UNIT-4

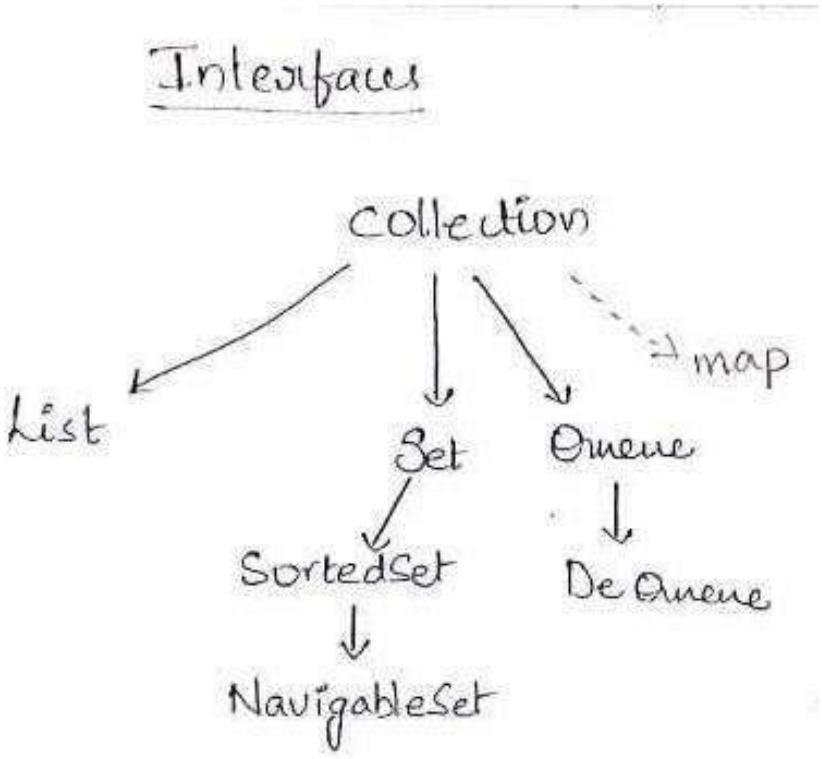
Collections Overview

The Java collection's framework provides a large set of readily usable general-purpose data structures and algorithms.

Collection framework standardizes the way in which group of objects are handled by our program.

These data structures and algorithms can be used with any suitable data type in a type-safe manner; this is achieved through the use of a language feature known as generics.

Collection Interfaces



Collection interface**:**

Common base interface for all the interfaces(classes) in the collection hierarchy.

Consists of methods that are common for all types of containers (List's, Set's and Map's)

List interface

List is for containers that store a sequence of elements

You can insert the elements using an index, and retrieve the same element later (so that it maintains the insertion order).

You can store duplicate elements in a List.

Set interface**:**

List for containers that do not allow duplicate elements.

SortedSet interface:

It maintains the set elements in a sorted order.

NavigableSet interface:

It allows searching the set for the closest matches

Queue interface:

It is the base interface for containers that holds a sequence of elements for processing.

For example, the classes implementing Queue can be LIFO (last in, first out—as in stack data structure) or FIFO (first in, first out—as in queue data structure).

Deque interface**:**

you can insert or remove elements from both the ends.

Map interface**:**

It is for containers that map keys to values.

SortedMap interface - the keys are in a sorted order.

NavigableMap interface - allows you to search and return the closest match for given search criteria.

Note : Map hierarchy does not extend the Collection interface.

Iterable interface - A class implementing this interface can be used for iterating with a for each statement.

Iterator interface - You can traverse over the container in the forward direction if a class implements the Iterator interface.

ListIterator interface - You can traverse in both forward and reverse directions if a class implements the ListIterator interface.

Important Methods - Collection interface

add(): Adds elem into the underlying container.

clear(): Removes all elements from the container.

isEmpty(): Checks whether the container has any elements or not.

iterator(): Returns an Iterator<Element> object for iterating over the container.

remove(): Removes the element if obj is present in the container.

size(): Returns the number of elements in the container.

toArray(): Returns an array that has all elements in the container.

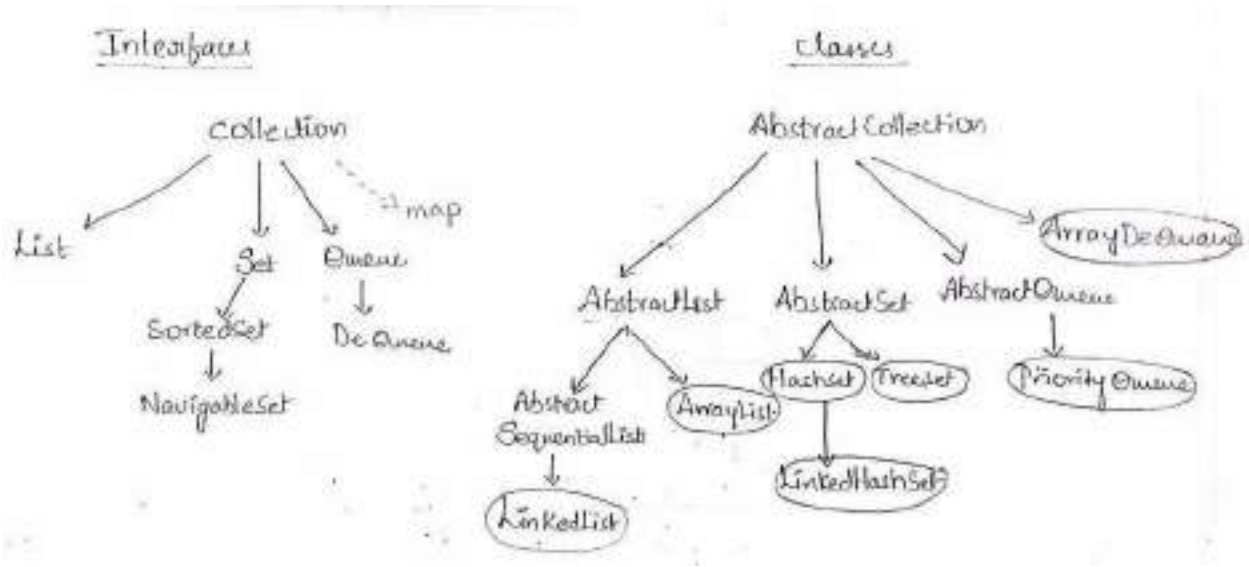
addAll(Collection c2): Adds all the elements in c2 into the underlying container.

containsAll(Collection c2): Checks if all elements given in c2 are present in the underlying container.

removeAll(Collection c2): Removes all elements from the underlying container that are also present in c2.

retainAll(Collection c2): Retains elements in the underlying container only if they are also present in c2; it removes all other elements.

The Collection classes



Array List

Java ArrayList class uses a dynamic array for storing the elements.

It is like an array, but there is no size limit.

We can add or remove elements anytime. So, it is much more flexible than the traditional array.

It is in the java.util package.

The ArrayList in Java can have the duplicate elements also.

It implements the List interface so we can use all the methods of List interface here.

The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

The important points about Java ArrayList class are:

Java ArrayList class can contain duplicate elements.

Java ArrayList class maintains insertion order.

Java ArrayList class is non synchronized.

Java ArrayList allows random access because array works at the index basis.

In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Example program:

```
import java.util.*;
class Demo1
{
  public static void main(String args[])
  {
    ArrayList<String>    al = new   ArrayList<String>();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    al.add("F");
    System.out.println(al);
    System.out.println(al.size());
    String   s=al.get(0);
    System.out.println("The First Element is: "+s);
    al.add(3,"Z");
    System.out.println(al);
    al.remove("E");
    System.out.println(al);
    al.remove(2);
    System.out.println(al);
  }
```

}

LinkedList

Java LinkedList class uses a doubly linked list to store the elements.

It provides a linked-list data structure.

It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

Java LinkedList class can contain duplicate elements.

Java LinkedList class maintains insertion order.

Java LinkedList class is non synchronized.

In Java LinkedList class, manipulation is fast because no shifting needs to occur.

Java LinkedList class can be used as a list, stack or queue.

Example program:

```
import java.util.*;
class Demo3
{
  public static void main(String args[])
 {
    LinkedList<String>   ll = new   LinkedList<String>();
    ll.add("C");
    ll.add("D");
    ll.add("E");
    System.out.println(ll);
    ll.addFirst("B");
    ll.addLast("F");
```

```
    System.out.println(ll);
   System.out.println(ll.peekFirst());
   System.out.println(ll.peekLast());
   System.out.println(ll);
   ll.pollFirst();
   ll.pollLast();
   System.out.println(ll);
  }
}
```

Hash Set

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

HashSet stores the elements by using a mechanism called hashing.

HashSet contains unique elements only.

HashSet allows null value.

HashSet class is non synchronized.

HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.

HashSet is the best approach for search operations.

The initial default capacity of HashSet is 16.

Example:

```
import  java.util.*;
class  Demo4
{
```

```
  public static void main(String args[])
  {
     HashSet<String>    hs = new   HashSet<String>();
     hs.add("Beta");
     hs.add("Alpha");
     hs.add("Eta");
     hs.add("Gamma");
     hs.add("Epsilon");
     hs.add("Omega");
     System.out.println(hs);
  }
}
```

Tree Set

Java TreeSet class implements the Set interface that uses a tree for storage.

It inherits AbstractSet class and implements the NavigableSet interface.

The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

Java TreeSet class contains unique elements only like HashSet.

Java TreeSet class access and retrieval times are quiet fast.

Java TreeSet class doesn't allow null element.

Java TreeSet class is non synchronized.

Java TreeSet class maintains ascending order.

Example program:

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

```
class Demo6
{
public static void main(String args[])
 {
   TreeSet<String>    ts = new   TreeSet<String>();
   ts.add("Beta");
   ts.add("Alpha");
   ts.add("Eta");
   ts.add("Gamma");
   ts.add("Epsilon");
   ts.add("Omega");
   System.out.println(ts);
  }
}
```

Priority Queue

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner.

It inherits AbstractQueue class.

Data structure : heap data structure.

A PriorityQueue is for retrieving elements based on priority.

Irrespective of the order in which you insert, when you remove the elements, the highest priority element will be retrieved first.

Example:

```
import java.util.*;
class  Demo10
```

```
{
  public static void main(String args[])
 {
    PriorityQueue<String>    p=new   PriorityQueue<String>();
    p.add("Amit");
    p.add("Vijay");
    p.add("Karan");
    p.add("Jai");
    p.add("Rahul");
   for(int i=1;i<=5;i++)
     {
       String  s=p.poll();
       System.out.println(s);
      }
   }
}
```

ArrayDeque

The ArrayDeque class provides the facility of using deque and resizable-array.

It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

Unlike Queue, we can add or remove elements from both sides.

Null elements are not allowed in the ArrayDeque.

ArrayDeque is not thread safe, in the absence of external synchronization.

ArrayDeque has no capacity restrictions.

ArrayDeque is faster than LinkedList and Stack.

Example Program:

```
import java.util.*;
class Demo7
{
  public static void main(String args[])
  {
     ArrayDeque<Integer>   adq = new   ArrayDeque<Integer>();
     adq.add(15);
     adq.add(20);
     adq.addLast(30);
     adq.addLast(40);
     adq.addLast(50);
     adq.addFirst(10);
     System.out.println(adq);
     adq.pollLast();
     System.out.println(adq);
  }
}
```

Accessing a Collection via an Iterator/ Using an Iterator

Iterator interface help us to retrieve objects in the collection one by one.

An iterator is an interface that is used in place of Enumerations in the Java Collection Framework. Moreover, an iterator differs from the enumerations in two ways:

1. Iterator permits the caller to remove the given elements from the specified collection during the iteration of the elements.
2. Method names have been enhanced.

Example Program:

```
import java.util.*;
class Demo8
{
public static void main(String args[])
 {
   ArrayList<String>   al = new  ArrayList<String>();
   al.add("A");
   al.add("B");
   al.add("C");
   al.add("D");
   al.add("E");
   Iterator<String>    itr = al.iterator();
   while(itr.hasNext())
     {
        String  x = itr.next();
        System.out.println(x);
     }
  }
}
```

The For-Each alternative

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

```
class Demo9
{
public static void main(String args[])
 {
    ArrayList<String>   al = new  ArrayList<String>();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    for(String   x: al)
    {
        System.out.println(x);
     }
   }
}
```

HashMap:

Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.

If you try to insert the duplicate key, it will replace the element of the corresponding key.

It is easy to perform operations using the key index like updation, deletion, etc.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized.

Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value.

It inherits the AbstractMap class and implements the Map interface.

Important Points

Java HashMap contains values based on the key.

Java HashMap contains only unique keys.

Java HashMap may have one null key and multiple null values.

Java HashMap is non synchronized.

Java HashMap maintains no order.

The initial default capacity of Java HashMap class is 16.

Example program:

```
import java.util.*;
class Demo14
{
  public static void main(String args[])
 {
    HashMap<String,Integer>    hm =  new   HashMap<String,Integer>();
    hm.put("bhaskar", 55000);
    hm.put("shiva",65000);
    hm.put("anil",35000);
    hm.put("younus",47000);
    Set< Map.Entry<String,Integer> >    s1 =    hm.entrySet();
    for(Map.Entry<String,Integer>   e : s1)
   {
      System.out.print(e.getKey() + ": ");
      System.out.println(e.getValue());
   }
```

```
  }
}
```

TreeMap

Java TreeMap class is a red-black tree based implementation.

It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

Java TreeMap contains values based on the key.

It implements the NavigableMap interface and extends AbstractMap class.

Java TreeMap contains only unique elements.

Java TreeMap cannot have a null key but can have multiple null values.

Java TreeMap is non synchronized.

Java TreeMap maintains ascending order.

Example program:

```
import java.util.*;
class Demo17
{
  public static void main(String args[])
  {
    TreeMap<String, Integer>   tm =  new  TreeMap<String, Integer>();
    tm.put("bhaskar", 55000);
    tm.put("shiva", 65000);
    tm.put("anil", 35000);
    tm.put("younus", 47000);
    Set<Map.Entry<String,Integer>>   s1 = tm.entrySet();
```

```
    for(Map.Entry<String,Integer> e : s1)
    {
        System.out.print(e.getKey() + ": ");
        System.out.println(e.getValue());
    }
  }
}
```

Comparators

Java Comparator interface is used to order the objects of a user-defined class.

This interface is in java.util package and contains 2 methods

1. compare(Object obj1,Object obj2)
2. equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Example program:

```
class  Emp
{
   String  name;
   int  exp;
  Emp(String  name, int exp)
  {
      this.name = name;
      this.exp = exp;
  }
  void  display()
```

```
  {
     System.out.print(name+ "  ");
     System.out.println(exp);
  }
}
import java.util.*;
class  Demo12
{
   public static void main(String a[])
  {
    TreeSet<Emp>    ts = new   TreeSet<Emp>(new A());
    Emp   e1 = new   Emp("shiva",15);
    Emp   e2 = new   Emp("younus",13);
    Emp   e3 = new   Emp("anil",12);
    Emp   e4 = new   Emp("ravi",18);
    ts.add(e1);
    ts.add(e2);
    ts.add(e3);
    ts.add(e4);
    for(Emp  x: ts)
     {
            x.display();
     }
 }
}
```

```
class A implements Comparator<Emp>
{
    public int compare(Emp e1, Emp e2)
    {
        if( e1.exp > e2.exp)
        {
            return 1;
        }
        else
        {
            return -1;
        }
    }
}
```

The Legacy Classes and Interfaces

Dictionary:

The Dictionary class operates much like Map and represents the key/value storage repository.

The Dictionary class is an abstract class that stores the data into the key/value pair.

We can define the dictionary as a list of key/value pairs.

Hashtable

It is similar to HashMap, but it is synchronized.

It provides concrete implementation of a Dictionary.

However, with the advent of collections, Hashtable was reengineered to also implement the Map interface.

Thus, Hashtable is integrated into the Collection Framework.

Example program:

```
import java.util.*;
class Demo4
{
public static void main(String args[])
 {
   Hashtable<String,Integer>   ht = new  Hashtable<String,Integer>();
   ht.put("cseE",70);
   ht.put("cseF",70);
   ht.put("cseG",60);
   ht.put("IT",60);
   System.out.println(ht);
 }
}
```

Properties

Properties is a subclass of Hashtable.

For each entry, key is a String and the value is also a String.

The Properties class is used by some other Java classes.

example:

System.getProperties( ) method returns environmental values.

Example program:

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

```
class Demo5
{
  public static void main(String args[])
 {
    Properties p = new Properties();
    p.put("Telangana","Hyderabad");
    p.put("Andhra Pradesh","Amaravathi");
    p.put("Tamilnadu","Chennai");
    p.put("Karnataka","Banglore");
    System.out.println(p);
  }
}
```

Stack:

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

With the release of JDK 5, Stack was retrofitted for generics

Example:

```
import java.util.*;
class Demo3
{
 public static void main(String args[])
 {
    Stack<Integer> st = new Stack<Integer>();
    st.push(30);
    st.push(20);
```

```
    st.push(10);
    System.out.println(st);
    st.pop();
    System.out.println(st);
  }
}
```

Vector

Vector implements a dynamic array.

It is similar to ArrayList, but with two differences:

1. Vector is synchronized

2. It contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework.

With the advent of collections, Vector was reengineered to extend AbstractList and to implement the List interface.

With the release of JDK 5, Vector was retrofitted for generics

Example:

```
import java.util.*;
class Demo1
{
  public static void main(String args[])
  {
     Vector<Integer> v = new Vector<Integer>(3, 2);
     System.out.println(v.size());
     System.out.println(v.capacity());
     v.addElement(5);
```

```
      v.add(6);
      v.add(0,4);
      System.out.println(v);
      System.out.println(v.capacity());
      v.addElement(7);
      System.out.println(v);
      System.out.println(v.capacity());
     System.out.println(v.firstElement());
     System.out.println(v.lastElement());
     if(v.contains(7))
         System.out.println("Vector contains 7");
   }
}
```

Enumeration interface:

Helps us to retrieve one object at a time from the group of objects.

This legacy interface works similar to Iterator interface of collection frame work.

Example:

```
import java.util.*;
class Demo2
{
  public static void main(String args[])
 {
    Vector<Integer>   v  =  new   Vector<Integer>();
    v.addElement(5);
```

```
    v.addElement(6);
    v.add(7);
    v.add(8);

    Enumeration<Integer>    e = v.elements();
    while(e.hasMoreElements())
   {
       System.out.println(e.nextElement());
    }
 }
}
```

More Utility classes

String Tokenizer

The java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class.

Example Program:

```
import java.util.*;
class Demo1
{
   static  String  s = "sunday,monday,tuesday,wednesday";
   public static void main(String args[])
  {
    StringTokenizer   st = new  StringTokenizer(s, ",");
```

```
    while(st.hasMoreTokens())
    {
      String   t  =  st.nextToken();
      System.out.println(t);
    }
  }
}
```

BitSet class**:**

The Java BitSet class implements a vector of bits.

The BitSet grows automatically as more bits are needed.

Each component of bit set contains at least one Boolean value.

The contents of one BitSet may be changed by other BitSet using logical AND, logical OR and logical exclusive OR operations.

The index of bits of BitSet class is represented by positive integers.

Each element of bits contains either true or false value.

Initially, all bits of a set have the false value.

A BitSet is not safe for multithreaded use without using external synchronization.

Example program:

```
import java.util.*;
class Demo2
{
public static void main(String args[])
 {
    BitSet   bits1 = new  BitSet(8);
```

```
 BitSet   bits2 = new  BitSet(8);
// set some bits
for(int i=0; i<8; i++)
{
        if((i%2) == 0)
                bits1.set(i);
        if((i%3) == 0)
                bits2.set(i);
}
System.out.println("Initial pattern in bits1: ");
System.out.println(bits1);
System.out.println("\nInitial pattern in bits2: ");
System.out.println(bits2);
// AND bits
bits2.and(bits1);
System.out.println("\nbits2 AND bits1: ");
System.out.println(bits2);
// OR bits
bits2.or(bits1);
System.out.println("\nbits2 OR bits1: ");
System.out.println(bits2);
// XOR bits
bits2.xor(bits1);
System.out.println("\nbits2 XOR bits1: ");
System.out.println(bits2);
```

```
  }
}
```

Date class:

The java.util.Date class represents date and time in java.

It provides constructors and methods to deal with date and time in java.

The java.util.Date class implements Serializable, Cloneable and Comparable<Date> interface.

It is inherited by java.sql.Date, java.sql.Time and java.sql.Timestamp interfaces.

After Calendar class, most of the constructors and methods of java.util.Date class has been deprecated.

Example program:

```
import java.util.Date;
class Demo3
{
   public static void main(String args[])
  {
      Date   d = new  Date();
      System.out.println(d);
     /* Display number of milliseconds since midnight, January 1, 1970 GMT */
     long   msec = d.getTime();
     System.out.println("Time: "+msec);
   }
}
```

Calendar class:

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc.

It inherits Object class and implements the Comparable interface.

Example program:

```
import java.util.*;
class Demo4
{
  public static void main(String args[])
  {
    Calendar  c1 = Calendar.getInstance();
    String  months[] = {
                "Jan", "Feb", "Mar", "Apr",
                "May", "June", "Jul", "Aug",
                "Sep", "Oct", "Nov", "Dec"};
    System.out.print("Date: ");
    System.out.print(c1.get(Calendar.DATE)+" ");
    int   m = c1.get(Calendar.MONTH);
    System.out.print(months[m]+" ");
    System.out.println(c1.get(Calendar.YEAR));
    System.out.print("Time: ");
    System.out.print(c1.get(Calendar.HOUR) + ":");
    System.out.print(c1.get(Calendar.MINUTE) + ":");
    System.out.println(c1.get(Calendar.SECOND));
  }
}
```

Random class:

Java Random class is used to generate a stream of pseudorandom numbers.

Example program:

```
import java.util.*;
class Demo5
{
  public static void main(String args[])
  {
     Random   r = new  Random();
     for(int i=1;i<=10;i++)
    {
       int   x = r.nextInt(100);
       System.out.println(x);
     }
  }
}
```

Formatter

The Formatter is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming.

The Formatter class is defined as final class inside the java.util package.

Example program:

```
import java.util.*;
class Demo6
```

```
{
  public static void main(String args[])
  {
     Formatter   fmt = new  Formatter();
     fmt.format("Formatting %s is easy %d %4.2f", "with Java", 10, 98.6);
     System.out.println(fmt);
     fmt.close();
   }
}
```

Scanner

Scanner class in Java is found in the java.util package.

Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default.

It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression.

It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

Example program:

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

```
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
        in.close();
    }
}
```