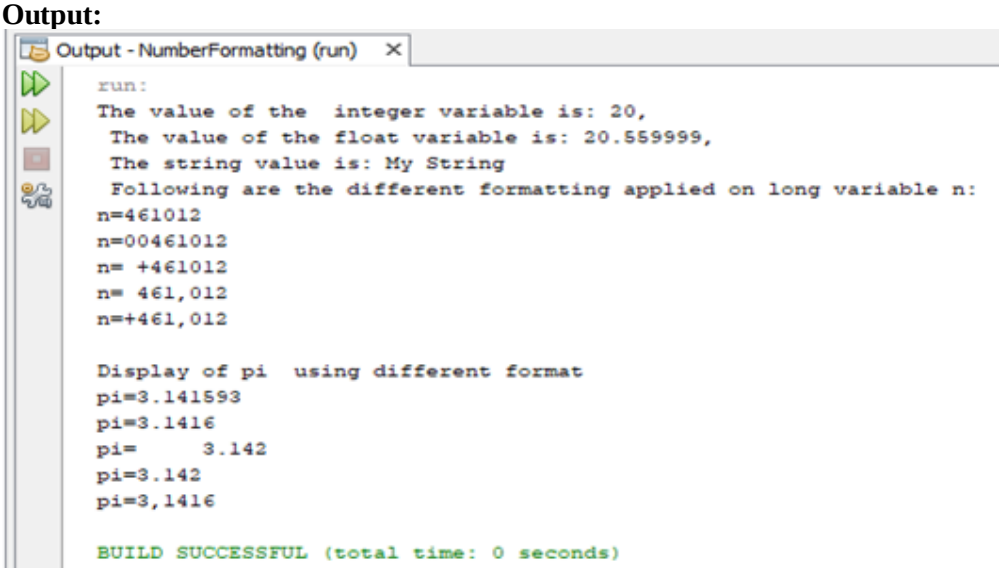
Table 10.3: Converters and Flags

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character
	08	Eight characters in width, with leading zeroes as necessary.
	+	Includes sign, whether positive or negative
	,	Includes locale-specific grouping characters.
	.3	Three places after decimal point.
	10.4	Ten characters in width, right justified, with four places after decimal point.

Example Pogram:

```
package numberformatting;
import java.util.Calendar;
import java.util.Locale;
public class NumberFormatting
{
    public static void main(String[] args)
    {
     long n = 461012;
     double pi = Math.PI;
     int intV= 20;
     float floatV = 20.56f;
     String stringV = "My String";
     System.out.format("The value of " + "the  integer variable is: " + "%d,%n The
    value of the " + "float variable is: %f,%n " + "The string value is: %s %n", intV,
    floatV, stringV);
     System.out.println(" Following are the different formatting applied on long
    variable n:");
     System.out.format("n=%d%n", n);
     System.out.format("n=%08d%n", n);
     System.out.format("n=%+8d%n", n);
     System.out.format("n=%,8d%n", n);
     System.out.format("n=%+,8d%n%n", n);
     System.out.println("Display of pi  using different format");
     System.out.format("pi=%f%n", pi);
     System.out.format("pi=%.4f%n", pi);
     System.out.format("pi=%10.3f%n", pi);
     System.out.format("pi=%-10.3f%n", pi);
     System.out.format(Locale.GERMAN,   "pi=%-10.4f%n%n", pi);
     }
}
```

Output:



10.3 CHARACTER CLASS AND ITS METHODS

In problem-solving, we often need to deal with variables (objects) of character and string type. In Java the following classes are proved to deal with characters and strings.

Character Class: Objects of this class hold a single character value only. Some methods are defined which can manipulate or check single-character data.

String Class: Objects of the String class are used to deal with the strings. The objects created using this class cannot be changed (immutable). This class proved many methods to perform different activities on string objects.

StrungBuilder Class: Objects of StringBuilder class are like String class objects, except that they can be modified. Internally, objects of StringBuilder class are treated like variable-length arrays of characters. At any point, you may change the length and content of the objects through the invocation of StringBuilder class methods.

StringBuffer Class: You can create string objects using this class, which is expected to be manipulated during the process.

Character Class

An object of Character class can contain only a single character value. While programming, if you need to use a single character value, most of the time you have to use the primitive char type. For example:

```
char ch = 'A';
// an array of chars
char[] charArray = { 'A', 'B', 'C', 'D', 'E' };
```

You sometimes need to use a Character object instead of a primitive char variable. For example, when you are required to pass a character value in a method while it changes the value passed as an argument or when you need to place a character value in a Java defined data structure, where the requirement is to work on objects. For example, Vector is one of the data structure that requires objects. You have already seen Java; we have *wrapper* class that "wraps" the char in a Character

object. As the Character class is immutable, i.e. once an object is created, it cannot be changed. You can create a Character object using the Character class constructor:

Character ch = new Character('a');

The ch is a character object created using Character class. Also, under some circumstances, the Java compiler creates a Character object for you. In Java when you pass a primitive char into a method as an argument which expects an object as an argument, then the Java compiler will automatically convert the char to a Character object for you. The following table 10.4 shows some of the commonly used Character class methods.

Table 10.4: Some Methods of Character Class

Method	Description
boolean isLetter(char ch) boolean isDigit(char ch)	This method tells whether the given char value is a letter or a digit.
boolean isWhitespace(char ch)	This method tells whether the specified char value is white space or not.
boolean isUpperCase(char ch) boolean isLowerCase(char ch)	This method tells whether the given char value is uppercase or lowercase.
char toUpperCase(char ch) char toLowerCase(char ch)	It returns the uppercase or lowercase form of the given char value.
toString(char ch)	It returns a String object representing the specified character value(i.e. a one-character string).
boolean compateTo(Character)	This method is used for comparing the values held by two character objects.
boolean equals(Character)	This method is used for comparing the values held by two character objects.
int compareTo (Character)	This method compare the value of the object on which this method is called with the value of the object passed as argument to the method.

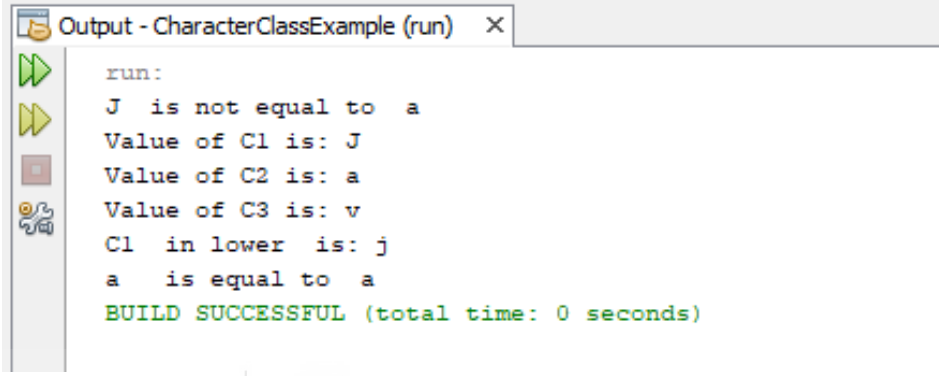
Now let us see an example in which Character objects are created and used. In the following program some character objects are created, some Character class methods are used and results are displayed.

Program

```
package characterclassexample;
public class CharacterClassExample
{
public static void main(String args[])
{
Character C1 = new Character('J');
Character C2 = new Character('a');
Character C3 = new Character('v');
Character C4 = new Character('a');
int diff;
diff = C1.compareTo(C2);
if (diff == 0)
System.out.println(C1+"   is equal to  "+ C2);
else
System.out.println(C1+"  is not equal to  "+ C2);
System.out.println("Value of C1 is: " + C1);
System.out.println("Value of C2 is: " + C2);
System.out.println("Value of C3 is: " + C3);
System.out.println("C1  in lower  is: " + Character.toLowerCase(C1));
```

```
diff = C2.compareTo(C4);
if (diff == 0)
System.out.println(C2+"   is equal to  "+C4);
else
System.out.println(C1+"  is not equal to  "+C2);
}
}
```

Output:



☞Check Your Progress-1

1) What is wrapper class? When is Number class used?

……

……

……

2) Write a program which displays the case (upper or lower) of a character object.

……

……

……

……

3) Explain the use of equals () method of Character class with the help of code statements.

……

……

……

……

10.4 STRING CLASS AND ITS METHODS

We use Strings widely in Java programming. Strings are a sequence of characters. In Java String, class is there to create and manipulate strings. In Java programming we may create objects of String class and use them. One important point to remember here is that the strings value of the String class objects will not change because String is immutable.

For example, if you create the following string object. The string contained by the object MyString given below cannot be changed.

String MyString = “This Java String cannot be changed!”.

Creating Objects of String Class

Following is the direct way to create a string:

String MyStringObj = "Welcome to Java String!";

Here " Welcome to Java String!" is a *string literal. It is a* series of characters in enclosed in double quotes (“ ”). Whenever Java compiler encounters a string literal in a program code it creates a String object with its value. In the above code, it is, Welcome to Java String!.

The String class has many constructors. Some of them are given below.

public String(): This constructor creates a String object which represents an empty character sequence.

public String(String value): This constructor is used for creating String objects that represent the same sequence of characters as passed in the argument.

public String(char value[]): This constructor you may use to create a new String object which is contained in the character array argument.

public String(byte bytes[], int offset, int length): This constructor is used to create a String object by converting the given sub-array of bytes using the platform’s default character encoding.

public String(byte bytes[]): This constructor is used to create a String object by converting the given array of bytes using the platform’s default character encoding.

As String class have many important methods for:

Examining individual characters of the strings,

Comparing strings,

Searching strings,

Extracting sub-strings, and

Creating a copy of a string

One important point to remember is that if you create a String object, and if it has the same value as another String object, in that case, Java will point both the object's references to the same memory location.

Creating Strings Objects

As you create any other object, you can create objects of String class using the new keyword and a constructor. The String class has many constructors which can be used to provide the initial value of the string using different sources, such as an array of characters:

```
char[] MyString = { 'B', 'o', 'o', 'k' '.' };
String Str1 = new String(MyString);
System.out.println(Str1);
```

The last line of above code snippet will display: Book.

Table 10.5 : Some Search Methods in the String Class

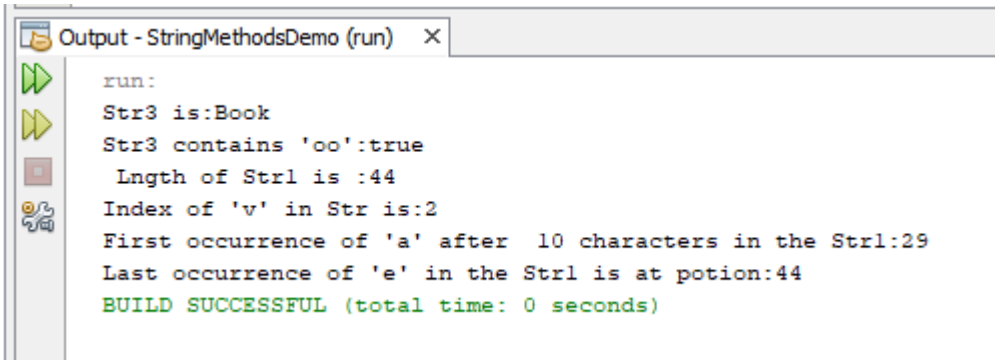
Method	Description
int indexOf(int ch) int lastIndexOf(int ch)	These methods returns the index of the first/last occurrence of the given character.
int indexOf(int ch, int fromIndex) int lastIndexOf(int ch, int fromIndex)	These methods returns the index of the first/last occurrence of the given character, searching forward (backward) from the specified index.
int indexOf(String str) int lastIndexOf(String str)	These methods returns the index of the first/ last occurrence of the given substring.
int indexOf(String str, int fromIndex) int lastIndexOf(String str, int fromIndex)	These methods return the index of the first/last occurrence of the specified substring, searching forward/backward from the specified index.
boolean contains(CharSequence s)	This method returns true if the string contains the specified character sequence.

The program given below demonstrates the use of some index methods.

Program

```
package stringmethodsdemo;
public class StringMethodsDemo
{
 public static void main(String[] args)
{
String  Str1 =  "Java is Object Oriented Programming Language";
char[]  Str2 =  {'B', 'o', 'o', 'k'};
String Str3 = new String(Str2);
System.out.println("Str3 is:"+Str3);
System.out.println("Str3 contains 'oo':"+Str3.contains("oo"));
System.out.println(" Lngth of Str1 is :"+ Str1.length());
System.out.println("Index of 'v' in Str is:"+Str1.indexOf('v'));
// Print first occurrence of character v in string Str1
System.out.println("First occurrence of 'a' after  10 characters in the Str1:"+
Str1.indexOf('a',9));
// Print first occurrence of character a in string Str1 after first 10 characters
System.out.println("Last occurrence of 'e' in the Str1 is at potion:"+
(Str1.lastIndexOf('e')+1));
// Print Last  occurrence of character e in string Str1
}
}
```

Output:



String class Method length ( )

This method is used to get the length of the string object. If you execute the following lines of code, you will find the length of Str1:

String Str1 = "My First String";
int len = Str1.length();

String Class Method charAt(int index)

This method return character at index i of the given String.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S	H	R	I	K	A	N	T		M	I	S	H	R	A							

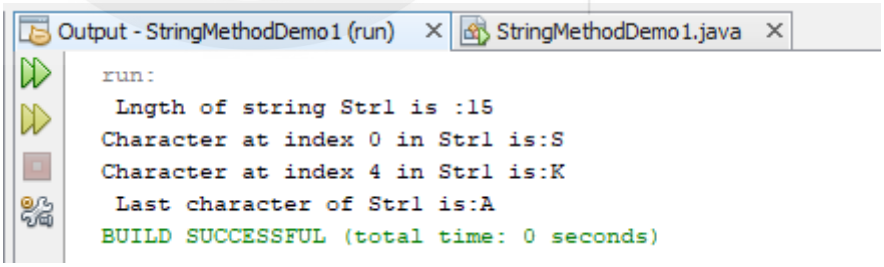
charAt (0): S charAt (9) : M
charAt (length ( ) –1): ?

Following program demonstrate use of lenth ( ) and charAt (int index) methods of String class.

Program

```
package stringmethoddemo1;
public class StringMethodDemo1
{
   public static void main(String[] args)
   {
   String  Str1 =  "SHRIKANT MISHRA";
   System.out.println(" Lngth of string Str1 is :"+ Str1.length());
    System.out.println("Character at index 0 in Str1 is:"+Str1.charAt(0));
   System.out.println("Character at index 4 in Str1 is:"+Str1.charAt(4));
   System.out.println(" Last character of Str1 is:"+Str1.charAt(Str1.length()-1));
   }
 }
```

Output:



Conversion Between Numbers and Strings

Converting Strings to Numbers

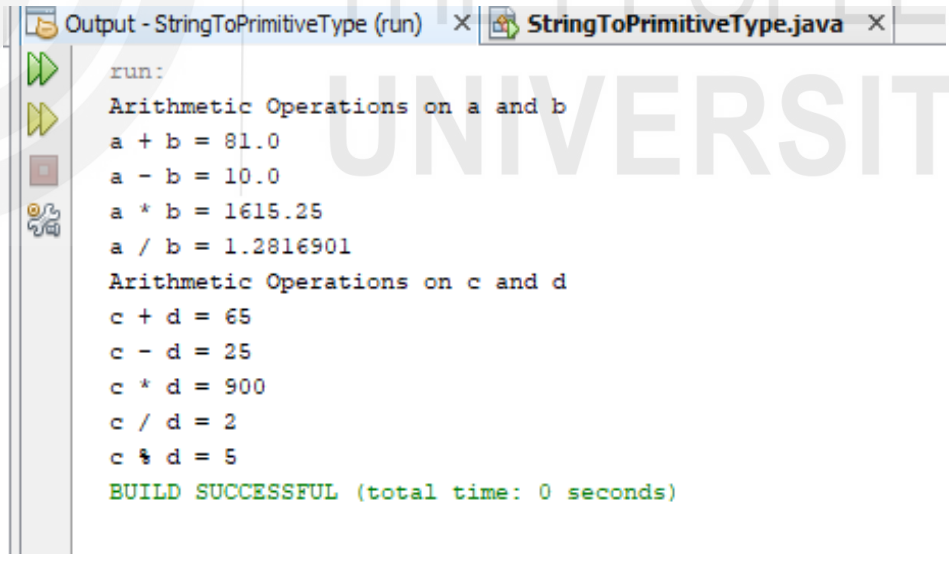
We need some time to convert numeric data in a string object to a numeric value. This conversion is done using some pre-defined methods valueOf. The Number subclasses that wrap primitive numeric types (Byte, Integer, Double, Float, Long, and Short) each subclass provide method *valueOf*( ), which converts a string to an object of that type. The following program converts the given strings to numbers and subsequently arithmetic operations are performed on the values.

Now let us see the example which uses the valueOf ( ) method to convert a string string to a primitive value.

Program:

```
package stringtoprimitivetype;
public class StringToPrimitiveType
{
 public static void main(String[] args)
 {
    String s1 = "45.5";
    String s2 = "35.5";
    String s3 = "45";
    String s4 = "20";
  // convert strings to float value
   float a = Float.valueOf(s1); //convert to float
   float b = Float.valueOf(s2); //convert to float
   //some arithmetic operations on a and b
   System.out.println("Arithmetic Operations on a and b");
   System.out.println("a + b = " + (a + b));
   System.out.println("a - b = " + (a - b));
   System.out.println("a * b = " + (a * b));
   System.out.println("a / b = " + (a / b));
    // convert strings to int value
   int c = Integer.valueOf(s3); // convert to integer
   int d = Integer.valueOf(s4); //convert to integer
   System.out.println("Arithmetic Operations on c and d");
   System.out.println("c + d = " + (c + d));
   System.out.println("c - d = " + (c - d));
   System.out.println("c * d = " + (c * d));
   System.out.println("c / d = " + (c / d));
   System.out.println("c % d = " + (c % d));
   }
 }
```

Output:



valueOf( ) Method for Converting Numbers to Strings:

As a programmer many times you need to convert a number to a string so that you may operate on the value in its string form. In the table 10.6 you can see some methods of String class which are used for converting Numbers to String.

Table 10.6: Methods for Converting Numbers to Strings

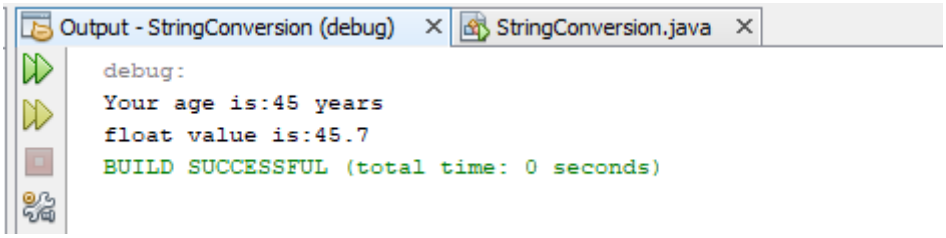
Method	Description
public static String valueOf(boolean b)	Return string representation of the boolean argument
public static String valueOf(char c)	Return string representation of the character argument
public static String valueOf(int i)	Return string representation of the integer argument
public static String valueOf(long l)	Return string representation of the long argument
public static String valueOf(float f)	Return string representation of the float argument
public static String valueOf(double d)	Return string representation of the double argument
public static String valueOf(char data[])	Return string representation of the character array argument
public static String valueOf(char data[],offset, int count)	Return string representation of a int specific subarray of the character array argument

In the folloing program, you can see that integer and float data are converted into a string type. Also, the operator '+' is used for concatenating two strings.

Program

```
package stringconversion;
public class StringConversion
{
public static void main(String[] args)
{
String s1 =  "Your age is:";
String s2 = new String();
String s3 = "float value is:";
int age = 45;
float f =45.7f;
s2 = String.valueOf(age);
s1 = s1+s2+ " years";
System.out.println(s1);
s2 = String.valueOf(f);
System.out.println(s3+s2);
}
}
```

Output:



A *palindrome* is a word or sentence that is symmetric in nature, and it is spelt the same from both forward and backward, ignoring case and punctuation. Now let us see a Java program that checks whether a given String is a *palindrome* or not.

Program

```
package palindromecheck;
public class PalindromeCheck
{
 public static void main(String[] args)
 {
       String MyPal = "MADAM";
       int len = MyPal.length();
       char[] tempChArray = new char[len];
       char[] ChArray = new char[len];
       System.out.println("Original string is:"+ MyPal);
       // in this loop we  put original string in an  array of chars
       for (int i = 0; i < len; i++)
       {
         tempChArray[i] =  MyPal.charAt(i);
       }
       // loop to  reverse array of chars
       for (int j = 0; j < len; j++)
       {
         ChArray[j] =   tempChArray[len - 1 - j];
       }
        String reversePal =   new String(ChArray);
        System.out.println("Reverse of String is:"+ reversePal);
     if (MyPal.toLowerCase().equals(reversePal.toLowerCase()))
     {
     System.out.println(MyPal + " is a Palindrome String.");
     }
    else
    {
     System.out.println(MyPal + " is not a Palindrome String.");
    }
 }
}
```

Output:

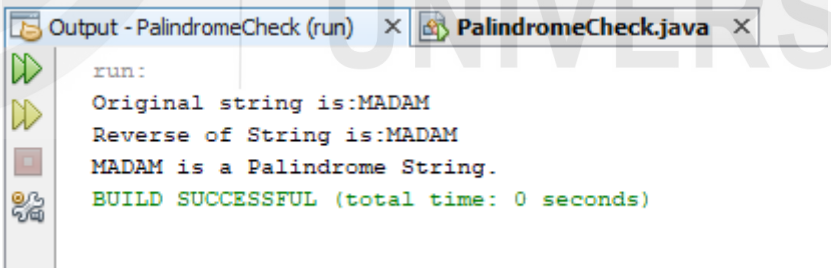


Table 10.7: String Class Comparisons Methods

Method	Description
public boolean equals(Object str)	Return true when strings contain the same characters in the same order.
public boolean equalsIgnoreCase(String aString)	Return true if both the strings, -contain the same characters in the same order, and ignore case in comparison.
public int compareTo(String aString)	Compares to aString returns <0 if a String< sourceString 0 if both are equal and return >0 if aString>sourceString ; this method is case sensitive and used for knowing whether two strings are identical or not.

public boolean startsWith (String prefix, int toffset)	Return true if prefix occurs at index toffset.
public boolean regionMatches(int toffset, String other, int offset, int len)	Return true if both the string have exactly the same symbols in the given region.
public boolean startsWith(String prefix)	Return true if string starts with prefix.
public boolean endsWith(String suffix)	Return true if string ends with suffix.

Table 10.8: Methods for Modifying Strings

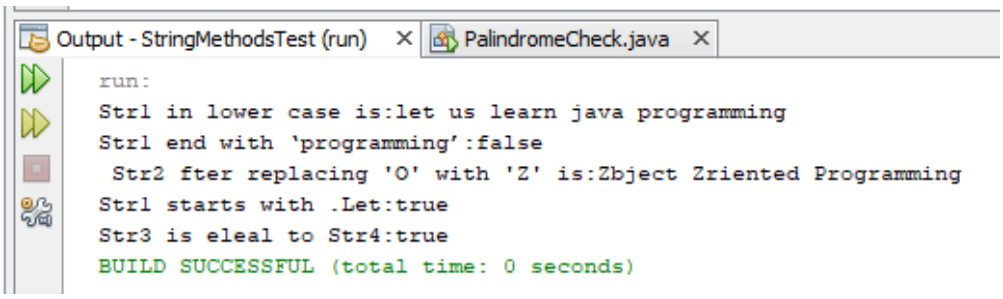
Method	Description
public String replace(char oldChar, char newChar)	Return a new string with all oldChar replaced by newChar
public String toLowerCase()	Return a new string with all characters in lowercase
public String toUpperCase()	Return a new string with all characters in uppercase.
public String trim()	Return a new string with whitespace deleted from front and back of the string.

The following program shows how to use the String class methods for comparisons and modifying strings.

Program

```
package stringmethodstest;
public class StringMethodsTest
{
 public static void main(String[] args)
 {
String Str1 = "Let us learn java programming";
String Str2 = "Object Oriented Programming";
String Str3 = "JAVA";
String Str4 = "java";
String Str5;
char a ='O';
char b= 'Z';
System.out.println("Str1 in lower case is:"+Str1.toLowerCase());
System.out.println("Str1 end with
'programming':"+Str1.endsWith("Programming"));
Str5= Str2.replace(a,b);
System.out.println(" Str2 fter replacing 'O' with 'Z' is:"+Str5);
System.out.println("Str1 starts with .Let:"+Str1.startsWith("Let"));
System.out.println("Str3 is eleal to Str4:"+Str3.equalsIgnoreCase(Str4));
}
 }
```

Output:



Some More String Class Methods

String class proves some methods for getting substring, comparison, and string modifying methods. These methods are useful in handling string objects, such as finding a character's location in a string, getting substrings, concatenation of strings, comparing strings, and matching string reasons, etc.

Table 10.9: Substring Mettthods

Methods	Description
public byte[] getBytes()	Return byte array of string characters
public String substring(int beginIndex)	Return substring from index beginIndex to the end of string
public String substring (int beginIndex, int endIndex)	Return substring from index beginIndex to the endIndIndex of string
public String concat(String str)	Return a concatenated string formed by the append/joining of two strings.
public char[] toCharArray()	Return character array of string

Explanation of String Class concat Method

This method of the String class is used for concatenating two strings. Syntax of using this method is:

string1.*concat*(string2);

The above code will return a new string which is the combination of string1 with string2 added to it at the end.

You can also use the **concat() method** with string literals, as in:
"This is ".concat(" Java Programming Course");

Also, strings are commonly concatenated with the '+' operator. You have widely used '+' operator in println statements. For example, the following code is an example of string concatenation:
String string1 = "This is my String";
System.out.println("string1 is:" + string1);

The above code will print: string1 is: This is My String.

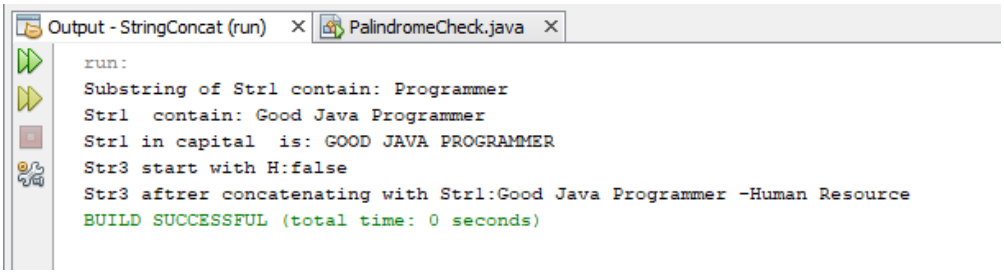
The following program demonstrates the use of substring methods and string concatenation method, and substring method.

Program:

```
package stringconcat;
public class StringConcat
{
public static void main(String[] args)
{
String Str1 = "Good Java Programmer ";
String Str2 ;
String  Str3 = "-Human Resource";
int index = Str1.indexOf('P');
Str2 =Str1.substring(index);
System.out.println("Substring of Str1 contain: " + Str2);
System.out.println("Str1  contain: " + Str1);
System.out.println("Str1 in capital  is: " + Str1.toUpperCase());
```

```
System.out.println("Str3 start with H:"+ Str3.startsWith("H"));
Str2 = Str1.concat(Str3);
System.out.println("Str3 aftrer concatenating with Str1:"+Str2);
}
}
```

Output:



10.5 STRINGBUILDER CLASS AND ITS METHODS

StringBuilder is another class in Java which is used for creating string objects. StringBuilder class objects are like String class objects, except that they can be modified. StringBulder objects are internally treated like variable-length arrays in Java. You can change the length and content of the sequence of StringBuilder objects at any point in time using StringBuilder class methods. The String Builder class is faster, as it performs no synchronization and it should generally be used in preference to String class. Concatenation of StringBuilder objects is more efficient compared to String objects. StringBuilder class supports all of the same operations as of the String class.

StringBuilder Class Constructors

StringBuilder(): This constructor is used to create an empty string builder with a capacity of 16.

StringBuilder(CharSequence cs): This constructor is used to create a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements which are trailing the CharSequence.The **CharSequence** is a readable sequence of char values.

StringBuilder(int initCapacity): This constructor is used to create an empty string builder with the specified initial capacity.

StringBuilder(String s): This constructor is used to create a string builder, which is initialized by the specified string, plus an extra 16 empty elements which are trailing in the string.

Some methods of the StringBuilder class are

length and capacity Methods of StringBuilder

The StringBuilder class have a length() method which returns the length of the string builder object. Also, every string builder has a *capacity*. The capacity is the number of character spaces that have been allocated to an string builder object and it is returned by the capacity() method. Capacity of string builder is always greater than or equal to the length (usually greater than) of object. The capacity of string builder

object automatically expand when there is a need to accommodate additions to the string builder object.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	S	T	R	I	N	G										

length() = 6 capacity( ) =16

Methods of StringBulder class

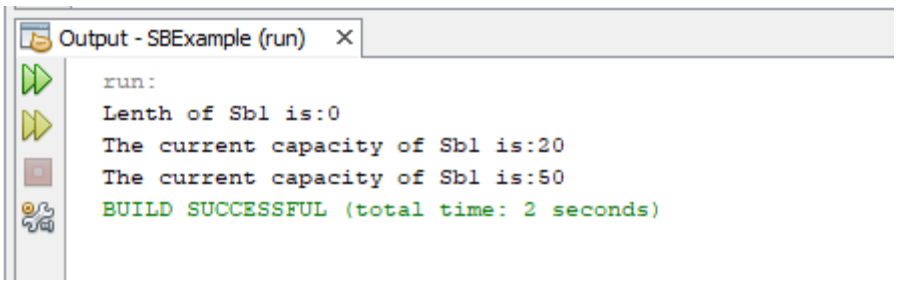
void setLength(int newLength): This method is used for setting the length of the character sequence. If newLength is less than length() of string builder object then the last characters which are remaining after newLenght coverage in the character sequence are truncated. If newLength is greater than length(), then null characters are appended at the end of the character sequence.

void ensureCapacity(int minCapacity): This method ensures that the capacity of the object is at least equal to the capacity passed as argument i.e. minCapacity.

Program

```
package sbexample;
public class SBExample
{
    public static void main(String[] args)
    {
      StringBuilder Sb1 = new StringBuilder(20);
      int len, cap ;
      len = Sb1.length();
      System.out.println("Lenth of Sb1 is:"+len);
      cap = Sb1.capacity();
      System.out.println("The current capacity of Sb1 is:"+ cap);
      Sb1.ensureCapacity(50);
      cap = Sb1.capacity();
      System.out.println("The current capacity of Sb1 is:"+ cap);
    }
}
```

Output:



StringBuilder Operations

Some operations on a StringBuilder that are not available in String are the append() and insert() methods. These methods are overloaded so that they can accept data of any type. Each method first converts the argument passed to it to a string, subsequently they append or insert the characters of that string to the character sequence in the string builder.

When we use the append method, it always adds the characters at the end of the existing character sequence. If we use the insert method, we can add the characters at a specified point in the object.

Table 10.10 : Somes More StringBuilder Methods

Method	**Description**
StringBuilder append(boolean b) StringBuilder append(char c) StringBuilder append(char[] str) StringBuilder append(char[] str, int offset, int len) StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s)	These methods are used to append the argument to the string builder object on which it is called. The argument value passed is converted to a string before the append operation takes place.
StringBuilder delete(int start, int end)	This method deletes the subsequence from the start point to end-1 (inclusive) in the StringBuilder's object.
StringBuilder deleteCharAt(int index)	This method deletes the character located at index.
StringBuilder insert(int offset, boolean b) StringBuilder insert(int offset, char c) StringBuilder insert(int offset, char[] str) StringBuilder insert(int index, char[] str, int offset, int len) StringBuilder insert(int offset, double d) StringBuilder insert(int offset, float f) StringBuilder insert(int offset, int i) StringBuilder insert(int offset, long lng) StringBuilder insert(int offset, Object obj) StringBuilder insert(int offset, String s)	These methods are used for Inserting the second argument into the string builder. In the methods, the first integer argument indicates the index before which the data passed as the second argument is to be inserted. Remember that the data passed as a second argument is converted to a string before the insert operation takes place.
StringBuilder replace(int start, int end, String s) void setCharAt(int index, char c)	This method replaces the specified character(s) in this string builder.
StringBuilder reverse()	This method is used to reverse the sequence of characters in this string builder object.

When we use append(), insert(), or setLength() methods, they increase the length of the character sequence in the object of the string builder. This results to increase in length () , and it would be greater than the current capacity of the string builder object. Therefore when such an operation is processed, the capacity is automatically increased.

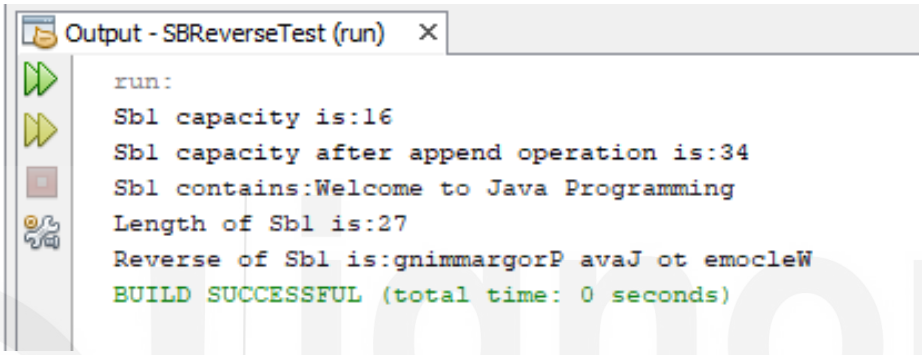
The program given below demonstrates the use of the append and reverse methods of StringBuilder.

Program:

```
package sbreversetest;
public class SBReverseTest
{
 public static void main(String[] args)
```

```
{
  // creates empty builder, capacity 16
 StringBuilder Sb1 = new StringBuilder();
 System.out.println("Sb1 capacity is:"+Sb1.capacity());
 // adds … character string at beginning
 Sb1.append("Welcome to Java Programming");
 System.out.println("Sb1 capacity after append operation is:"+Sb1.capacity());
 System.out.println("Sb1 contains:"+Sb1);
 System.out.println("Length of Sb1 is:"+Sb1.length());
 System.out.println("Reverse of Sb1 is:"+Sb1.reverse());
 }
}
```

Output:



Also, you can use any String class method on a StringBuilder object. To use this facility, first, you have to convert the StringBuilder object to a String object with the toString() method of the StringBuilder class. After that, using object using the StringBuilder(String str) constructor, you can convert the String object back into a StringBuilder.

Now, the following program that we have implemented in section "Strings". This program is about reversing a palindrome. Here you can see that the program written here is more efficient when we have used a StringBuilder instead of a String.

Program using String class:

```
package palindromecheck;
public class PalindromeCheck
{
 public static void main(String[] args)
 {
       String MyPal = "MADAM";
       int len = MyPal.length();
       char[] tempChArray = new char[len];
       char[] ChArray = new char[len];
       System.out.println("Original string is:"+ MyPal);
       // in this loop we  put original string in an  array of chars
       for (int i = 0; i < len; i++)
       {
         tempChArray[i] =  MyPal.charAt(i);
       }
       // loop to  reverse array of chars
       for (int j = 0; j < len; j++)
       {
         ChArray[j] =   tempChArray[len - 1 - j];
```

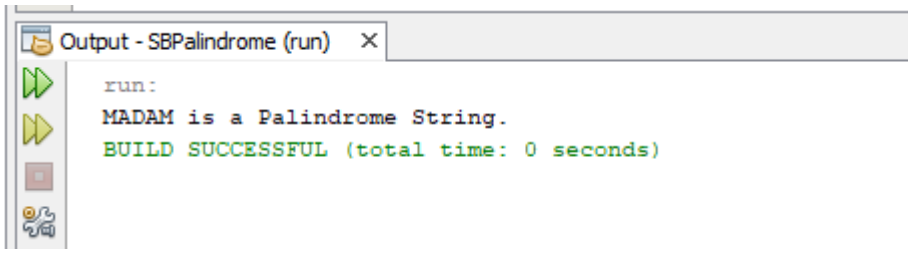
```
        }
         String reversePal =   new String(ChArray);
         System.out.println("Reverse of String is:"+ reversePal);
      if (MyPal.toLowerCase().equals(reversePal.toLowerCase()))
      {
      System.out.println(MyPal + " is a Palindrome String.");
      }
     else
     {
      System.out.println(MyPal + " is not a Palindrome String.");
     }
 }
}
```

The program of palindrome checking is written using StringBuilder class below. You can see that the below-written program is more simple and small compared to the above program. It is possible because, in a string builder, there is a method named reverse() , which is used to make the code simpler and easier to read:

Program:

```
package sbpalindrome;
public class SBPalindrome
{
public static void main(String[] args)
{
        String MyPal = "MADAM";
        StringBuilder Sb = new StringBuilder(MyPal);
        StringBuilder S3;
       // reverse Sb
       S3= Sb.reverse();
       if (Sb.equals(S3))
       {
       System.out.println(MyPal + " is a Palindrome String.");
       }
       else
       {
       System.out.println(MyPal + " is not a Palindrome String.");
       }
   }
}
```

Output:



☞Check Your Progress-2

1) Write a java program which displays digits after dot(.) of a double data type variable.

……

……

2) What will be the output of the following program?

```
package strtest;
public class StrTest
{
public static void main(String[] args)
{
   String Str = new String("Sring Handling in Java");
   System.out.println("Length of Str is :"+Str.length());
   int i = Str.lastIndexOf('a');
   int j = Str.indexOf('a');
   System.out.println("Difference between first and last occurence of 'a' in
    MyStr is :"+ (i-j));
 }
}
```

……

……

……

3) Write a java program using StringBuilder which append Str2 to Str1 where Str1 is " Java is Simple" and Str2 is "Programming Language". Also reverse the resultant string after apped operation.

……

……

……

10.6 STRINGBUFFER CLASS AND ITS METHODS

The String class objects are of fixed length and cannot be modified. If there is a need of strings to be modified, then you have another option of using StringBuffer class object. Now StringBuffer class is almost obsolete. Almost all the features provided in StringBuffer class are supported by StringBuilder class which you have learned in the previous section. Now let us have a look at StringBuffer class in detail. String buffers class is generally used for constructing string objects dynamically when you need a string which may change. The StringBuffer class and the StringBuilder class are the same, except that StringBuffer class is thread-safe because its methods are synchronized, which means StringBuffer objects are safe for use by multiple threads. Major operations on a StringBuffer are the append and insert methods. These methods are overloaded so that they can accept data of any type.

StringBuffer Class Constructors

StringBuffer(): Used to define StringBuffer object with no characters in it, and it has the initial capacity of 16 characters.

StringBuffer (int size): It is used to construct StringBuffer object with no characters in it and with an initial capacity specified by the size argument.

StringBuffer (String str): It is used to construct StringBuffer object with the given sequence of characters as the string argument.

Some Methods of String Buffer Class

SrtingBuffer provides the functionality of String class methods plus some more methods which are used :

- to modify strings by inserting substrings,
- for deleting contents of string,
- to append string, and
- for altering size of string dynamically.

public int length(): This method returns the length of the string buffer object.

public int capacity(): This method returns the total allocated capacity to the string buffer.

Every string buffer object has a capacity. Until the length of the character sequence contained in the object is less than this capacity,there is no need to allocate a new internal buffer array for the object. If the internal buffer overflows(increases more than capacity), it is automatically made larger.

public void ensureCapacity (int MinCapacity): This method ensures the capacity of the buffer at least equal to the specified MinCapacity.

When the current capacity of the string buffer object (on which **ensureCapacity** method is called) is less than the argument value, then a new internal buffer is allocated which has greater capacity . The new capacity of the object is the larger than:

i). The MinCapacity argument
ii). Twice the old capacity, plus 2 .
iii) If you pass a non-positive value as the MinCapacity then this method takes no action.

public void **setLength**(int newLength) : This method is used for setting the length of the string buffer object. If the new length is less than the current length of the object then, the string content is truncated so that it only contains exactly the same number of characters as given by the newLength argument. In case the newLength argument is greater than or equal to the current length of the object, there is is no truncation of characters; rather object is appended with sufficient null characters so that its length increases to the newLength argument.

public void **setCharAt**(int index, char ch): This method is used to place given character 'ch' at the position given by index in the object. This operation result into altering the character sequence because of addition of new character in object. After this operation the existing characters at index and characters after that in the object are shifted right by one position.

public StringBuffer **append**(String newStr) This method appends the newStr to the StringBuffer object. When you use this operation it result into increasing the length of the object.

public StringBuffer **append**(int i): This method can be used to append the string representation of the int argument to the string buffer. Internally the argument passed is converted to a string then appended to the string buffer.

public StringBuffer **insert** (int offset,char[] str): Using this method you can insert the string representation of the character array argument into the string buffer object. The position of insertion is the value indicated by offset. Using this operation

increases the length of the StringBuffer object by the length of the argument passed to append.

public StringBuffer **insert** (int offset, String str): Using this method you can insert a given string in the string buffer object at position indicated by offset. It original contents of the object move up its characters after the offset position.

public StringBuffer **reverse**(): This method reverses the string contained in the StringBuffer object.

public StringBuffer **deleteCharAt**(int index) : This method removes the character located at the index position in the string buffer object.

public StringBuffer **replace** (int start, int end, String Str): This method is used to replace the characters of the string buffer object from start index point to the end index point by the argument Str.

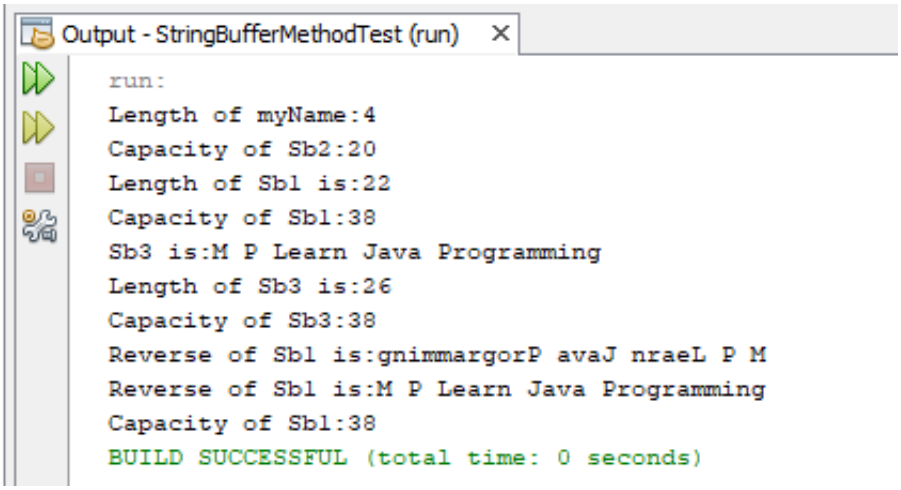
public StringBuffer **delete** (int start, int end): This method deletes characters (total characters delete: end–start +1) from location start and up to including location end.

Following program use of some of the methods like capacity, append,replace,delete, and reverse etc. of StringBuffer class. In the program given below capacity of the StringBuffer object SB1 never changes, while the length of the the Sb1 increased after inserting elements in it.

Program:

```
package stringbuffermethodtest;

public class StringBufferMethodTest
   {
      public static void main ( String args[])
      {
      StringBuffer Sb1 = new StringBuffer("Learn Java Programming");
      String myName = new String("M P ");
      StringBuffer Sb2 = new StringBuffer(myName);
      StringBuffer Sb3;
      System.out.println("Length of myName:"+Sb2.length());
      System.out.println("Capacity of Sb2:"+Sb2.capacity());
      System.out.println("Length of Sb1 is:"+Sb1.length());
      System.out.println("Capacity of Sb1:"+Sb1.capacity());
      Sb3 = Sb1.insert(0,myName);
      System.out.println("Sb3 is:"+Sb3);
      System.out.println("Length of Sb3 is:"+Sb3.length());
      System.out.println("Capacity of Sb3:"+Sb3.capacity());
      System.out.println("Reverse of Sb1 is:"+Sb1.reverse());
      //reverse back to original string
      System.out.println("Reverse of Sb1 is:"+Sb1.reverse());
      System.out.println("Capacity of Sb1:"+Sb1.capacity());
      }
   }
```

Output:

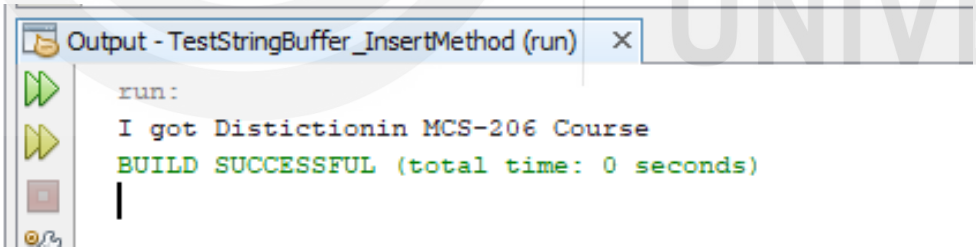


Now let us see one more example program in which you can insert data into the middle of a StringBuffer object using insert methods.

Program:

```
package teststringbuffer_insertmethod;
public class TestStringBuffer_InsertMethod
{
public static void main ( String args[])
{
StringBuffer MyBuffer = new StringBuffer("I got in MCS-206 Course");
MyBuffer.insert(6,"Distiction");
System.out.println(MyBuffer.toString());
}
}
```

Output:



10.7 AUTOBOXING AND UNBOXING

Autoboxing: Automatic conversion made by the Java compiler between the primitive types and their corresponding object wrapper classes is called *Autoboxing*. For example, it converts an int primitive type to an Integer object ,or a double primitive type to a Double, and so on. When this conversion is done in the other way by the Java compiler, it is known as *unboxing.*

Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 20; i ++)
   li.add(i);
```

In the above code, the int values of i(primitive type value) are added in the List object li, rather than Integer objects (as required by List), the code compiles. As li is a List of Integer objects, not a List of int values, still the Java compiler does not issue a compile-time error because it creates an Integer object from the primitive int value i and adds the object to li. Actually, the compiler converts the above code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 20; i++)
   li.add(Integer.valueOf(i));
```

Therefore here, when a primitive value (an int, for example) is converted into an object of the corresponding wrapper class (Integer), it is called autoboxing.

The process of autoboxing is applied by the Java compiler when a primitive value is:

- Passed in a method as a parameter and that parameter expects an object of the corresponding wrapper class.
- Need to be assigned to a variable of the corresponding wrapper class.

Unboxing

In Java unboxing is a process through which an object of a wrapper type, let us say Integer, is converted to its corresponding primitive type (int) value. The unboxing is applied by the Java compiler when an object of a wrapper class:

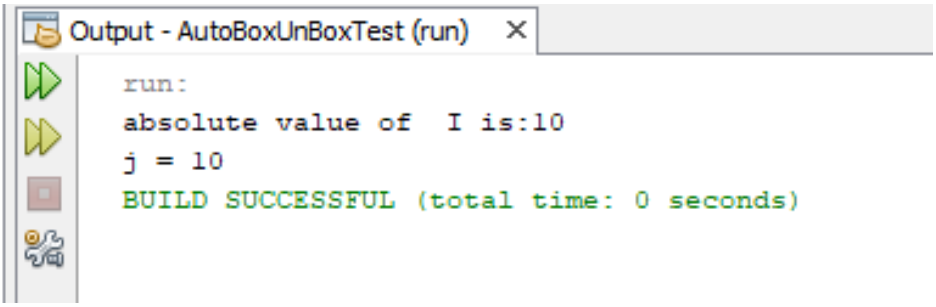
- Is being passed as an argument to a method where a value of the corresponding primitive type is expected.
- Has been assigned to a variable of the corresponding primitive type.

Following is a Java program to demonstrate how the unboxing works:

Program:

```
package autoboxunboxtest;
import java.util.ArrayList;
import java.util.List;
public class AutoBoxUnBoxTest
{
      public static void main(String[] args)
      {
       Integer I = new Integer(10);
      // Unboxing through method invocation
      int iVal = I.intValue();
      System.out.println("absolute value of  I is:" + iVal);
      List<Integer> li = new ArrayList<>();
      // Autoboxing in add( ) method
      li.add(iVal);
      // Unboxing through assignment
      int j=li.get(0);
      System.out.println("j = " + j);
   }
}
```

Output:



As a Java programmer, you should avoid using autoboxing and unboxing for scientific computing, or in other applications which are having performance-sensitive numerical code. Both autoboxing and unboxing help Java developers in writing cleaner code. Also, it makes code easier to read as an Integer is not a substitute for a primitive int value. In fact when autoboxing and unboxing processes are performed, they only blur the difference between primitive types and reference types, but they are unable to totally eliminate it.

The table given below shows the primitive types and their corresponding wrapper classes. These are used by the Java compiler for autoboxing and unboxing:

Table 10.11 : Primitive Type and Corresponding Wrapper Classes

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

☞Check Your Progress-3

1) What is the initial capacity of the following StringBuilder object Sb1? Also, find substring(6, 12) and charAt(9) in StringBbuilder Sb1 given below.

StringBuilder Sb1 = new StringBuilder("Now StringBuilder is used in place of StringBuffer");

……

……

……

……

2) When should a StringBuffer object be preferred over a String object?

……

……

……

……

3) Write a Java program which displays the length and initials of the given name (string) of a person using the StringBuffer class.

……

……

……

……

4) What will be the output of the following program?

Program:

```
package arraylistsum;
import java.util.ArrayList;
public class ArrayListSum
{
 public static void main(String[] args)
 {
     ArrayList<Integer> list = new ArrayList<>();
     //autoboxing
     list.add(1);
     list.add(2);
     list.add(3);
     list.add(4);
     System.out.println("ArrayList: " + list);
     // unboxing
     int sum =0;
     for(Integer i:list)
     sum =sum+i;
     System.out.println("The sum of list elements is:"+sum);
  }
}
```

……

……

……

……

10.8 SUMMARY

Dealing with characters and strings are among the routine job in programming. Java provides Character, String, StringBuilder and StringBuffer classes for handling characters and strings. Also, Java provides *wrapper* classes for each primitive data type. These classes are used to create object of that data type. Each primitive type in Java has a corresponding wrapper class. Java String class is used for creating the string objects which cannot be changed. This unit explained about wrapper clasess, Number class and its conversion methods. Character class provides various methods like compareTo, equals, isUpperCase. This unit explains about use of Character class. Also, use of prinf and format method are explained in this unit. Subsequetly String class and some of its methods are explained. In general string classes is not used for dynamic Strings, for that purpose StringBuilder class and StringBuffer class and their methods are explained. Finally concept of autoboxing and unboxing are explaind.

10.9 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

☞Check Your Progress-1

1) For all the primitives, there are classes that "wrap" the primitive in an object. The Number Class (an abstract class) and its subclasses are wrapper classes, and each wrapper class has **Object** as a superclass. Byte, Short, Integer, Long, Float and Double have **Number** as their direct superclass. For more details, refer to section 10.2 of this unit.

   The following are three reasons to use a Number object rather than a primitive:

   i) When a method expects an object as an argument.

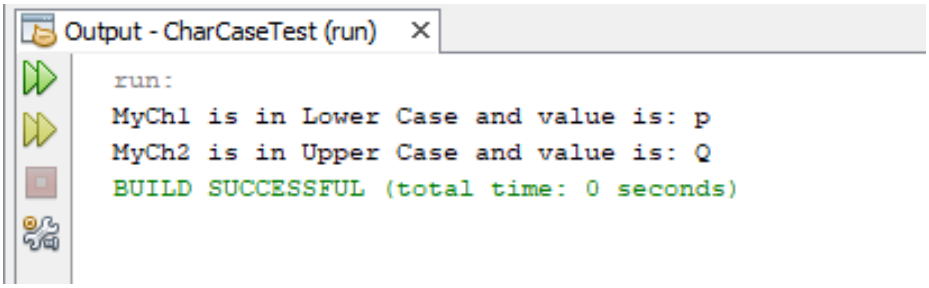
   ii) Using constants defined by the class, such as MIN_VALUE, which lowers the bounds of the data type and MAX_VALUE, which provide the upper bounds of the data type.

   iii) When there is a need to use class methods for converting values to primitive types and vice-versa. For example, when you need to convert to and from strings or to convert between number systems.

2) //Program to test whether the content of a Character object is Upper or Lower

```
package charcasetest;
public class CharCaseTest
 {
        public static void main(String args[])
        {
        Character MyCh1 = new Character('p');
        Character MyCh2 = new Character('Q');
  //Test for MyCh1
        if (MyCh1.isUpperCase(MyCh1.charValue()))
      System.out.println("MyCh1 is in Upper Case and value is: "+MyCh1);
        else
      System.out.println("MyCh1 is in Lower Case and value is: "+MyCh1);
  // Test for MyCh2
        if (MyCh2.isUpperCase(MyCh2.charValue()))
      System.out.println("MyCh2 is in Upper Case and value is: "+MyCh2);
        else
      System.out.println("MyCh2 is in Lower Case and value is: "+MyCh2);
    }
  }
```

Output:



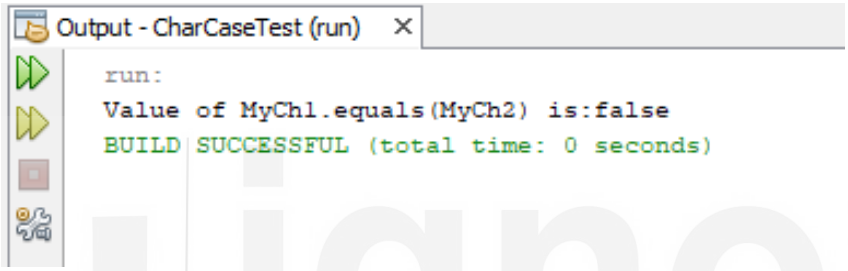
3) This instance method is used to compare the value held by the current object with the value held by another object. For example, let MyCh1 and MyCh2 be two objects of Character class then

MyCh1.equals(MyCh2);

the method returns true if the values held by both objects are equal; otherwise false.

```
package charcasetest;
public class CharCaseTest
 {
      public static void main(String args[])
      {
              Character MyCh1 = new Character('p');
              Character MyCh2 = new Character('Q');
            System.out.println("Value of MyCh1.equals(MyCh2)
is:"+MyCh1.equals(MyCh2));
      }
  }
```

Output:



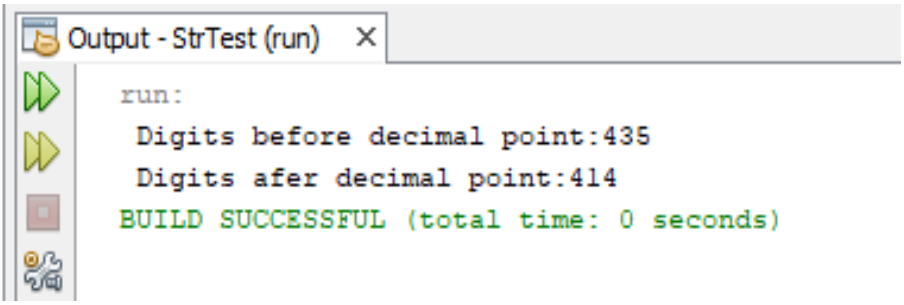
☞Check Your Progress-2

1)

Program

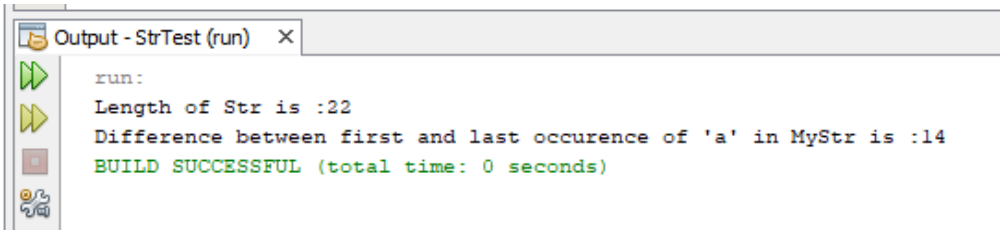
```
package strtest;
public class StrTest
{
public static void main(String[] args)
{
   double d = 435.414;
   String Str = Double.toString(d);
   int dot = Str.indexOf('.');
   int len = Str.length();
   System.out.println(" Digits before decimal point:"+Str.substring(0,dot));
   System.out.println(" Digits afer decimal point:"+Str.substring(dot+1, len));
  }
 }
```

Output:



2)

Output:

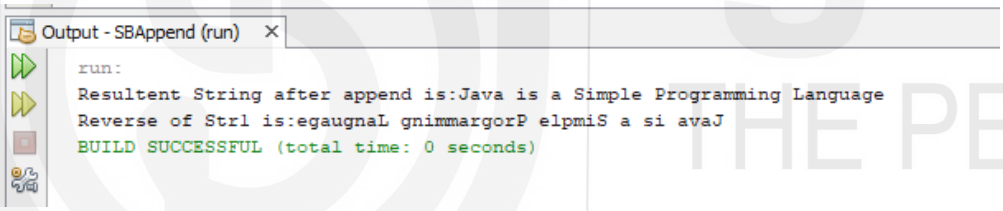


3)

Program:

```
package sbappend;
public class SBAppend
{
 public static void main(String[] args)
 {
   StringBuilder Str1 = new StringBuilder("Java is a Simple ");
   StringBuilder Str2 = new StringBuilder("Programming Language");
   Str1.append(Str2);
   System.out.println("Resultent String after append is:"+Str1);
   System.out.println("Reverse of Str1 is:"+Str1.reverse());
 }
}
```

Output:



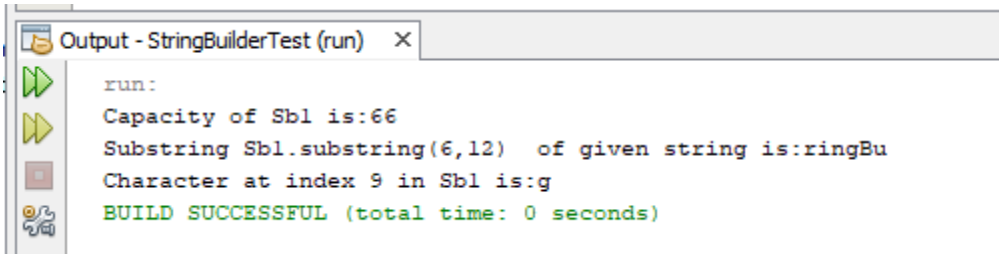
☞Check Your Progress-3

1)

Program:

```
package stringbuildertest;
public class StringBuilderTest
{
   public static void main(String[] args)
   {
    StringBuilder Sb1 = new StringBuilder("Now StringBuilder is used in place
of StringBuffer");
    String Sb2;
    System.out.println("Capacity of Sb1 is:"+ Sb1.capacity( ));
    Sb2 = Sb1.substring(6, 12);
    System.out.println("Substring Sb1.substring(6,12)  of given string is:"+Sb2);
    System.out.println("Character at index 9 in Sb1 is:"+ Sb1.charAt(9));

   }
}
```

Output:



2) The StringBuffer class object should be used when there is a need of modification (change may take place) in the content of the string. When we insert some content in an object of StringBuffer class , it increases the size of the object as per need. Therefore StringBuffer object should be given preference over String objects when you need:

   i. to insert a substring in a given string.

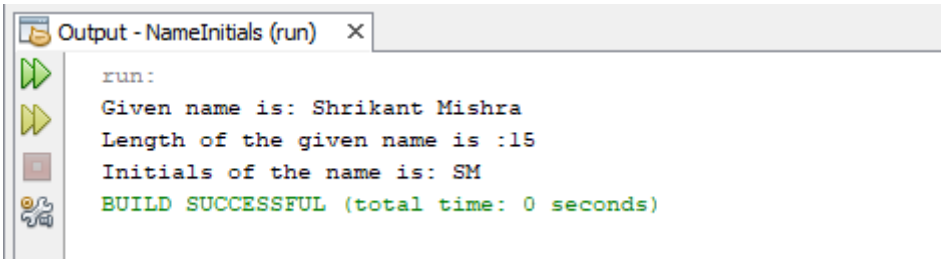
   ii. to reverse the content of a string object.

3)

Program:

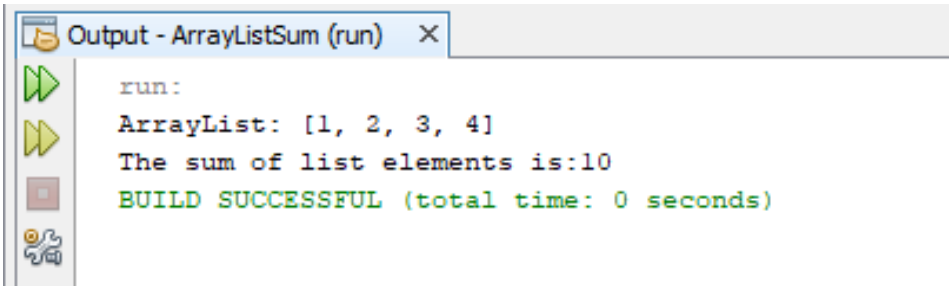
```
package nameinitials;
public class NameInitials
 {
   public static void main(String[] args)
   {
      String Str1 = "Shrikant Mishra";
        StringBuffer Str2 = new StringBuffer();
        // Find initial of the given name
        int len = Str1.length();
        for (int i = 0; i < len; i++)
         {
           if (Character.isUpperCase(Str1.charAt(i)))
           {
          Str2.append(Str1.charAt(i));
            }
       }
        System.out.println("Given name is: " + Str1);
        System.out.println("Length of the given name is :"+len);
        System.out.println("Initials of the name is: " + Str2);
    }
   }
```

Output:



4)

Output:



10.10 REFERENCES/FURTHER READINGS

- MCS-024 Course of IGNOU BCA(Revised) Programme, Block -3 Unit-3: Strings and Characters
- (https://egyankosh.ac.in/handle/123456789/11742)
- Herbert Schildt, “Java The Complete Reference”, McGraw-Hill,2017.
- https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html
- https://docs.oracle.com/javase/tutorial/java/data/characters.html
- https://docs.oracle.com/javase/tutorial/java/data/buffers.html
- https://docs.oracle.com/javase/tutorial/java/data/buffers.html
- https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html
- https://medium.com/@bpnorlander/java-understanding-primitive-types-and-wrapper-objects-a6798fb2afe9

UNIT 11 I/O STREAMS

Structure **Page No.**

11.0 Introduction 280
11.1 Objectives 281
11.2 Introduction to I/O Class and Interfaces 281
11.3 File Class and its Methods 283
11.4 The AutoCloseable, Closeable and Flushable Interfaces 287
11.5 I/O Exceptions 287
11.6 Introduction to Stream Classes 289
11.7 Byte Stream Classes 289
11.8 Character Stream Classes 296
11.9 Serialization 301
11.10 Summary 305
11.11 Solution /Answers to Your Progress 306
11.12 References /Further Reading 308

11.0 INTRODUCTION

Input refers to any piece of information required for completion of execution of a program(generally provided by users) while output refers to the information that the program provides to the user. For the development of an application, both input and output are essential. Computer programs can enhance their usefulness when they are able to interact with the rest of the world, which consists of users, machines, computers, etc., by some means. Interaction here refers to input, output, or I/O operations. The previous chapters mainly focused on only one type of interaction i.e. interaction with a user either through command line prompt or through a graphical user interface. But it is only one means of computer program and world interaction in which the user acts as both source and destination of the information. Similarly, one other type of input/output, i.e. TextIO, is used for reading and writing text from a file. In addition to TextIO, Java features a much powerful and flexible input/output framework enabling diverse ways of I/O interaction (socket, files, etc.). Java also features networked communication thereby increasing the program's usefulness.

Since in java, every I/O interaction is carried out via program interface. The following are the points that the java input/output interface must address.

1. Identification of end of the line in a file when a different convention is used by the different operating systems.
2. Handling different encoding systems used by the same operating system
3. Handling of file naming and path naming conventions as they may be different for a different system.

These all create a challenge for Java developers to counter these issues for Java users while performing input/output operations from an externalsource/destination. Java addresses these issues by integrating the java.io.* package enabling users to worry-free input/output operations.

In this unit, we elaborate on what are streams, different types of streams, how to read data from the input stream, how to send data over the output stream, etc. The

unit also discusses the concept of I/O classes and interface, File class, Byte stream, Character stream, and serialization in detail for proper understanding.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- explain a clear view of the Java input/output hierarchy,
- use I/O class and interface in your program,
- use File class for handling files and directories,
- use I/O class for reading and writing data,
- use different types of I/O stream,i.e. Byte stream and Character stream,
- handling of I/O Exception in the programs, and
- use the concept of Serialization.

11.2 INTRODUCTION TO I/O CLASS AND INTERFACES

In java to perform I/O operation, the concept of stream is used. Stream is a sequential flow of input or output data. Stream enables java programs to serially accept input from various data sources such as socket, file, buffer, etc. Stream also helps the programmer to create a file and send data over the internet. The Input stream is used to read data from the input source, while the Output stream helps to write data to a destination. Here source and destination can be anything that generates, consumes and stores the data such as program, disk, peripheral device, etc. The operation of I/O in java is shown in figure 11.1.

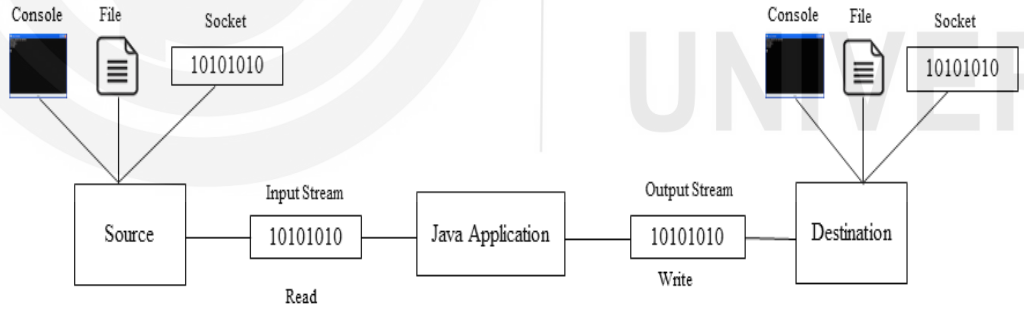


Figure 11.1: Java I/O

The section discusses different classes and interfaces defined in package java.io. The package mainly classifies the I/O class in 5 different class hierarchy sets: InputStream, OutputStream, Reader, Writer, and File. Input Stream and Output Stream are used for reading and writing byte data, while Reader and Writer classes are used for reading and writing the character data. Based on types of data (byte, file, character) and form of I/O device (file, array) further, these classes are classified. The hierarchy of the I/O classes has been illustrated in Figure 11.2.

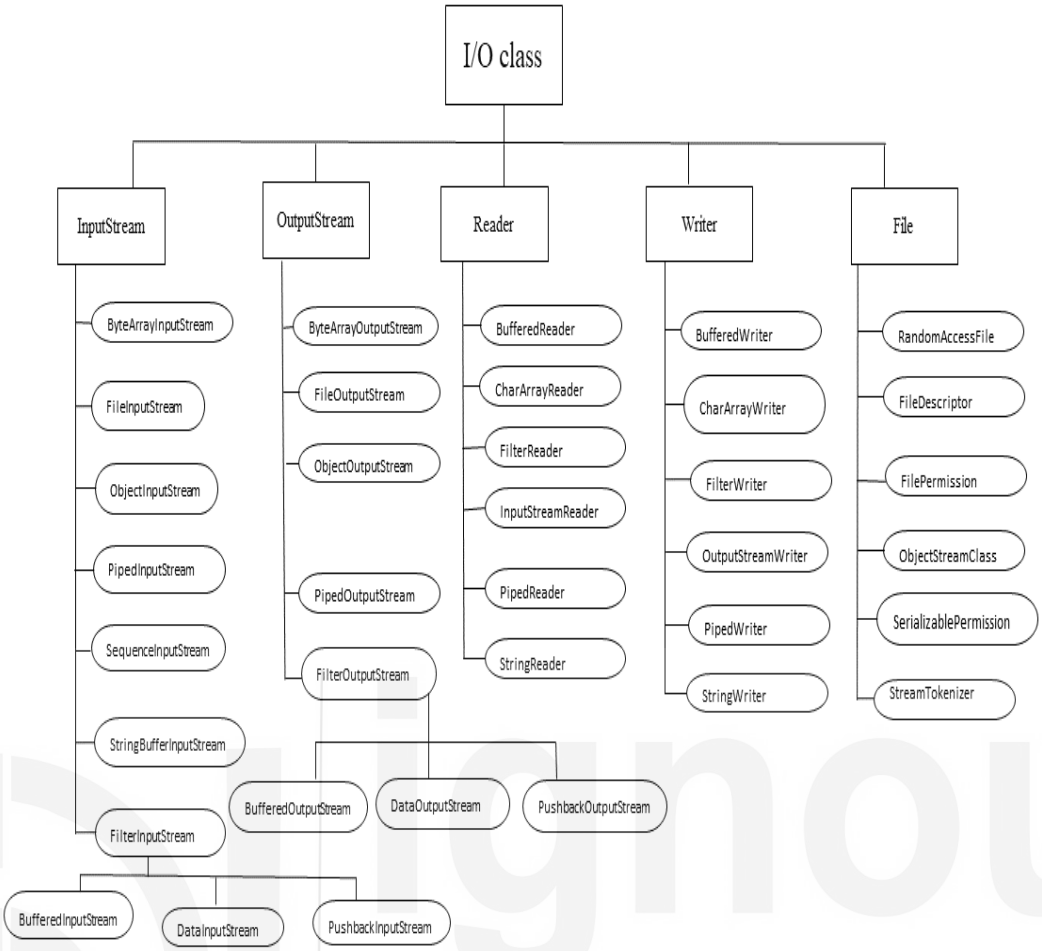


Figure 11. 2: I/O Class hierarchy

Each of these classes defined above serves only one objective but, when combined with other classes can serve complex task objectives. For example, a Buffered Reader allows buffering of input while a FileReader allows a method to connect file. When both are used together, the user can perform buffered reading from a connected file.

11.2.1 Java Interfaces

The Interface provides a way to perform input and output operations using data streams, file, and serialization. The list of interfaces provided by java.io is summarized in Table 11.1.

Table 11.1: List of Interfaces

Sr.No.	Interface & Description
1	**Closeable** Allows source and destination of data to be closed.
2	**DataInput** Allows reading of byte data from binary stream and converting these data to any of Java primitive types.
3	**DataOutput** Allows converting of Java primitive data to byte and write into binary stream.

4	**Externalizable** Enables instance of class object to save and restore its content.
5	**FileFilter** Provides filter for abstract pathnames.
6	**FilenameFilter** Provides filter for filenames.
7	**Flushable** Allows destination of data to flushed.
8	**ObjectInput** Extends DataInput interface for reading of objects and allow deserialization of objects
9	**ObjectInputValidation** Represents callback interface and enable validation of object within the graph.
10	**ObjectOutput** Extends DataOutput interface for writing of objects and allow serialization of objects
11	**ObjectStreamConstants** Enables constant to be written in Object Serialization stream.
12	**Serializable** Allows serialization of objects/class by implementing the java.io.Serializable interface.

11.3 FILE CLASS AND ITS METHOD

Most of the java.io package classes operate on stream concept, but the File class defined in java.io doesnot use stream concept. It directly interacts with the file and file system. The file class only describe the file properties but doesn't provide a view of how information related to file are stored or retrieved. For manipulating or obtaining information associated with a file, an object known as File object is created. The properties of file here represent permission, date and time of file creation and directory path of file. Even though there are some restrictions on using file (within applets) due to security issues, still file is the primary source of storing and sharing information of many programs. The directory in java is a collection of files, whose filenames (list of files) can be fetched using the list() method.

The following are how an object of the File class can be created using a different form of File constructor.

File $f1$ = new File("c:/JavaPrograms/File_Name.txt");

File $f2$ = new File("c:/JavaPrograms", "File_Name.txt");

File $f3$ = new File("JavaPrograms", "File_Name.txt");

In the first form, $f1$ file object is created by File constructor accepting only one string parameter. The string parameter consists of the path of the file and the file name. For $f1$ filepath is c:/JavaPrograms while the name of the file is File_Name.txt. Similarly, $f2$ is created using the File constructor that accepts two parameters. The first parameter represents the path of the directory, while the second parameter represents the file name. Lastly, $f3$ is created using File constructor, which accepts directory name and file name as the input parameter.

11.3.1 File Methods

There are many methods defined in the file class that can allow users to examine the properties of the file object. Some of them are list below, along with their addressed function in Table11.2.

Table 11.2: Methods of File

Method	Function
getName()	Returns the name of file.
boolean exists()	Returns true if file exists; otherwise, return false.
boolean canWrite()	Returns true if file is writable; otherwise, return false.
boolean canRead()	Returns true if file is readable; otherwise, return false.
boolean isFile()	Returns true if reference is a file and return false if reference is directories.
boolean isDirectory()	Returns true if reference is a directory; otherwise, return false.
String getAbsolutePath()	Returns application absolute path.
boolean renameTo(File newFileName)	Returns true if file name is renamed; otherwise, return false

To properly understand how the file object is created and how the file method can be accessed, we illustrate the concept by using the following program as an example.

```
//program
import java.io.File;
class DemoFile_1
{
   public static void main(String args[])
   {
      File fil_1 = new File ("/testfile_1.txt");
      System.out.println(" Display File name : " + fil_1.getName());
      System.out.println(" Specify the Path : " + fil_1.getPath());
      System.out.println("Absolute Path of file location: " + fil1.getAbsolutePath());
      System.out.println(fil_1.exists() ? "Do Exists" : "Doesn't exist");
      System.out.println(fil_1.canWrite()?"writable" : " not writable");
      System.out.println(fil_1.canRead() ? " file is readable" :"  file not readable");
      System.out.println("Size of file : " + fil_1.length() + " bytes");
   }
}
```

Output: (When testfile_1.txt does not exist)

```
Display File name: testfile_1.txt
Specify the Path: \testfile_1.txt
Absolute Path of file location : C:\testfile_1.txt
Doesn't exist
Not writable
file not readable
Size of file : 0 bytes
```

Output: (When testfile_1.txt exists)

```
Display File name : testfile_1.txt
Specify Path : \testfile_1.txt
Absolute Path of file location : C:\testfile_1.txt
Do Exists
writable
file is readable
Size of file : 17 bytes
```

The following program checks whether the filename FileCreated_1.txt is created or not. If the creation of the file is successful, then the output of program is "Creation of new File is successful!". The else part will be generated as output if the file FileCreated_1.txt is already present in the system.

```
import java.io.*;
public class FileCreation1
{
  public static void main(String[] args)
  {
    try
    {
      File f1_ex1 = new File("FileCreated_1.txt");
      if (f1_ex1.createNewFile())
      {
        System.out.println("Creation of new File is successful!");
      }
      else
      {
        System.out.println("File is already present.");
      }
    }
    catch (IOException fileexp)
    {
      fileexp.printStackTrace();
    }
  }
}
```

Output of the program: Creation of new File is successful!

The following program is used to check whether the folder jdk-15.0.1 consists of how many files and how many directories. The program use isDirectory() method for checking whether the inputted file is a directory or file. The list() method in the program is used to fetches all files and directories inside the folder jdk-15.0.1.

```
import java.io.File;
class DirCheckList
{
  public static void main(String args[])
  {
    File dir1 = new File("C:\\Program Files\\Java\\jdk-15.0.1");
    if (dir1.isDirectory())
    {
```

```
        System.out.println("Directory of " + "Java JDK");
        String str1[] = dir1.list();
        for (int i=0; i< str1.length; i++)
        {
          File chk_f1 = new File("C:\\Program Files\\Java\\jdk-15.0.1"+ "/" + str1[i]);
          if (chk_f1.isDirectory())
          {
            System.out.println(str1[i] + " is a directory");
          }
          else
          {
            System.out.println(str1[i] + " is a file");
          }
        }
      }
      else
      {
        System.out.println("C:\\Program Files\\Java\\jdk-15.0.1" + " is not a
          directory");
      }
    }
}
```

The output of program displays file in list and directory category.

```
Directory of Java jdk-15.0.1
bin is a directory
conf is a directory
COPYRIGHT is a file
include is a directory
jmods is a directory
legal is a directory
lib is a directory
release is a file
```

☞Check Your Progress-1

1) Why do we need Java I/O Package?

……

……

……

2) Write the name of any three interfaces provided in java.io with a description.

……

……

……

3) Write the functions of following methods: booleanexists(), booleancanRead(), booleanisDirectory()

……

……

……

……

4) Draw and explain the class hierarchy of Java I/O.

……

……

……

11.4 CLOSEABLE, FLUSHABLE, AND AUTOCLOSEABLE INTERFACES

Closeable interface: Closeable interface has only one abstract method i.e. close(). When a call to this method is invoked, system resources held by the stream object is released and these resources can be used further. Generally, stream classes implements Closeable interface and overrides close() method to close the stream. A sub class can use close() method, if its super class implements Closeable interface. E.g. close() method can be used by FileInputStream, because its super class InputStream implements Closeable interface.

Flushable interface: Like Closeable interface, Flushable interface also has only one abstract method i.e. flush(). When flush() method is called, data stored in the buffer is flushed out to be written to the destination file. Like Closeable interface, stream classes need to implement Flushable interface in order to override the flush() method.

Both close() and flush() method throws IOExaception.

AutoCloseable interface: Like Closeable interface, AutoCloseable interface also includes close() method, but this method is automatically called. An object that implements AutoCloseable interface, when it is done with the held resource within a try-with-resources block, close() method is automatically called.

11.5 I/O EXCEPTIONS

Exceptions refer to the interruption that breaks the normal flow of program execution. For example, if you try to access a file that is not present, it will result in a FilenotFound exception. The following are different types of I/O Exceptions listed in Table 11.3.

Table 11.3: Different I/O Exceptions

Sr.No.	Interface & Description
1	**CharConversionException** Represent the base class responsible for handling character conversion exceptions.
2	**EOFException** Generate exception when (unexpected) end of stream or end of file is encountered during input.
3	**FileNotFoundException** Thrown while opening a file using a specified file path, but the file path is not present/invalid.
4	**InterruptedIOException** Thrown when I/O operation is interrupted.

5	**InvalidClassException** This is thrown when a problem is detected with a Class during Serialization runtime.
6	**InvalidObjectException** Thrown when one or more deserializable objects failed validation test.
7	**IOException** Thrown when some sort of I/O exception occurs.
8	**NotActiveException** Thrown when serialization and deserialization is inactive.
9	**NotSerializableException** This is thrown when an instance variable of failed to have Serializable interface.
10	**ObjectStreamException** Represent superclass of all exceptions occurred in object stream classes.
11	**OptionalDataException** Throws exception when serialized object of stream performing in read operation result in failure either due to unread primitive data or data end.
12	**StreamCorruptedException** Thrown when control information read from object stream fails consistency checks.
13	**SyncFailedException** Thrown when sync operation failed.
14	**UnsupportedEncodingException** Thrown when the character encoding is not supported.
15	**UTFDataFormatException** Thrown when unsupported format data(not in UTF-8 format) is read by input stream or by any class that implement data input interface.

The following program demonstrates that an exception will be thrown if we try to read a file that does not exist.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class DemoFileNotFoundException
{
  public void checkFileExistance()
  {
    try
    {
      FileInputStream f = new FileInputStream("abc.txt");
      System.out.println("File Exists");
    }
    catch (FileNotFoundExceptionexceptionFileNotFound)
    {
      exceptionFileNotFound.printStackTrace();
    }
  }
  public static void main(String[] args)
  {
    DemoFileNotFoundException demo = new DemoFileNotFoundException();
    demo.checkFileExistance();
  }
}
```

Output:

```
java.io.FileNotFoundException: abc.txt (The system cannot find the file specifie
d)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at DemoFileNotFoundException.checkFileExistance(DemoFileNotFoundExceptio
n.java:6)
        at DemoFileNotFoundException.main(DemoFileNotFoundException.java:15)
```

11.6 INTRODUCTION TO STREAM CLASSES

As discussed above in the chapter, the java program uses the concept of the stream to perform I/O. A stream can be defined as an abstraction which either consumes or produces information. To perform I/O operationswith a physical device in Java, the system makes a link connection between the stream and physical device. Even though they are connected to different physical devices, all streams behaved in the same manner. Thus, it enables the input stream to obtain input from different input sources such as a disk file, keyboard, a network socket, etc.

Similarly, it allows the output stream to transmit output to different data destinations such as console, file disk, network connection. Stream provides an efficient way to perform input/output by the program without worrying about different types of physical devices. All the stream classes are implemented in the java.io package.

Byte Streams and Character Streams

In Java basically, there are two types of streams: Byte Streams and Character Streams. Byte streams are used to deal with input and output of binary or byte data, while Character streams deal with characters. Though all I/O are byte-oriented at the lowest level but character streams provide a convenient method for handling character using Unicode representation and can be easily internationalized making it more useful.

Java I/O classes are mainly built on four abstract classes i.e.InputStream, OutputStream, Reader, and Writer. The first two classes are categorized as Byte stream class, while the latter two of them are categorized as Character stream class. Byte stream classes are discussed in section 11.7, and Character stream classes are discussed in detail in section 11.8. In order to implement these classes, one must import java.io.

11.7 BYTE STREAM CLASSES

Byte Stream class provides a rich repository for the user that wants to perform byte or binary I/O operations. Any type of object can access these stream classes thus making these stream classes important for many programs. Byte Stream classes are further classified into two abstract class hierarchies i.eInputStream and OutputStream. Further, with these two top hierarchy abstract classes, several concrete subclasses are created, which ease the interaction when encountered with different types of devices such as files, network connection, buffer, etc. Fig 2. shows different types of stream classes. Later in the chapter few of these concrete subclasses are discussed.

Several key methods are defined in these two abstract classes that other types of stream classes can implement. For example, the read() and write() methods defined in InputStream and OutputStream are used for reading and writing byte data. These methods are abstract in nature and are overridden by any derived stream classes.

As InputStream class and OutputStream class are the top hierarchized class of byte stream, we start our discussion with InputStream class.

InputStream

InputStream is an abstract class that models the streaming of byte input. As it is a top-hierarchy abstract class, it defines numerous methods that are used/implemented by various other concrete subclasses. Table 11.4 describes the methods defined in the InputStream class. The methods (all of them) defined in InputStream class throws IOException in case of occurrence of error.

Table 11.4: Methods of InputStream

Method	**Descriptions**
int available( )	Returns the estimation of available number of bytes present for reading.
void close( )	Results in closing the input source and releases the associated system resources. Throws IOException if further attempts of read is made.
void mark(int *num_Bytes)*	Marks current point in the input stream, and it remains valid until the size of *num_Bytes* bytes are read.
Boolean markSupported( )	Returns **true** if the invoking stream supports **mark( )/reset( ) .**
int read( ).	Returns the byte of input stream that is next available in integer form if the end of the file is encountered during reading return -1.
int read(byte *buffer*[ ])	Returns the total size of byte that was successfully read from buffer when an attempt of reading bufferlength byte (input size) is made. If the end of the file is encountered during reading return -1.
void reset( )	Resets the current input pointer to previously set mark.
long skip(long *numBytes*)	Skips or Ignores input of size numBytes and return the number of bytes that are skipped.

OutputStream

The next top hierarchy abstract class is the Output stream class. It is an abstract class that models the streaming of byte output. The methods(all of it) defined in this abstract class are void in nature and throws IOException when an error occurs. The methods defined in this class are summarized in Table 11.5.

Table 11.5: Methods of OutputStream

Methods	**Description**
void close( )	Results in closing the output stream and releases associated system resources. Throws IOException if further attempts of write are made.
void flush( )	Clears or flushes the output buffers.
void write(int *b*)	Writes b bytes of data to output stream.
void write(byte *buffer*[ ])	Writes an array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *num_Byte*)	Writes an array of bytes of size num_Byte to output stream from the array *buffer* starting from *buffer*[offset].

Besides the two top hierarchy classes of Byte stream, other concrete subclasses are defined using the methods defined in InputStream and OutputStream class. Some of these subclasses are discussed as follows.

FileInputStream

The FileInputStream class allows users to read bytes from a file. To create an input stream of FileInputStream, the following constructor is used.

FileInputStream(String *pathFile*)

FileInputStream(File *objtFile*)

Both these constructors throw FileNotFoundException in case any exception occurs (when no file is found or invalid path of the file). Here, *pathFile* represents the full pathname of the referenced file, while *objtFile* represents the object of the referenced file. Let's see how to create FileInputStream using these two constructors with an example. In the following examples, both constructors use the same disk file.

FileInputStream fl_1 = new FileInputStream("/FileDemo_1.java")

File fl_1 = new File("/FileDemo_1.java");

FileInputStream fl_2 = new FileInputStream(fl_1);

As stated in the above example, the first constructor uses file path name while the second uses File object to read byte from file. Commonly the first constructor is used to create FileInputStream, but we can use the second constructor when we want to examine File methods before attaching input streams. Besides mark() and reset() methods of InputStream, this class overrides the other six methods of InputStream class. Any attempt to override mark() and reset() methods, results in the generation of IOException. As soon as FileInputStream is created, it automatically opened.

FileOutputStream

The FileInputStream class allows users to write bytes to a file. To create the output stream of File OutputStream, the most common constructor is defined using the following syntax:

FileOutputStream(String *pathFile*)

FileOutputStream(File *objtFile*)

FileOutputStream(String *pathFile*, boolean*app*)

These constructors throw either FileNotFoundException or a SecurityException. Here, pathFile represents the full pathname of the referenced file, while objtFile represents the object of the referenced file. If app is true in the third constructor, then the file is opened in append mode. When the user tried to write byte data to a read-only file, IOException will be thrown.

ByteArrayInputStream

ByteArrayInputStream creates an input stream that allows users to read data from byte array(input source). The class uses two types of constructors to create ByteArrayInputStream. In both these constructors, byte array is the source of input

data. The syntax of the constructor is of the form:

ByteArrayInputStream(byte *name_of_array*[ ])

ByteArrayInputStream(byte *name_of_array* [ ], int *start*, int *noBytes*)

Here, *name_of_array*is the source of input. In the second constructor, input stream is created from a subset of byte array,i.e.*name_of_array*, starting from index denoted by *start* parameter till the length of *noBytes.*

The following examples are to illustrate how ByteArrayInputStream is created using the above-discussed constructor.

```
// Demonstrating the creation of  byte input stream
import java.io.*;
class ByteArrayInputStream_Demo1
{
   public static void main(String args[]) throws IOException
   {
      String s1 = "howareyouinthishour";
      byte y_1[] = s1.getBytes();
      ByteArrayInputStream input_1 = new ByteArrayInputStream(y_1);
      ByteArrayInputStream input_2 = new ByteArrayInputStream(y_1, 0,3);
   }
}
```

The input_1contains howareyouinthishour, while input_2contains only how.

ByteArrayOutputStream

ByteArrayOutputStream creates an output stream that allows users to write data to a byte array(output destination). The class uses two types of constructors to create ByteArrayOutputStream. The syntax of the constructor is of the form:

ByteArrayOutputStream()

ByteArrayOutputStream(int *noBytes*)

From the first constructor, a buffer of 32-byte is created, while from the second constructor buffer of size *noBytes* is created. The count field of ByteArrayOutputStream class is used to denote the number of bytes stored in the buffer. The count field is protected in nature.

Following program demonstrates the use of ByteArrayOutputStream class

```
import java.io.*;
class DemoByteArrayOutputStream
{
   public static void main(String args[]) throws IOException
   {
      ByteArrayOutputStream op=new ByteArrayOutputStream();
      byte b[]= “Have a nice day”.getBytes();
      op.write(b);
      System.out.println(op.toString() );
      op.close(); //closing the stream
   }
}
```

Output: Have a nice day

Filtered Byte Streams

Filtered streams are an abstract class that supports basic input or output streams but with additional functionality. These streams are implemented by the class, which requires generic streams. The additional functionality here refers to buffering, character, and raw data translation. **Filter Input Stream** and **Filter Output Stream** are types of filtered byte streams.

The constructor of this class is of the form:

FilterInputStream(InputStream*is*)

FilterOutputStream(OutputStream*os*)

InputStreamand OutputStream methods and these classes' methods are similar in nature.

Buffered Byte Streams

Buffered streams extend the FilterStream class and allow buffering of byte I/O stream by attaching a buffer memory. Due to this buffering, it allows java to perform I/O operation on more than one byte simultaneously, thereby increasing the program's performances. Buffer also allows users to perform skipping, marking and resetting of input streams. BufferedInputStream and BufferedOutputStream are buffered byte stream classes. PushbackInputStream also implements the buffered stream.

BufferedInputStream

One of the common ways of performance optimization is achieved by buffering of I/O. In java, this performance is achieved using BufferedInputStream class that enables buffering of input streams. The constructor of this class is of the following form:

BufferedInputStream(InputStream *streamInput*)

BufferedInputStream(InputStream *streamInput*, int *size_Buf*)

In the first form of the constructor, a stream is created and buffered using the default size buffer while in the second form of the constructor , the stream is buffered in buffer of size *size_Buf*. For significant performance improvement, the buffer is usually taken in terms of a memory page, disk block.

Buffering also enables to trace stream in backward direction due to the storage of input. Hence, it supports mark() and reset() methods defined in Inputstream class. For examining this support, BufferedInputStream.markSupported**( )**is used that returns true when it supports the aforementioned methods.

BufferedOutputStream

The BufferedOutput stream behaves similarly to other OutputStream but with an additional flush() method. The flush() method defined here confirms/ensures that the data in buffers are physically written to actual output devices.
BufferedOutputStream also tries to improve the performance by decreasing the number of times writes data.

The following are the form of its constructor:

BufferedOutputStream(OutputStream **streamOutput**)

BufferedOutputStream(OutputStream **streamOutput**, int **buf_Size**)

The first one creates a buffer of 512 bytes for a write operation of output streams, while the second one creates a size of the buffer **buf_Size** for writing output streams.

PushbackInputStream

Buffering allows pushback operations in java. Pushback allows a byte to read from the input stream and then returns this byte to the input stream. It allows users to see what is coming from the input stream without disturbing the flow of the input stream. In java, the pushback of byte is implemented via PushbackInputStream class.

The following are the form of constructor defined in PushbackInputStream:

PushbackInputStream(InputStream *input_stream*)

PushbackInputStream(InputStream *input_stream,* int *num_Bytes*)

The first constructor creates a stream object that allows readied byte to return to the input stream. The second constructor creates a buffer of multiple data of size num_Bytes and returns it to the input stream.

SequenceInputStream

When a user wants to concatenate multiple Input streams, it can use **SequenceInputStream** class. The constructor of this class either uses a pair of InputStream or Enumeration of InputStream class. Thefollowing are the form of the constructor:

SequenceInputStream(InputStream *first_Stream,* InputStream *second_Stream*)

SequenceInputStream(Enumeration *stream_Enum*)

In the first case, the class starts reading data from the first InputStream until its end and then switches to the next or second InputStream for reading. While in the second case i.e. Enumeration, it will continue reading data until the last stream end is encountered.

PrintStream

The PrintStream class offers all types of formatting capabilities for the stream as used from the starting of books,i.e. as in System.out and system filehandle.

The following are its constructor form:

PrintStream(OutputStream *output_Stream*)

PrintStream(OutputStream *output_Stream*, boolean *flush_On_Newline*)

Here *flush_On_Newline* controls the flushing of the output stream whenever a

newline(\n) is encountered. *If flush_On_Newline* results in false, flush operation is not invoked automatically; else, it is automatically invoked.

These class objects support both print() and println() methods to accept all types of arguments /parameters, including objects. In both methods, if parameters are not of simple type, a call toString() method is invoked, and then the result is printed.

RandomAccessFile

When a user wants to randomly access a file, it can use RandomAccessFile class. This class is not derived from either InputStream or OutputStream class. It implements the DataInput and DataOutput interfaces defined in I/O methods. The class also supports the feature of positioning the file pointer. The defined constructor is of the following forms:

RandomAccessFile(File *file_Obj1*, String *mode_Access*) throws FileNotFoundException

RandomAccessFile(String *name_file*, String *mode_Access*) throws FileNotFoundException

In the first constructor, *file_Obj1* represents the name of the file to be open as a File object while in the second constructor,*name_file* represents the name of the file to be opened. The mode_Access here represents the type of access permitted regarding file access. If 'mode_Access =r', the file can only be read but cannot be written. If 'mode_Access =rw' then the file can be used for both reading and writing operations. If 'mode_Access =rwd', the file is eligible for both read and write operations, and the changes made to files are directly written to the physical device.

To set the file pointer to the current point in the file, seek() method is defined.

void seek(long *new_Pos*) throws IOException

Here, the file pointer is set to a new position, i.e. *new_Pos from the starting* of the file. After this read or write operation is performed from this new position,another method *setLength()* is used to increase or decrease the length of the file.

The following program writes a string "new end" at the end of the file,i.e. append a string "new end" using RandomAccessFile class.

```
import Java.io.*;
public class DemoRandomAccessFile
{
  public static void main(String args[])
  {
    try
    {
      RandomAccessFile rf = new RandomAccessFile("xyz.txt","rw");
      rf.seek(rf.length());
      rf.writeBytes("\n new end \n");
    }
    catch (IOException e)
    {
      System.out.println("Error Opening a file" + e);
    }
  }
}
```

Ouput: Above program will append "new end" in xyz.txt file.

☞Check Your Progress-2

1) What is the difference between Byte Streams and Character Streams?

……

……

……

2) What is the difference between closeable and autocloseable interfaces?

……

……

……

3) Which of the following classes is used for input and output in Byte streams?

(a) FileInputStream (b) BufferedReader (c) BufferedWriter (d) File

……

……

……

4) Which of the following class supports print() and println() methods?

(a) System (b) BufferedOutputStream (c) PrintStream (d) System.out

……

……

……

5) Write a program for reading data from two or more input stream using Sequnce InputStream class.

……

……

……

11.8 CHARACTER STREAM CLASSES

Character Streams

Even though the byte stream classes are sufficient for handling any type of I/O operation, but the limitation of these classes is that they never directly work with Unicode characters. So, to support one of the main features,i.e. write once and run anywhere of java, support for direct character-oriented I/O is vital. Thus, in this section discussion of different character I/O classes is carried out. As seen earlier in this chapter, character stream classes are top hierarchized by two abstract classes,i.e. Reader and Writer. Many other stream classes use the methods defined in these classes. For example, read() and write() defined here are used by other stream classes to perform read and write character operations. So, let's begin with Reader abstract class.

This is an abstract class that models the streaming of character input in Java. IOExceptions are thrown by every method defined in this class on the occurrence of any error. The methods of this class are described in Table 11.6.

Table 11.6: Methods of Reader Class

Methods	**Description**
void close( )	Performs closing of source input. Throws I/OException if any further read attempt is carried out.
void mark(int *num_Chars*)	A pointer (marker) is placed at current point in input stream, and the pointer is valid until num_ Chars are read.
boolean markSupported( )	Returns true if stream supported mark()/reset().
int read( )	Reads a character from the input stream and return its integer representation when file end is encountered return -1.
int read(char *buffer*[ ])	Returns the actual number of characters that are successfully read when reading of characters of buffer.length is attempted in buffer, when file end is encountered return -1.
abstract int read(char *buffer*[ ], int *offset*, int *num_Chars)*	Returns the number of characters that were successfully read when the attempt of reading *num_Chars* character into buffer is performed starting from the *buffer [offset]*. When file end is encountered during reading return -1.
boolean ready( )	Returns false if the next input request is not ready(waiting) else, return true.
void reset( )	The input pointer is reset to the previously set mark.
long skip(long *noChars*)	Returns the number of characters that are successfully skipped when skip of noChars input character is carried out.

Writer Class:

This is an abstract class that models the streaming of character output. The methods defined in this class are void in nature and throws IOException when an error occurred. The methods defined in this class are described in Table 11.7.

Table 11.7: Methods of Writer Class

Methods	**Description**
abstract void close( )	Used to perform the closing of the output stream. Throws I/OException if any further write attempt is carried out.
abstract void flush( )	Flushes the output buffers.
void write(int *ch*)	Used to perform writing of a single character to the invoked output stream.
void write(char *buffer*[ ])	An array of stream is written to the invoked output stream.

abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	A subarray of the character of length num_Chars from buffer array starting from buffer[offset] is written to the invoked output stream.
void write(String *str*)	str is written to the invoked output stream.
void write(String *str*, int *offset*, int *num_Chars*)	A subrange of*num_Chars*character from array str starting from str[offset] is written to the invoked output stream.

FileReader

The FileReader class allows users to read the contents (input stream) from the file. Syntax of two of its most used constructors are as follows:

FileReader(String *pathFile*)

FileReader(File *objtFile*)

Here *pathFile* represents the path where the file is located, while *objtFile* represents the File object of the invoked file.

In order to demonstrate how to read from a file is described using the following program. The program demonstrates reading of line from file and printing to the output stream.

```
// Demonstrating the reading of file using FileReader
import java.io.*;
class FileReader_Demo1
{
  public static void main(String args[]) throws Exception
  {
    FileReader rf_1 = new FileReader("FileReader_Demo1.java");
    BufferedReader rb_1= new BufferedReader(rf_1);
    String str;
    while((str = rb_1.readLine()) != null)
    {
      System.out.println(str);
    }
    rf_1.close();
  }
}
```

FileWriter class

The class allows users to write characters (input stream) to file. The most common constructors defined in this class are of the following syntax:

FileWriter(String *pathFile*)

FileWriter(String *pathFile,* boolean*app)*

FileWriter(File *objtFile*)

FileWriter(File *objtFile,* boolean*app*)

These constructors defined here throws **IOException.** Here, *pathFile* represents full pathname of the referenced file while *objtFile* represents the object of the referenced file. If *app* is true, then the file is opened with appended output. FileWriter first creates a file by creating an object and then open it for output writing. When a user tries to open read-only file, IOException will be thrown.

CharArrayReader

The class provides another method for users to read the input stream (character) from character array (Input source). The constructor of this class uses character array as a source of data reading. The form of constructors is as follows:

CharArrayReader(char *name_ Array*[ ])

CharArrayReader(char *name _Array*[], int *begin*, int *num_Chars*)

Here *name_ Array* is the data source, while in the second constructor, a subset of character of length *num_Chars* is read from *name_ Array* starting with index specified by begin parameter.

CharArrayWriter

The class provides another method for users to write the character stream (character)to the character array (data destination). The constructor of this class uses a character array as the destination of data writing. The form of constructor is as follows:

CharArrayWriter( )

CharArrayWriter(int *num_Chars*)

In the first constructor, a default size buffer is created for performing write operations, while in the second constructor, a buffer size equal to *num_Chars* size is created. CharArrayWriter uses buf field to denote/hold buffer information. Automatically the size of the buffer increases when required. A count field defined in the class is used to keep track of the number of characters in the buffer. Both these fields are protected fields.

BufferedReader

The class enables the java program to increase its performance by attaching a buffer to the input stream. The buffer allowsthe user to hold character and allows it to perform I/O operation on more than one character at a time.The following are the form of this class constructor.

BufferedReader(Reader *in_Stream*)

BufferedReader(Reader *in_Stream*, int *buf_Size*)

The first constructor creates buffered character stream using default buffer size, while in the second constructor the size of the buffer is set to *buf_Size*for creating buffered character streams. Since we have seen that buffering enable moving stream in the backward direction by implementing mark() and reset() methods and return true when checked by calling **BufferedReader.markSupported( ).**

The following program demonstrates the reading of data using BufferedReader class.

```
import java.io.*;
public class BufferedReaderDemo
{
  public static void main(String[] args) throws IOException
  {
    InputStreamReaderabc = new InputStreamReader(System.in);
    BufferedReaderxyz = new BufferedReader(abc);
    String str = "";
    System.out.print("Please enter a string: ");
    str = xyz.readLine();
    System.out.println("Entered string is: " + str);
  }
}
```

Output:

Please enter a string: abcdef

Entered string is: abcdef

BufferedWriter

The BufferedReader stream behaves in a similar fashion as other Writer class but with additional flush() method. The flush() method defined here makes the data in buffers to be physically written to the actual output stream. BufferedWriter also tries to improve the performance by decreasing the number of times actually writes operation is performed.

The following are the form of these constructors:

BufferedWriter(Writer *output_Stream*)

BufferedWriter(Writer *output_Stream*, int *buf_Size*)

The first one creates a buffer of 512 bytes for write operation of output character streams, while the second create a size of the buffer *buf_Size*for writing output streams.

The following program demonstrates the writing of data to a file using BufferedWriter class.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
public class DemoBufferedWriter
{
  public static void main(String args[])
  {
    String str = "This is the output file";
    try
    {
      FileWriter f = new FileWriter("output.txt");
      BufferedWriter op = new BufferedWriter(f);
      op.write(str);
      op.close();
    }
    catch (Exception e)
    {
      e.getStackTrace();
```

```
    }
  }
}
```

PushbackReader

Buffering enables pushback operations in java. PushbackReader allows one or more character to read from input stream and then return these characters to input stream. Basically, it allows users to see what is coming from input stream without disturbing the flow of input stream. In java, push back of character is implemented via PushbackReader class. The following are the form of its constructor:

PushbackReader(Reader *inStream*)

PushbackReader(Reader *inStream*, int *bufSize*)

In the first constructor,the first character is read and push backed to in_Stream, while the second constructor allows pushback of stream equal to the size of*bufSize*. One must check whether the pushback buffer is full or not before returning a character. If the pushback buffer is full, it results in IO Exception. If we want to return one or more than one characters to the input stream, we can use unread() method of PushbackReader.

PrintWriter

PrinterWriter works similar to Print Stream and can be thought of as character oriented version of PrintStream. It provides all types of formatting functionality by defining print() and println() methods. The following are its Constructor forms:

PrintWriter(OutputStream *output_Stream*)

PrintWriter(OutputStream *output_Stream*, boolean *flushOn_Newline*)

PrintWriter(Writer *output_Stream*)

PrintWriter(Writer *outputStream*, boolean *flushOn_Newline*)

Where flushOn_*Newline* controls flushing of output whenever println() is invoked. Flushing of output is done automatically when *flushOn_Newline*is true else flushing is not automatic.This class objects provide support for both print() and println() methods to accept all types of arguments /parameters, including object also. In both methods, if parameters passed are not of simple type, a call toString() method is invoked, and then the result is printed.

11.9 SERIALIZATION

Java not only allows users to read and write text but also provides support to read and write objects of file. With serialization, users can write the state of objects to a byte stream. Serialization is useful to users as they allow users to save the content of their program in any non-volatile storage (file, memory) and later allows the users to restore these objects when required or necessary. The restoring process is known as Deserialization.

In Java, Remote Method Invocation (RMI) is implemented with the help of Serialization. With RMI in Java, the object of one machine can invoke methods of another object located on a different machine. In RMI, the object is passed as an

argument to the invoked remote method. Before passing this object as a parameter, serialization is used to serialize the passed object and then transmit it. On receiving the serialized object, the destination machine deserialize it.

One important question that arises is how serialization in Java is easy as compared to serialization in another language. For instance, suppose an object has a reference to another object, and the referenced object has a reference to any other object. In some cases, they may form circular references among objects creating a lot of confusion or chance of inconsistency. Java efficiently handles this situation by doing the following things. If you serialize one object, all its referenced objects are recursively located and serialized. Similarly, when performing deserialization all its referenced objects get restored correctly in the same order. In java if you want to serialize an object, you have to call the writeObject() method defined in ObjectOutputStream, and when you want to deserialize any object, invoke the readObject() method defined in ObjectInputStream. Serialization also enables transferring of object state over the network i.e Marshalling of objects. Implementation of a Serializable interface is also necessary for the serialization of objects.

The following are the interfaces and class that supports serialization.

Serializable

The object that implements Serializable interfaces can only be saved and restored using the serialization process. No members are defined in the Serializable interface. To serialize the object of the class, the class must be declared as public and the class must have no parameter constructor. All subclass of the serializable class is also serializable. Transient and static variables are not saved by serialization facilities. java.io.Serializable interface is by default implemented by the string and all wrapper class. In the following example, Employee class implements Serializable interface. Now object of Employee class can be converted into stream.

```
import java.io.Serializable;
public class Employee implements Serializable
{
   int e_id;
   String e_name;
   public Employee(int e_id, String e_name)
   {
      this.e_id = e_id;
      this.e_name = e_name;
   }
}
```

Externalizable

In java, serialization and deserialization have automatically made the state of an object to be stored and restored. But in some cases, the programmer desires to control this automatic process. For example, when implementing encryption and compression techniques, the programmer can use Externalizable interface to control these automatic processes.

ObjectOutput:

The interface extends the DataOutput interface and provides support for object serialization, For making object serializable, call to writeObject() method defined in this interface is invoked. Methods defined in this interface throws IOException in case of error occurrence.

ObjectOutputStream:

This class implements the ObjectOutput interface along with extending OutputStream class for writing serializable objects into the output stream. The following is the form of this class constructor:

ObjectOutputStream(OutputStream **out_Stream**) throws IOException

Here the serializable objects are written to the output stream,i.e.out_Stream.

ObjectInput:

The interface extends the DataInput interface and provides support for object deserialization. For making object deserializable, call to readObject() method defined in this interface is invoked. The method defined in this interface throws IOException in case of error occurrence.

ObjectInputStream:

This class implements the ObjectInput interface along with extending OutputStream class for reading serializable objects from the input stream. The following is the form of this class constructor:

ObjectInputStream (InputStream*in_Stream*) throws IOException, Stream Corrupted Exception

Here from the input stream,i.e.*in_Stream*, the serializable objects are read.

For an illustration of how object serialization and deserialization takes place, we use the following program as an example. First, an object of class MyClass is instantiated. The objects define three instance variables of types int, String, and double. For this example, we tried to save and restore the information of these three-instance variables.

First, we create FileOutputStream with the file name “DemoSerial”. Then ObjectOutputStream is created for the created file stream. Then for object serialization, we call writeObject() method defined in ObjectOutputStream. After serialization, it's flushed and closed.

Now we create FileInputStream by referring to the same file,i.e. “DemoSerial”. Then we create ObjectInputStream for the respective input stream. Then we perform deserialization of object using readObject() method of ObjectOutputStream class. Then this object input stream is closed.

Here we defined MySeriClass that implements Serializable interface. In case MySeriClass does not implement Serializable, it will throw Not Serializable Exception error.

```
import java.io.*;
public class DemoSerialization1
{
  public static void main(String args[])
  {
    // Object Serialization
    try
    {
```

```
            MySeriClass objt_1 = new MySeriClass("Welcome Ram", 85, 99.5);
            System.out.println("First object: " + objt_1);
            FileOutputStream filestr_1 = new FileOutputStream("DemoSerial");
            ObjectOutputStream objtstr_1= new ObjectOutputStream(filestr_1);
            Objtstr_1.writeObject(objt_1);
            Objtstr_1.flush();
            Objtstr_1.close();
        }
        catch(Exception exp)
        {
            System.out.println("Catching Exception: " + exp);
            System.exit(0);
        }

        // Deserialization of object
        try
        {
            MySeriClass objt_2;
            FileInputStream filestr_2= new FileInputStream("DemoSerial");
            ObjectInputStream objtstr_2 = new ObjectInputStream(filestr_2);
            objt_2 = (MySeriClass)objtstr_2.readObject();
            objtstr_2.close();
            System.out.println("Second object: " + objt_2);
        }
        catch(Exception exp)
        {
            System.out.println("Catching Exception : " + exp);
            System.exit(0);
        }
    }
}
class MySeriClass implements Serializable
{
    String str;
    int a1;
    double db;
    public MySeriClass(String str, int a1, double db)
    {
        this.str = str;
        this.a1 = a1;
        this.db = db;
    }
    public String toString()
    {
        return "str=" + str + "; a1=" + a1 + "; db=" + db;
    }
}
```

Output:

First object: str=Welcome Ram; a1=85; db=99.5

Second object: str=Welcome Ram; a1=85; db=99.5

The same output of object instance variables here represents successful serialization and deserialization operation.

☞Check Your Progress-3

1) What are Serialization and Deserialization?

……

……

……

2) What is the role of Externalizable?

……

……

……

3) Which of the following classes is used for input and output in Character streams?

(a) BufferedReader (b) FileInputStream (c) FileOutputStream (d) InputStream

……

……

……

4) Which of the following exceptions, read() method throws?

(a) Interrupted Exception (b) IOException (c) System Input Exception (d) SystemException

……

……

……

5) Write a program to read only first five character from character array C_arr[]={'E','x','c','e','l','l','e','n','t'} using CharArrayReader Class. Also write a separate program to the write the content of character array CharArr[] to a file name "CharWriteClass.txt" using CharArrayWriter Class.

……

……

……

11.10 SUMMARY

Generally, java-based applications process some inputs and, based on the inputgenerates output. The source of input and the destination of output may be a file, physical devices, network, etc. Java provides an API, JAVA IO, which helps in reading and writing the data,i.e. input and output. This chapter provides a clear view of the Java input/output hierarchy. This chapter also explained how various classes and interfaces in the IO package are organized and their purpose. Further, the File class is discussed that helps in handling files and directories. Various IO exceptions are discussed and summarized. Java supports two types of streams: Byte Streams and Character Streams. Both streams and associated classes are discussed in the chapter. This chapter discusses serialization that allows us to represent an object in a sequence of bytes,i.e. byte stream.

11.11 SOLUTION/ANSWERS TO YOUR PROGRESS

☞Check Your Progress-1

1) Java I/O Package consists of input streams and output streams. The Input stream is used to read data from input source, while the Output stream helps to write data to destination. Here source and destination can be anything that generates, consumes and stores the data such as program, disk, peripheral device, etc.

2) DataInput:Allow reading of byte data from the binary stream and converting these data to any of Java primitive types.

   FileFilter: Provides filter for abstract pathnames.

   ObjectOutput: Extends DataOutput interface for the writing of objects and allow serialization of objects

3) booleanexists(): Returns true if file exists otherwise return false.

   booleancanRead(): Returns true if file is readable otherwise return false.

   booleanisDirectory(): Returns true if reference is a directory otherwise return false.

4) The diagram of class hierarchy of I/O is shown in figure 2 in section 11.2.

☞Check Your Progress 2

1) Byte streams process the data byte by byte, while Character streams process the data character by character in Unicode.

2) Closeable interface has only one abstract method i.e. close(). When a call to this method is invoked, system resources held by stream object are released, which can be used further in the program. Like Closeable interface, AutoCloseable interface also includes close() method but this method is automatically called unlike Closeable interface.

3) FileInputStream

4) PrintStream

5) In order to show how to perform concatenation of multiple input streams, a program is written. The program concatenates two file input streams named BufferReadFile.txt and ReadFile.txt, respectively. The SequenceInputStream first read the input from BufferReadFile.txt and then read input from ReadFile.txt (sequentially one by one in the order mentioned in the program).

```
import java.io.*;
class SequenceInStreamEx
{
  public static void main(String args[])throws Exception
  {
    FileInputStream fin_str1 =new FileInputStream("D:\\BufferReadFile.txt");
    FileInputStream fin_str2=new FileInputStream("D:\\ReadFile.txt");
    SequenceInputStreamcon_seq=new SequenceInputStream(fin_str1, fin_str2);
    int re;
    while((re=con_seq.read())!=-1)
    {
      System.out.print((char)re);
```

```
    }
    con_seq.close();
    fin_str1.close();
    fin_str2.close();
  }
}
```

☞Check Your Progress 3

1) With serialization, users can write the state of objects to a byte stream. Serialization is useful to users as they allow users to save the content of their program in any non-volatile storage (file, memory) and later allows the users to restore these objects when required or necessary. The restoring process is known as Deserialization.

2) Externalizable is used to customize the serialization. In java, serialization and deserialization have automatically made the state of an object to be stored and restored. But in some cases, the programmer desires to control this automatic process. For example, when implementing encryption and compression techniques, the programmer can use Externalizable interface to control these automatic processes.

3) BufferedReader

4) IOException

5) Program 1: For reading character using CharArrayReader Class

```
import java.io.CharArrayReader;
public class ProgressCheck3
{
  public static void main(String[] ag) throws Exception
  {
    char[] C_arr = {'E', 'x', 'c', 'e', 'l', 'l', 'e', 'n', 't' };
    CharArrayReader ch_rd1 = new CharArrayReader(C_arr,0,5);
    int k = 0;
    // Read until the end of a file
    while ((k = ch_rd1.read()) != -1)
    {
      char ch_read = (char) k;
      System.out.println(ch_read);
    }
  }
}
```

Output:

```
E
x
c
e
l
```

Prcogram 2: For writing content of C_arr to file name “CharWriteClass.txt” using CharArrayWriter Class.

```
import java.io.CharArrayWriter;
import java.io.FileWriter;
public class CharProgramWrite3
{
   public static void main(String args[])throws Exception
   {
```

```
        char[] C_arr = {'E', 'x', 'c', 'e', 'l', 'l', 'e', 'n', 't'};
        String conv_str1 = new String(C_arr);
        CharArrayWriter arr_wr1 =new CharArrayWriter();
        arr_wr1.write(conv_str1);
        FileWriter new_f1=new FileWriter("D:\\CharWriteClass.txt");
        arr_wr1.writeTo(new_f1);
        new_f1.close();
        System.out.println("Successfully Written");
    }
}
```

Output: Successfully Written

11.12 REFERENCES /FURTHER READING

- Herbert Schildt “Java The Complete Reference”, McGraw-Hill,2017
- Horstmann, Cay S., and Gary Cornell, “ *Core Java: Advanced Features*” Vol. 2, Pearson Education, 2013.
- https://docs.oracle.com/javase/tutorial/essential/io/

ignou THE PEOPLE'S UNIVERSITY

UNIT 12 JAVA API

Structure **Page No.**

12.0 Introduction 309

12.1 Objectives 310

12.2 Date and Time 311

12.3 Set 314

12.3.1 Classes that implement Set interface 315

12.3.2 Interface that extends Set interface 319

12.4 Map 321

12.5 HashMap 323

12.6 List 328

12.7 Vector 329

12.8 Stack 334

12.9 Summary 336

12.10 Solutions/ Answer to Check Your Progress 336

12.11 References/Further Reading 339

12.0 INTRODUCTION

In this unit, you are going to learn about Java Application Programming Interface (API) for Date, Time and theCollections. The Java API is a group of pre-written packages, classes, interfaces along with their methods and attributes. You can import a particular package or a class as per your requirement and use its pre-written code in your program to perform basic and advanced programming tasks. This, in turn, reduces the lines of code of your program. Java API is a component of Java. Development Kit. Java 8 introduced a completely new API in the java.time package that comprises a large number of classes to perform various Date and Time operations.

Collections API in Java is used for storing and manipulating a group of objects. The Collection framework is contained in java.util package. Students are advised to refer to Figure 1 in order to understand this section on the Introduction of Java API. The Iterable interface is the root interface of all collection classes and is extended by the Collection interface. All derived classes of collection interface implement the Iterable interface.

The Collection framework comprises interfaces like Collection (Figure 12.1) and Map (Figure 12.2). List, Set and Queue are the subinterfaces of the Collection interface, as shown in Figure 12.1. For the implementation of the List subinterface, there are Standard classes such as ArrayList, LinkedList and Vector. Further, the Vector class is inherited by Stack class. Next, for implementation of the Set subinterface, there are Standard classes such as, HashSet, LinkedHashSet andTreeSet. For the implementation of Queue subinterface, PriorityQueue class and ArrayDeque class are used.

There are few classes that provide an implementation of the Map interface. Student should refer Figure 12.2,which shows hierarchy of Map interface. AbstractMap class provides an implementation of the Map interface, and further HashMap class

extends AbstractMap class in order to use a hash table. As shown in Figure 12.2, the SortedMap interface extends the Map interface and is inherited by the TreeMap class.

Finally, as discussed earlier Collection interface extends the Iterable interface, which is the root interface of all collection classes. The iterator() method of the Iterable interface facilitates the traversal of the elements in a collection. You can perform searching, sorting, insertion, and deletion easily and efficiently with Java Collections.

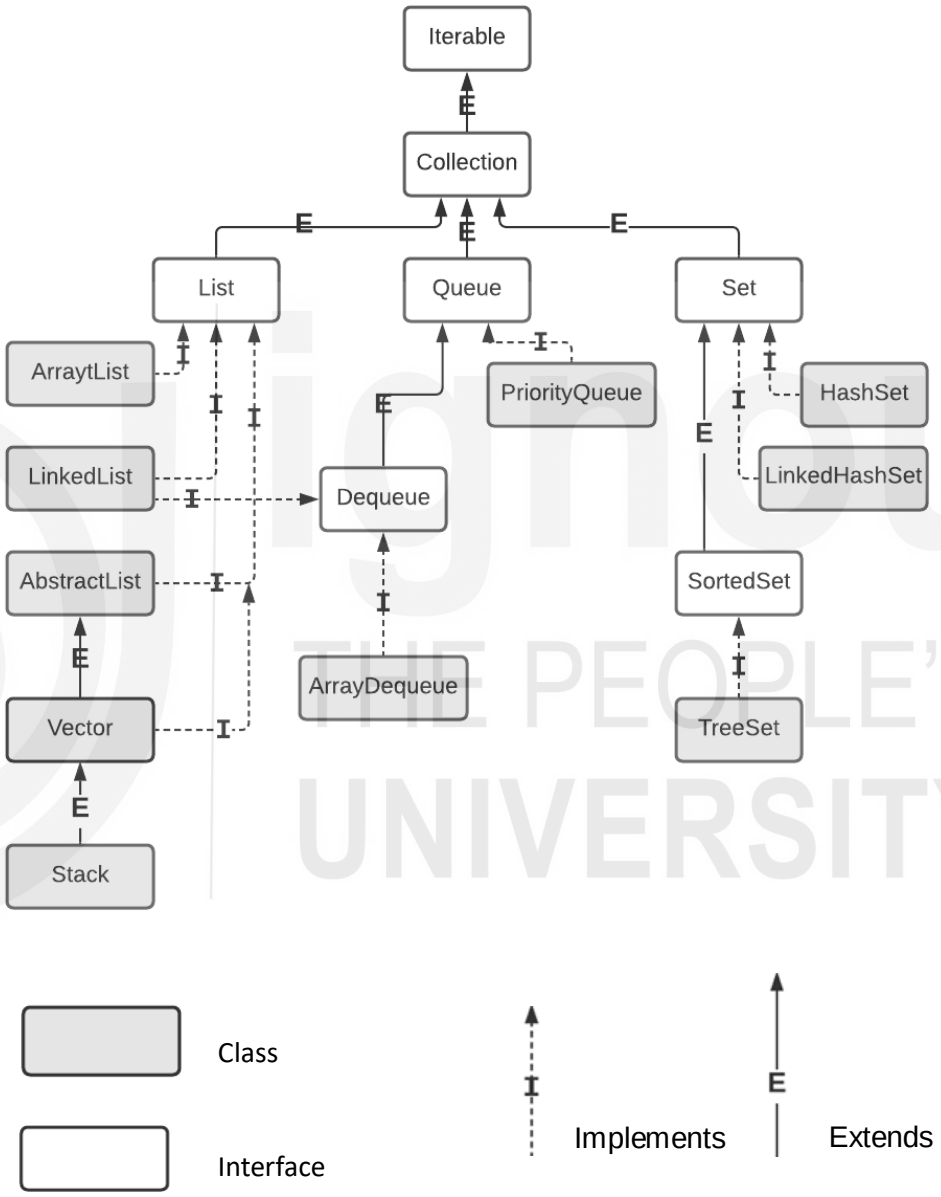


Figure 12.1: Collection Hierarchy in Java

12.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the benefits of Java APIs,
- write programs that compare different date and time values,
- describe the use of various interfaces and classes of collection framework,

- use various interfaces, namely Set, Map and List, in order to write programs in a simpler way,
- use Traversal of Set using Iterator, and
- write programs by using the methods to perform various operations on maps.

12.2 DATE AND TIME

The older Date class before Java 8, was provided by java.util package, and it offered various functions related to date and time, as described in Table 12.1. There are two constructors for the Date class as given below: -

- Date( ) : It constructs Date object and initializes it with the current date and time.
- Date(long millisec) : It receives an argument corresponding to the quantity of milliseconds that have passed after twelve midnight, 1st January, 1970.

Table 12.1: Date class and its methods

S.No.	Method	Description
1	boolean after(Date dt)	It gives true when the Date object that invoked **after**the method has a value of date which is later than the one stated by object dt, else gives false.
2	boolean before(Date dt)	Gives true when the Date object that invoked **before**the method has a value of date which is before the one stated by object dt, else gives false.
3	Object clone( )	Returns duplicate object of the Date object that invoked clone method.
4	int compareTo (Date dt)	Compares the value date of invoking object to the value of date of dt object. If both are equal, this method returns 0. It returns a negative value if the value of date of invoking object is earlier than that of dt object. It returns a positive value if the value of date of invoking object is after the date of the dt object.
5	int compareTo (Object ob)	This function is identical to compareTo(Date) provided ob is an object of class Date. If not, an exception named ClassCastException is thrown.
6	boolean equals (Object dt)	Gives true when the Date object that invoked **equals** method has the same time and date value as mentioned in dt object, else the method gives false.
7	long getTime( )	It gives value of time in terms of number of milliseconds passed since 1st January 1970 (midnight).
8	int hashCode( )	Gives the hash code of the object that invoked this method
9	void setTime(long t)	Sets the current time and date as stated by t. The time in long corresponds to the time in milliseconds that have passed since midnight, 1st January, 1970.
10	String toString( )	Changes the Date object that invoked this method into a string, and the same is returned.

```
import java.util.Date;
class DateEx
{
   public static void main(String args[])
   {
    // Create an object of Date class
    Date date = new Date();
    // Print time and date
    System.out.println(date);
    // Retrieve the  number of milliseconds since midnight, January 1, 1970 GMT
    long msec = date.getTime();
    System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
    // Create a Date object with msec value (as long datatype)as passed to constructor
    Date date1 = new Date(1599117817589l);
    // Retrieve the number of milliseconds in date1 object
    long msec1 = date1.getTime();
    // Compare current date with date1 object and print message accordingly
    int result = date.compareTo (date1);
    if (result==0)
    System.out.println("Both dates are same");
    else if (result==1)
    System.out.println(msec+" is later than "+msec1);
    else
    System.out.println(msec+" is earlierr than "+msec1);
   }
}
```

Output:

```
Tue Nov 03 12:04:32 IST 2020
Milliseconds since Jan. 1, 1970 GMT = 1604385272113
1604385272113 is later than 1599117817589
```

Date Formatting Using SimpleDateFormat

In order to give an understandable format for date, there is a class called Simple DateFormat. This class allows us to choose any pattern for displaying date-time. You can also change a string to date by means of an appropriate method of this class.

```
import java.util.*;
import java.text.*;
public class Date_form
{
  public static void main(String args[])
  {
    Date date = new Date( );
    SimpleDateFormat date_format = new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
    System.out.println("Current Date: " + date_format.format(date));
  }
}
```

Output:

```
Current Date: Tue 2020.11.03 at 12:16:55 PM IST
```

There are various codes for SimpleDateFormat. These are given below in Table 12.2

Table 12.2: SimpleDateFormat codes

Character	Description	Example
G	Era designator	AD
y	Year in four digits	2001
M	Month in year	July or 07
d	Day in month	10
h	Hour in A.M./P.M. (1~12)	12
H	Hour in day (0~23)	22
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)
w	Week in year	40
W	Week in month	1
a	A.M./P.M. marker	PM
k	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	10
z	Time zone	Eastern Standard Time
'	Escape for text	Delimiter
"	Single quote	`

This Date-Time API was not thread-safe and offered very few operations related to date. With the advent of Java 8, these drawbacks were removed as a new Date-Time API was introduced, which is immutable, doesn't have setter methods and includes many date operations. Two important classes present in this API are mentioned below:

- LocalDate: It is a simplified date-time API, and there is not any complexity of timezone handling.
- ZonedDateTime: It is a special date-time API that makes you deal with various timezones in an easy manner.

☞Check Your Progress-1

1) What do you understand by the Collection framework in Java?

……

……

……

……

2) Which of these classes is not part of Java's collection framework?

a) Array

b) Maps

c) Stack

d) Queue

3) How do you know if two date objects are equal?

……

……

……

……

12.3 SET INTERFACE

We can define a set as a group of dissimilar objects. We all know that sets are one of the most fundamental concepts in mathematics. Java provides Set as an interface that extends the Collection interface, and hence it has access to all the methods inherited from the Collection interface, as explained in Table 12.3.

Table 12.3 Description of methods of Collection interface

No.	Method	Description
1	public boolean add(object o)	It is written to insert an object in the collection. Returns true if object is added.
2	public boolean addAll(Collection c)	The purpose is to insert the elements of the mentioned collection c, in the collection that invoked the addAll method. The addAll() method returns true if the collection got changed because of the method call otherwise, false is returned.
3	public boolean remove(Object o)	The purpose of this method is to remove an object from a particular collection that invoked this method. It returns true if object is removed.
4	public boolean removeAll (Collection c)	The purpose of this method is to remove all the elements of the mentioned collection c, from the collection that invoked this method. The removeAll() method returns true only if the invoking collection was changed otherwise, false is returned.

5	default boolean removeIf (Predicate filter)	The purpose of this method is to remove all the elements of the collection that satisfy the given predicate. The parameter 'filter' represents a predicate which returns true for the elements to be removed. This method returns true if we are able to remove elements.
6	public boolean retainAll (Collection c)	The purpose of this method is to remove all the elements of invoking collection except the mentioned collection c. It returns true if it is successful.
7	public int size()	The purpose of this method is to tell the count of elements of the invoking collection.
8	public void clear()	This method deletes all elements of the mentioned collection.
9	public boolean contains(Object o)	With this method, we can search an object in the collection. The method will give true when the object is found.
10	public boolean containsAll(Collection c)	With this method, we can search the stated collection in the invoking collection. Again, the method will give true when the collection c is found.
11	public Iterator iterator()	It gives an iterator over elements of the invoking collection.
12	public Object[] toArray()	It changes the invoking collection to an array and returns the same.
13	public <T> T[] toArray(T[] a)	It changes the invoking collection to an array and returns the same. In this method, the returned array's runtime type is same as that of the mentioned array.
14	public boolean isEmpty()	This method would test whether the invoking collection is empty and returns true if it is so.
15	public boolean equals(Object element)	This method tests the two collections for similarity and returns true if they match.
16	public int hashCode()	This method gives the hash code value of the invoking collection.

Set represents an unordered group of members, and the members must be unique. So, a false is returned by the add() method when you try to add duplicate elements in it. You are allowed to have just one null value in a Set. There are no extra methods defined by Set; only the methods of collection interface are available for use with Set.

12.3.1 Classes that implement Set interface

The Set interface is implemented by HashSet and LinkedHashSet class. There is an interface named SortedSet that extends Set interface, and the class TreeSet implements the Sortedset interface.

HashSet class: Its purpose is to produce a collection and keep it in a hash table. Each element in the Set has a unique value of its own, called hash code which is utilized as an index at which element related to that hash code is present. Hashing ensures that the execution time of operations like add(), remove(), remain constant irrespective of the size of the HashSet. The members are inserted in HashSet irrespective of the order of insertion, and the order depends on the hashcode of elements. Constructors for HashSet class are given below: -

- HashSet( ) : Constructs a default HashSet which is empty.
- HashSet(Collection coll) : Initializes the HashSet with the members of collection coll.
- HashSet(int cap) : Initializes the HashSet with the size of the integer value of cap.
- HashSet(int cap, float fill_Ratio) : This constructor is used to initialize the data members - capacity and the fill ratio, of HashSet. The valid value of fill ratio has to be between 0.0 and 1.0, and it governs how full the HashSet is allowed to be before it is resized upwards. If the number of elements in HashSet is more than the capacity (cap) of the Hashset multiplied by its fill ratio, then the hash set is lengthened.

Corresponding to the first three constructors, the value of fill ratio is 0.75.

For instantiating a object of the HashSet class, we have to mention the data type of elements of set. For example:-

```
HashSet<datatype> hset = new HashSet<datatype>();
```

Here, datatype is the type of elements that can be stored in hset.

```
import java.util.*;
class HashSetProgram
{
  public static void main(String args[])
  {
    // create a HashSet
    HashSet<String> hset1 = new HashSet<String>();
    // add elements to the hash set
    hset1.add("Java");
    hset1.add("C++");
    hset1.add("Python");
    hset1.add("R");
    hset1.add("Matlab");
    hset1.add("Scala");
    System.out.println(hset1);
  }
}
```

Output:

```
[Java, C++, R, Scala, Python, Matlab]
```

We can see that the set elements are not printed as per the order of insertion of elements.

Traversal of Set using Iterator

If we want to pass over the elements of any collection, we can make use of Iterable interface, which is the most important interface of the collection framework and is at the root of the hierarchy. As we know that the Iterable interface is extended by collection interface; hence all the interfaces and classes of collection framework implement this interface. The most important feature of the Iterable interface is to deliver an iterator for the collections. Iterable interface has just one method which is abstract and whose name is iterator(). It returns an object of type Iterator.

Iterator iterator();

The methods declared by Iterator are described in Table 12.4.

Table 12.4 Description of methods of Iterator interface

S.No.	Method	Description
1.	boolean hasNext( )	Gives true only if there are more elements in the collection. Else, this method returns false.
2.	Object next( )	Givesthe next element. This method throws NoSuchElementException if there does not exist any next element.
3.	void remove( )	Eliminates the current element. It throws IllegalStateException if an attempt is made to invoke remove( ) method that is not preceded by a call to next( ) method.

In order to use an iterator to pass over the elements of a collection, you should follow the following steps.

1. Get an iterator to the start of the collection by invoking the iterator( ) method of the collection.
2. Write a while loop that invokes hasNext( ). The loop would iterate till the time hasNext( ) returns true.
3. Inside the loop, get and display each element by invoking next( ) method.

Same program is written here again by using iterator.

```
import java.util.*;
class HashSetwithIterator
{
  public static void main(String args[])
  {
    // create a HashSet
    HashSet<String> hset1 = new HashSet<String>();
    // add elements to the hash set
    hset1.add("Java");
    hset1.add("C++");
    hset1.add("Python");
    hset1.add("R");
    hset1.add("Matlab");
    hset1.add("Scala");
    // traverse the set using Iterator and print each element one by one
    Iterator<String> itr = hset1.iterator();
    while(itr.hasNext())
    {
      System.out.println(itr.next());
    }
  }
}
```

Output:

```
Java
C++
R
Scala
Python
Matlab
```

LinkedHashSet class

A LinkedHashSet is an ordered form of HashSet that keeps a doubly-linked list of the elements in the Set. The sequence of insertion is preserved in LinkedHashSet and while iterating through LinkedHashSet, the elements are returned in the order of insertion. This is an advantage over HashSet. The hash code is used for indexing in the same way as in HashSet.
Constructors for LinkedHashSet class are given below,

Given below are the list of constructors supported by the LinkedHashSet:

- LinkedHashSet(): Constructs a default LinkedHashSet which is empty.
- LinkedHashSet(Collection Coll): This constructor would Initialize the LinkedHashSet with the elements of collection Coll.
- LinkedHashSet(int cap): This constructor would initialize the HashSet with capacity of the integer value of cap parameter.
- LinkedHashSet(int capacity, float fillRatio): This particular constructor would initialize both the capacity and the fill ratio of the LinkedHashSet. The permissible value of fill ratio is between 0.0 and 1.0, and it governs how full the HashSet can be before it is resized upwards. If the number of elements in LinkedHashSet is greater than the capacity of the LinkedHashset multiplied by its fill ratio, then it is expanded.

For the first three constructor functions, the value of the fill ratio is 0.75.

All methods of Collection interface and Iterable interface are available to LinkedHashSet class.

In order to instantiate an object of LinkedHashSet class, we have to mention the data type of its elements. For example:-

```
LinkedHashSet<datatype> Lhset = new LinkedHashSet<datatype>();
```

Here, datatype is the type of elements that can be stored in Lhset.

Let's see a program based on LinkedHashSet class.

```
import java.util.*;
class LinkedHSet
{
  public static void main(String args[])
  {
    // create a LinkedHashSet
    LinkedHashSet<String> Lhset = new LinkedHashSet<String>();
    // add elements to the LinkedHashSet
    Lhset.add("Operating System");
    Lhset.add("Java Programming");
    Lhset.add("Data Structures");
    Lhset.add("Database Management System");
    Lhset.add("Computer Networks");
    Lhset.add("Computer Architecture");
    // traverse the set using Iterator and print each element one by one
    Iterator<String> itr = Lhset.iterator();
    System.out.println("Original contents of LinkedHashset");
    while(itr.hasNext())
    {
        System.out.println(itr.next());
    }
```

```
    //Remove an element from the LinkedHashSet
    Lhset.remove("Data Structures");
    // traverse the set again
    System.out.println("Elements of LinkedHashset after removing 'Data Structures' ");
    System.out.println(Lhset);
  }
}
```

Output:

```
Original contents of LinkedHashset
Operating System
Java Programming
Data Structures
Database Management System
Computer Networks
Computer Architecture
Elements of LinkedHashset after removing 'Data Structures'
[Operating System, Java Programming, Database Management System, Computer Networks,
Computer Architecture]
```

Students can notice here that the elements iterate in the order of their insertion.

12.3.2 Interface that extends Set interface

Sorted Set Interface implemented by TreeSet class

This interface extends Set interface and affirms that the elements of set are sorted in increasing order. It defines first() and last() methods to quickly return the first and last element respectively. This interface is implemented by **TreeSet class**. The important features of TreeSet class are given below: -

- It uses a tree for storage of the elements.
- Elements can be accessed and retrieved very fast.
- TreeSet class does not allow null element to be stored.
- As TreeSet class extends Sorted Set interface, automatically it maintains ascending order.
- Although elements are stored in increasing order, we can pass through in decreasing sequence also by using methodTreeSet.descendingIterator().

The constructors supported by the TreeSet class are given below: -

- TreeSet(): Constructs a default TreeSet which is empty.
- TreeSet(Collection C): Initializes the TreeSet with the elements of collection c.
- TreeSet(Comparator cmp): Creates an empty TreeSet that will be sorted as per the value of comparator specified by cmp..
- TreeSet(SortedSet sset): Creates a treeset that comprises of members of SortedSet sset.

You can instantiate an object of TreeSet class in one of the following ways.

```
TreeSet<datatype> tset = new TreeSet<datatype>();
```

Here, datatype is the type of elements that are allowed to be stored in tset.
Another way to instantiate an object of TreeSet is by using SortedSet interface, as given below: -

```
SortedSet<datatype> tset = new TreeSet<datatype>();
```

Let's see a program based on TreeSet class.

```
import java.util.*;
class TreeSetProg
{
  public static void main(String args[])
  {
    // create a TreeSet
    TreeSet<String> tset1 = new TreeSet<String>();
    // add elements to the TreeSet
    tset1.add("India");
    tset1.add("Australia");
    tset1.add("USA");
    tset1.add("Canada");
    tset1.add("Japan");
    // traverse the set using Iterator and print each element one by one
    Iterator<String> itr = tset1.iterator();
    System.out.println("Original contents of TreeSet, by default in ascending order");
    while(itr.hasNext())
    {
      System.out.println(itr.next());
    }
    //Print the last element
    System.out.print("The last element is ");
    System.out.println(tset1.last());
    // traverse the treeset again but now in descending order
    System.out.println("Contents of TreeSet in descending order are...");
    Iterator<String> desc_itr= tset1.descendingIterator();
    while(desc_itr.hasNext())
    {
      System.out.println(desc_itr.next());
    }
  }
}
```

Output :

```
Original contents of TreeSet, by default in ascending order
Australia
Canada
India
Japan
USA
The last element is USA
Contents of TreeSet in descending order are...
USA
Japan
India
Canada
Australia
```

☞Check Your Progress-2

1) What do you understand by Iterator in the Java Collection Framework?

……

……

……

2) What is the HashSet class in Java, and how does it store elements?

……

……

……

……

3) Differentiate between TreeSet and HashSet.

……

……

……

……

4) What is the initial fill ratio of HashSet?

a) 1.5
b) 0.5
c) 0.75
d) 1.0

5) What is the output of the following program

```
import java.util.*;
class test1
{
  public static void main(String[] args)
  {
    HashSet<String> set=new HashSet<String>();
    set.add(null);
    set.add("One");
    for (String s: set)
    System.out.println(s);
  }
}
```

12.4 MAP INTERFACE

Map interface is a part of java.util package but map is not a collection as it does not implement the collection interface. Map interface represents a mapping between a key and a value. A map contains unique keys though the values for two or more keys can be same. Each key can map to at the most one value only. A Map is suitable whenever we want to search, update or delete items on the basis of a key. An example of map is zip code (key) and city (value). The hierarchy of classes and interfaces of Map interface is shown in Figure 12.2. The methods to perform various operations on maps are given in Table 12.5.

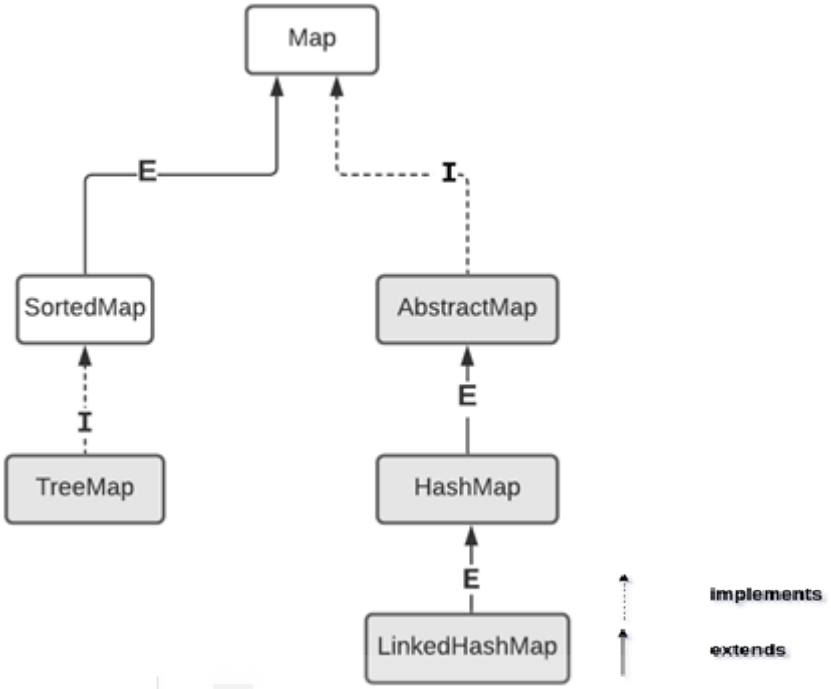


Figure 12.2: Map interface of Collections Framework

Table 12.5: Description of methods of Map interface

S.No.	Method	Description
1	void clear()	Deletes all key/value pairs from the invoking map.
2	boolean containsKey(Object k)	Gives true if the map object that invoked this method has k as one of the keys, else it gives false.
3	Boolean containsValue(Object obj)	It gives true if the map object that invoked this method has obj as one of the values, else its gives false.
4	Set entrySet( )	This method gives a Set that comprises of all the elements of the map. The set has in it all objects of type Map.Entry. We can use this method to provide a Set-view of the invoking map.
5	boolean equals(Object ob)	It gives true if ob is a Map and has the same entries as invoking Map. Else, it gives false.
6	Object get(Object obj)	It gives the value corresponding to the key obj.
7	int hashCode( )	It gives the hash code of the map that invoked this method.
8	boolean isEmpty( )	If the object that invoked this method does not have any element in it, then it gives true, else false.

9	Set keySet( )	This method produces a Set that has all the keys of the invoking map. It is used to provide a set-view of only the keys present in the invoking map.
10	Object put(Object k, Object v)	It adds an element in the invoking map. If there is already an element in the map having key same as k, then the value of that key is overwritten. Here, k corresponds to the key and v corresponds to the value. This method gives null if the key was not present already in the map. If a key was present, then the previous value linked to that key is returned.
11	void putAll(Map mp)	Adds all the elements of map mp into the invoking map.
12	Object remove(Object k)	Deletes the entry of map whose key is equal to k.
13	int size( )	It gives the number of key/value pairs that are present in the invoking map.
14	Collection values( )	This method gives a collection-view of only the values of the map.

12.5 HASHMAP CLASS

HashMap class implements the Map interface. A hash table is used by HashMap in order to implement Map interface. The advantage of hash table is that the time for basic operations like get() and put() remains constant irrespective of size of Map. Given below are the constructors for instantiating an object of HashMap.

- HashMap( ) : It instantiates a default hash map.
- HashMap(Map mp) : It initializes the hash map with the elements of m.
- HashMap(int cap) : It constructs a hash map with a capacity equal to argument cap.
- HashMap(int cap, float fillR): This constructor uses the arguments cap and fillR to initialize both the capacity and fill ratio. The gist of capacity and fill ratio remains the same as for HashSet, discussed earlier.

You should remember that there is no guarantee of the order of elements in the case of HashMap. So, the order in which elements are inserted into a hash map may not match the order of their retrieval by the iterator.

You can instantiate an object of HashMap class by using one of the given below techniques.

1. Non-Generic way (Old Style)

   Map mp=new HashMap(); or HashMap mp=new HashMap();

Anything involving HashMap or Map without a datatype argument (the angle brackets < and > and the part between them) is a raw type that is an older approach and shouldn't be used. A raw type is not generic and lets you do hazardous things.

2. Generic way (New Style)

Map<T1,T2> mp1=new HashMap<T1,T2>();

T1, T2 − The generic type parameter passed to the generic interface Map and its implementation class HashMap, during map declaration.

Generics declaration allows compiler to check the 'mp1' usages during compile-time and is very safe.

We will be using the Generic way in the example given below: -

```
import java.util.*;
class HashMapEx
{
  public static void main(String args[])
    {
      Map<Integer,String> map1=new HashMap<Integer,String>();
      map1.put(1,"Anita");
      map1.put(2,"Charu");
      map1.put(3,"Geeta");
      System.out.println();
      System.out.print("Map Elements :-" +map1);
      System.out.println();
      //For traversing Map, convert it into Set
      Set set=map1.entrySet();//Converting to Set so that we can traverse
      Iterator itr1=set.iterator();
      System.out.println("Map elements by traversing the map after being converted to
set");
      while(itr1.hasNext())
      {
        //Converting to Map.Entry so that we can get key and value separately
        Map.Entry entry=(Map.Entry)itr1.next();
        System.out.println(entry.getKey()+" "+entry.getValue());
      }
      System.out.println("Number of elements in map is "+map1.size()+"
elements.");
      boolean b=map1.containsKey(1);
      System.out.println("It is "+b+" that the map conatins an element with key 1");
      //Remove the entry whose key is 2
      map1.remove(2);
      System.out.println("Contents of map after deleting element with key 2 are
"+map1);
      if (map1.isEmpty())
      System.out.println("Map is empty");
      else
      System.out.println("Map is not empty");
      //Delete all elements of map
      map1.clear();
      System.out.println("All elements cleared");
```

```
        if (map1.isEmpty())
        System.out.println("Now Map is empty");
        else
        System.out.println("Now Map is not empty");
    }
}
```

Output:

```
Map Elements :-{1=Anita, 2=Charu, 3=Geeta}
Map elements by traversing the map after being converted to set
1 Anita
2 Charu
3 Geeta
Number of elements in map is 3 elements.
It is true that the map conatins an element with key 1
Contents of map after deleting element with key 2 are {1=Anita, 3=Geeta}
Map is not empty
All elements cleared
Now Map is empty
```

Given below is a program that uses HashMap for storing employee ids with their names. This program is menu-driven and takes input from the user, and accordingly uses an appropriate method to display the result.

```
import java.util.*;
class HashMapEx1
{
   public static void main(String args[])
   {
     Scanner sc = new Scanner(System.in);
     Map<Integer,String> map1=new HashMap<Integer,String>();
     map1.put(401,"Ramesh");
     map1.put(205,"Balram");
     map1.put(156,"Vinod");
     map1.put(555,"Geeta");
     System.out.println();
     int choice=1;
     do
     {
       System.out.println("Main Menu");
       System.out.println("1: Size of Map");
       System.out.println("2: Remove all elements of Map");
       System.out.println("3: Check if hashmap is empty?");
       System.out.println("4: Search for an employee");
       System.out.println("5: Input Employee ID and display name of that employee");
       System.out.println("6: Display names of all employees");
       System.out.println("7: Display employee id with their names");
       System.out.println("8: Exit");
       System.out.println("Enter choice: ");
       choice = sc.nextInt();
       switch (choice)
       {
         case 1:  System.out.println("Size of the hash map: "+map1.size()+" records");
                  break;
         case 2:  map1.clear();
                  System.out.println("The New map: " + map1);
                  break;
         case 3:  boolean result = map1.isEmpty();
```

```
                System.out.println("It is " + result + " that hash map is empty");
                break;
            case 4: System.out.print("Enter employee id to be searched.. ");
                int eid=sc.nextInt();
                if (map1.containsKey(eid))
                        System.out.println("yes! " + eid + " is present.");
                else
                        System.out.println("no! " + eid + " is not there.");
                break;
            case 5: System.out.print("Enter employee id whose name is required.. ");
                eid=sc.nextInt();
                if (map1.containsKey(eid))
                {
                        System.out.println("yes! " + eid + " is present.");
                        String val=(String)map1.get(eid);
                        System.out.println("Name for " + eid + " is " + val);
                        }
                else
                System.out.println("no! " + eid + " is not there.");
                break;
            case 6: System.out.println("Names of all employees: "+ map1.values());
                break;
            case 7: Set set = map1.entrySet();
                System.out.println("Employee ID with names: " + set);
                break;
            case 8: System.out.println("Byeeee....");
                break;
            default: System.out.println("Invalid choice");
                break;
          }
        }
        while(choice!=8);
      }
}
```

Output:

```
Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee
6: Display names of all employees
7: Display employee id with their names
8: Exit
Enter choice:
1
Size of the hash map: 4 records
Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee
6: Display names of all employees
7: Display employee id with their names
8: Exit
```

```
Enter choice:
3
It is false that hash map is empty
Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee
6: Display names of all employees
7: Display employee id with their names
8: Exit
Enter choice:
5
Enter employee id whose name is required.. 6
no! 6 is not there.
Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee

Enter choice:
6
Names of all employees: [Ramesh, Geeta, Vinod, Balram]
Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee
6: Display names of all employees
7: Display employee id with their names
8: Exit
Enter choice:
8
Byeeee....
```

☞Check Your Progress-3

1) Differentiate between Set and Map.

   ……

   ……

   ……

2) Why do we use Map interface? What are the main classes implementing Map interface?

   ……

   ……

   ……

3) What is the difference between HashSet and HashMap?

   ……

   ……

   ……

4) Is hashmap an ordered collection.

   a) True

   b) False

5) Write down the output produced by the following program

```
import java.util.*;
public class test2
{
  public static void main(String[] args)
  {
      HashMap<String, Integer> map = new HashMap<>();
      map.put("Ram", 20);
      map.put("Shyam", 25);
      map.put("Harish", 22);
      System.out.println("Size of map is:- "+ map.size());
      System.out.println(map);
      if (map.containsKey("Shyam"))
      {
        Integer a = map.get("Shyam");
        System.out.println("value for key"+ " \"Shyam\" is:- "+ a);
      }
  }
}
```

12.6 LIST INTERFACE

List interface extends the Collection interface, and hence it comprises all the methods available in Collection interface. It is used to manage the ordered collection of objects. Duplicate data is also allowed in list. Elements can be accessed by referring to their position in the list, with zero-based index. Various methods defined by list interface are mentioned in Table 12.6. The list interface is implemented by Vector, ArrayList and LinkedList classes. The object of list can be instantiated by any of the following approaches: -

List <T> aList = new ArrayList<> ();
List <T> lList = new LinkedList<> ();
List <T> v = new Vector<> ();
Where T is a generic type parameter passed to the generic interface List.

Table 12.6: Description of methods of List Interface

Sr.No.	Method	Description
1	void add (int idx, Object ob)	Adds ob into the invoking list at the index mentioned in the idx. None of the elements is lost by overwriting because any pre-existing element at or after the position of insertion are shifted up.
2	boolean addAll (int idx, Collection col)	Adds all elements of col in the list at the index passed in the idx. None of the elements is lost by overwriting because any pre-existing element at or after the point of insertion are moved up. This method gives true if the invoking list modifies otherwise, it gives false.
3	Object get(int idx)	It gives the object available at the mentioned idx of the invoking list.

4	int indexOf(Object ob)	It gives the index of the first occurrence of ob in the invoking list. If ob is not present in the list, it returns -1.
5	int lastIndexOf(Object ob)	It gives the index of the last occurrence of ob in the invoking list. If ob is not found in the list, then it returns -1
6	ListIterator Iterator( )	It gives an iterator to the beginning of the invoking list.
7	ListIterator Iterator(int index)	It gives an iterator that begins at the specified index of the invoking list.
8	Object remove(int idx)	It deletes the element at position idx from the invoking list and gives that deleted element. This makes the resulting list compact, and hence the total number of elements is decreased by one.
9	Object set (int idx, Object ob)	Allocates ob to the location specified by idx within the list that invoked this method.
10	List subList (int begin, int end)	It gives a list that includes elements from begin to (end-1) from the invoking list.

12.7 VECTOR CLASS

The Vector class extends AbstractList class. The List interface is implemented by this class. Vector implements a dynamic array that can grow or shrink as per the requirement. The elements of vector can be retrieved with an integer index. Vector is quite alike ArrayList, but it is synchronized i.e. it is thread-safe. The Vector class should be used in the thread-safe implementation only, and if you don't require synchronous access then ArrayList should be used as it will perform better.

Given below are the Constructors for Vector class:
Vector(): This constructs a default vector with the original capacity to store ten elements.
Vector(int s): A vector is constructed that has initial capacity equal to argument s.
Vector(int s, int increm): This constructor initializes both the size and increment of vector by using its arguments. The meaning of increm is the number of elements to assign each time when a vector is resized upward. (If the argument increm is not mentioned then vector's capacity will be doubled in each allocation round).
Vector(Collection col): This particular constructor initializes the vector with the elements of collection col.
Vector defines the following data members that have protected visibility mode:
int capacityIncrement: Stores the value of the increment.
int elementCount: Contains the count of elements presently in vector.
Object elementData[]: An array that stores the elements of the vector.

Vector class has all the methods inherited from its super classes, and it also defines additional methods explained in Table 12.7.

Table 12.7: Description of methods of Vector class

S.No.	Method	Description
1	void add (int idx, Object obj)	Adds the specified object at the mentioned idx in the Vector. This method is not synchronous.
2	boolean add (Object obj)	Returns true after appending the mentioned object to the end of the Vector.
3	boolean addAll(Collection col)	Appends all elements of the mentioned Collection to the end of the Vector, in the same sequence as they are returned by the iterator of the mentioned Collection. Returns true after appending.
4	boolean addAll (int idx, Collection c)	Inserts each and every element of the mentioned Collection into the Vector at the mentioned index. Returns true after inserting element.
5	void addElement (Object ob)	Adds the mentioned object to the end of the invoking vector, and increases its size by one. addElement() is synchronized.
6	int capacity()	The current capacity of the invoking vector is returned.
7	void clear()	All elements from the vector are removed.
8	Object clone()	A clone of the vector is returned.
9	boolean contains(Object obj)	Returns true if the mentioned object is a member in the invoking vector.
10	boolean containsAll(Collection c)	If the invoking vector contains all the elements of the mentioned Collection, then this method returns true.
11	void copyInto (Object[] anArray)	All the components of the invoking vector are copied into the mentioned array.
12	Object elementAt(int idx)	The component at the mentioned index is returned.
13	Enumeration elements()	An enumeration of all components of the invoking vector is returned.
14	void ensureCapacity (int minCapacity)	This method increases the capacity of the invoking vector, if required, so that it can store no less than the number of components mentioned by minCapacity.

15	boolean equals (Object ob)	It the invoking vector is equal to Object, then true is returned.
16	Object firstElement()	It gives the first element (the object at index 0) of the invoking vector.
17	Object get(int idx)	It gives the element at the mentioned index of the vector.
18	int hashCode()	The hash code for the invoking vector is returned.
19	int indexOf(Object obj)	The position of first occurrence of passed object is searched and returned
20	int indexOf (Object obj, int idx)	The first occurrence of passed object is searched but the search begins at idx.
21	void insertElementAt (Object ob, int idx)	The passed object is inserted (as a component) in the vector at the mentioned index.
22	boolean isEmpty()	Returns true if the vector does not have any components.
23	Object lastElement()	The last element of the invoking vector is returned.
24	int lastIndexOf(Object obj)	The index of the last occurrence of object obj, in the vector is returned.
25	int lastIndexOf (Object obj, int index)	Searches towards the back for passed object, beginning at the mentioned index, and gives the position of it.
26	Object remove(int idx)	Deletes and returns the element at the mentioned index of the vector.
27	boolean remove (Object obj)	Deletes the first occurrence of the mentioned object in the invoking vector and returns true. If the element is not found in the vector, it returns false.
28	boolean removeAll (Collection c)	Eliminates all elements of the specified collection from the vector and returns true. The method returns false, if the collection is not found.
29	void removeAllElements()	All elements of the vector are removed and its size is set to zero.
30	boolean removeElement (Object obj)	The first occurrence of the passed object is removed from the vector.
31	void removeElementAt (int idx)	The element at the mentioned index is removed from the vector.
32	protected void removeRange (int	All elements whose index lies between fmIndex, inclusive and toIndex, exclusive, are

	fmIndex, int toIndex)	removed from the vector.
33	boolean retainAll(Collection c)	Only the elements that are contained in the mentioned Collection are retained in the invoking vector.
34	Object set(int index, Object element)	Element at the specified index in the invoking vector is replaced with the mentioned element.
35	void setElementAt(Object obj, int idx)	The object at the specified index (idx) of the vector is set to the object obj.
36	void setSize(int newSize)	The size of the vector is set to the newSize.
37	int size()	It gives the number of elements in the invoking vector.
38	List subList (int fromIndex, int toIndex)	A view of the slice of the List between fromIndex, inclusive, and toIndex, exclusive is returned.
39	Object[] toArray()	An array containing all the components of the vector in the exact order is returned.
40	Object[] toArray(Object[] a)	An array comprising all of the elements in this vector in the correct order is returned. The runtime type of the returned array is same as that of the specified array.
41	String toString()	A string representation of the vector is returned.
42	void trimToSize()	The capacity of the vector is trimmed to its current size.

You can instantiate an object of HashMap class in one of the following ways.

1. Non-Generic way (Old Style)
   Vector v=new Vector(); or List v=new Vector();
   Anything involving HashMap or Map without a datatype argument (the angle brackets < and > and the part between them) is a raw type that is an older approach and shouldn't be used. A raw type is not generic and lets you do hazardous things.

2. Generic way (New Style)
   Vector<T> v=new Vector<T>();
   T − The generic type parameter passed to the generic interface List and its implementation class Vector.
   Generic declaration allows compiler to check the 'v' usages during compile-time and it is very safe.
   We will be using the Generic way in the example given below: -

We will be using the Generic way in the example given below: -

```
import java.util.*;
public class VectorEx
```

```
  public static void main(String args[])
  {
    //Create an empty vector
    Vector<String> vec1 = new Vector<String>();
    //Add elements to the vector
    vec1.add("Television");
    vec1.add("Refrigerator");
    vec1.add("Washing Machine");
    //display size and capacity
    System.out.println("Size is: "+vec1.size());
    System.out.println("Default capacity is: "+vec1.capacity());
    //Display Vector elements
    System.out.println("Vector element is: "+vec1);
    //Add two more elements
    vec1.add("Microwave Oven");
    vec1.add("Mobile Phone");
    //Again display size and capacity after two insertions
    System.out.println("Size after addition: "+vec1.size());
    System.out.println("Capacity after addition is: "+vec1.capacity());
    //Display Vector elements again
    System.out.println("Elements are: "+vec1);
    //Checking if Television is present or not in this vector
    if(vec1.contains("Television"))
    {
      System.out.println("Television is present at the index "
     +vec1.indexOf("Television"));
    }
    else
    {
      System.out.println("Television is not present in the list.");
    }
    //Print the first element
    System.out.println("The first element of the vector is = "+vec1.firstElement());
    //Print the last element
    System.out.println("The last element of the vector is = "+vec1.lastElement());
    //Add Refrigerator again
    vec1.add("Refrigerator");
    //Display Vector elements again
    System.out.println("Elements are: "+vec1);
    //use remove() method to delete the first occurence of an element
    System.out.println("Remove first occurrence of element Refrigerator:
    "+vec1.remove("Refrigerator"));
    //Display the vector elements after remove() method
    System.out.println("Values in vector: " +vec1);
    //Remove the element at index 4
    System.out.println("Remove element at index 3: " +vec1.remove(3));
    System.out.println("New Values in vector: " +vec1);
    //Get the hashcode for this vector
    System.out.println("Hash code of this vector = "+vec1.hashCode());
    //Get the element at specified index
    System.out.println("Element at index 1 is = "+vec1.get(1));
  }
```

}

Output:

```
Size is: 3
Default capacity is: 10
Vector element is: [Television, Refrigerator, Washing Machine]
Size after addition: 5
Capacity after addition is: 10
Elements are: [Television, Refrigerator, Washing Machine, Microwave Oven, Mobile Phone]
Television is present at the index 0
The first element of the vector is = Television
The last element of the vector is = Mobile Phone
Elements are: [Television, Refrigerator, Washing Machine, Microwave Oven, Mobile Phone, Refrigerator]

Remove first occurrence of element Refrigerator: true
Values in vector: [Television, Washing Machine, Microwave Oven, Mobile Phone, Refrigerator]
Remove element at index 3: Mobile Phone
New Values in vector: [Television, Washing Machine, Microwave Oven, Refrigerator]
Hash code of this vector = 1730521510
Element at index 1 is = Washing Machine
```

12.8 STACK CLASS

The stack is a linear data structure that extends the Vector class. It is based on Last-In-First-Out (LIFO). Only one constructor is defined by the Stack class, which creates an empty stack.

Stack<T> stk = new Stack<T>();

T − The generic type parameter passed to the Stack.

All methods defined by Vector class are available in Stack in addition to few of its own methods. These added methods are given in Table 12.8.

Table 12.8: Description of methods of Stack class

S.No.	Method	Description
1	boolean empty( )	If stack is empty, true is returned otherwise false is returned.
2	Object peek( )	The element at the top of the stack is returned without removing it from stack.
3	Object pop( )	The element at the top of the stack is returned as well as removed from stack.
4	Objectpush (Object element)	The element is inserted onto the stack.
5	int search (Object element)	The mentioned element is searched in the stack. If found, its offset from the top of the stack is returned otherwise, -1 is returned.

Given program demonstrates some of the operations on stack.

import java.util.Stack;

```
import java.util.Scanner;
public class StackExample
{
   public static void main(String[] args)
   {
      int j;
      Scanner in = new Scanner(System.in);
      //creating an instance of Stack class
      Stack<Integer> stk= new Stack<>();
      // checking stack is empty or not
      boolean result = stk.empty();
      System.out.println("It is "+result+" stack is empty");
      // pushing five elements into stack
      for (int i=1;i<6;i++)
      {
         System.out.println("Enter element number "+i+" to be pushed");
         j = in.nextInt();
         stk.push(j);
      }
      System.out.println("Elements in Stack: " + stk);
      System.out.println("Enter the element to be searched...");
      j = in.nextInt();
      Integer pos = (Integer) stk.search(j);
      if(pos == -1)
      System.out.println("Element not found");
      else
      System.out.println("Element is found at position: " + pos);
   }
}
```

Output:

```
It is true stack is empty
Enter element number 1 to be pushed
56
Enter element number 2 to be pushed
43
Enter element number 3 to be pushed
98
Enter element number 4 to be pushed
73
Enter element number 5 to be pushed
122
Elements in Stack: [56, 43, 98, 73, 122]
Enter the element to be searched...
73
Element is found at position: 2
```

☞Check Your Progress-4

1) Write down the differences and similarities between List and Set.

……

……

……

……

2) Why do we use the List interface? Name the classes that implement the List

interface.

……

……

……

3) What is the output of the following program?

```
import java.util.*;
public class test3
{
public static void main(String[] args)
  {
    List<Integer> list1 = new ArrayList<Integer>();
    list1.add(0, 1);
    list1.add(1, 2);
    System.out.println(list1);
    List<Integer> list2 = new ArrayList<Integer>();
    list2.add(1);
    list2.add(2);
    list2.add(3);
    list1.addAll(1, list2);
    System.out.println(list1);
    list1.remove(1);
    System.out.println(list1);
    System.out.println(list1.get(3));
    list1.set(0, 5);
    System.out.println(list1);
  }
}
```

4) Explain the Vector class and its importance.

……

……

……

5) What do you mean by the Stack class, and what are the various methods provided by it?

……

……

……

12.9 SUMMARY

This unit explains various classes and interfaces that are part of java.util package. We have deliberated the functions of date and time classes along with suitable program codes. This unit deals with List and Set that are important interfaces of collection framework. Different classes that implement these interfaces have been elaborated with various programs. We have discussed Map interface along with HashMap class that implements it. Overall, the facilities offered by Collection framework have been explained in detail.

12.10 SOLUTIONS/ANSWERS

☞Check Your Progress-1

1) Collection Framework is a group of classes and interfaces to store and manipulate the group of objects. Various operations like searching, sorting, insertion and deletion can be performed on these objects. Various classes such as ArrayList, Vector, Stack, and HashSet, etc. are provided by the collection framework. Also, there are many interfaces such as List, Queue, Set, etc. available in collection framework.

2) Maps is not a part of the collection framework.

3) In order to check whether two date objects are equal, the following method is used.

   int compareTo (Date date) : The value of the invoking object is compared with that of date object. If both are equal, the function returns 0. A negative value is returned, if the invoking object is earlier than date. Or a positive value is returned, if the invoking object is later than the date.

☞Check Your Progress-2

1) In order to pass over the elements of any collection, we can make use of Iterable interface, which is the root interface of the whole collection framework. The Iterable interface is extended by the collection interface. Hence, all the interfaces and classes of collection framework implement this interface. The most important feature of Iterable interface is to offer an iterator for the collections. Only one method, which is abstract and whose name is iterator(), is contained in this interface. It returns an object of type Iterator.

   Iterator iterator();

2) java.util.HashSet class is a member of the Java collections framework, which implements the Set interface. Its purpose is to create a collection and store it in a hash table. Each element in Set has a unique value of its own, called hash code which is used as an index at which element associated with that hash code is stored. Hash code is obtained by converting the informational content into a unique value (known as hash code). Hashing ensures that the execution time of operations like add(), remove(), remain constant even for set of bigger size. The elements are inserted in HashSet irrespective of the order of insertion, and the order depends on the hash code of elements.

3) The HashSet and TreeSet, both classes, implement Set interface. The differences between the both are listed below:

HashSet	TreeSet
It does not maintain any sequence of elements added to it.	It maintains ascending order of elements.
It is implemented by hash table.	It is implemented by a Tree structure.
Performance is better than TreeSet.	TreeSet performance is not as good as HashSet.
HashSet is backed by HashMap.	TreeSet is backed by TreeMap.

4) 0.75
5) null
   One

☞Check Your Progress-3

1) The differences between Set and Map are given below:

- Set contains only the values, whereas Map contains key and values both.
- Set has unique values, whereas Map can contain unique Keys with duplicate values (Keys are unique but two or more keys can have the same value.)
- Set can include just one null, whereas Map can include a single null key with n number of null values.

2) Map interface is a part of java.util package, but map is not a collection as it does not implement the collection interface. Map interface represents a mapping between a key and a value. A map contains unique keys though the values for two or more keys can be the same. At the most, one value can be mapped by each key. A Map is suitable if we have to search, update or delete elements on the basis of a key.The classes that implement Map interface are: HashMap, Hashtable, LinkedHashMap.

3) The differences between the HashSet and HashMap are listed below.

HashSet	HashMap
Set interface is implemented by Hashset.	Map interface is implemented by the HashMap.
It contains only values.	It contains a mapping between a key and its value.
It can be traversed.	HashMap needs to be converted into Set to be iterated.
It cannot have any duplicate value.	It can have duplicate values with unique keys.
It can hold just one null value.	It can contain a single null key with n number of null values.

4) False. Hashmap outputs in the order of hashcode of the keys. Though actually it is unordered but will always have the same result for the same set of keys.

5)

```
Size of map is:- 3
{Shyam=25, Harish=22, Ram=20}
value for key "Shyam" is:- 25
```

☞Check Your Progress-4

1) The List and Set both extend the collection interface. However, there are some differences between the both, as mentioned below :

List	Set
Duplicate elements are allowed in List.	Only unique elements are allowed in Set.
It is an ordered collection that maintains the order of insertion of elements.	It is an unordered collection that does not preserve the order of insertion.

The List interface can allow any number of null values.	The Set interface allows just a single null value.

2) The List interface is an ordered collection of elements. List preserves the insertion order and allows duplicate values to be stored. This interface comprises different methods that enable easy manipulation of elements based on the element index. The main classes that implement List interface are ArrayList, LinkedList, Stack, and Vector.

3)

```
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
[5, 2, 3, 2]
```

3) An index is used to access the elements of a vector. Vector is just like a dynamic array. There is no specific size of a vector; it can shrink or grow automatically whenever required. It is quite similar to ArrayList, but there are two differences as given below:

- Vector is synchronized.
- Many legacy methods are available in Vector class that are not part of the collections framework.

5) Last-in-first-out is the basic principle of Stack class. The elements are added as well as removed from the rear end (that is where the latest item was inserted or removed from). The action of adding an element to a stack is called push, while removing an element is referred to as pop. In order to retrieve or fetch the top element of the Stack, we can use peek() method. The element retrieved is only accessed, not deleted or removed from the Stack.

12.11 REFERENCE/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" ,Vol.2, Pearson Education, 2013.
- E Balagurusamy , "Programming with Java:, McGraw-Hill Education,2019.
- Benjamin J. Evans, David Flanagan , "Java in a Nutshell: A Desktop Quick Reference", O'Reilly,2019.
- https://www.javatpoint.com/collections-in-java
- https://www.tutorialspoint.com/java/java_collections.htm
- https://www.geeksforgeeks.org/collections-in-java-2/
- https://www.programiz.com/java-programming/collections
- https://en.wikipedia.org/wiki/Java_collections_framework
- https://dzone.com/articles/an-introduction-to-the-java-collections-framework