INTERFACES AND COLLECTION FRAMEWORK

1. INTERFACES

Interface

- An Interface is a specification of method prototypes i.e only method names are written in the interfaces without method bodies.
- An Interface will have 0 or more abstract methods which are all public and abstract by default.
- An Interface can have variables which are public, static and final by default, means all the variables of the interface are constants.
- Objects cannot be created to an interface whereas reference can be created.
- *Once interface is defined, any number of classes can implement an interface.*
- *Also one class can implement any number of interfaces.*
- To Implement an interface, a class must create the complete set of methods defined by the interface.
- By providing the interface keyword, java allows to fully utilize the "one interface, multiple methods" aspects of polymorphism.

1.1 Defining an Interface

- An Interface is basically a kind of class
- Like classes, interface contain methods and variables but with a major difference.
- *The difference is that interfaces define only*
  - *Abstract Method &*
  - *Final and Static Variables*
- i.*e methods* are declared without any body *and variables* are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized.
- All *Methods* and *Variables* in the interface are implicitly *public.*
- The syntax for defining an interface is very similar to that of defining a class

```
Interface InterfaceName
        {
                Variable declaration
```

```
        Method declaration
    }
```

Where *Interface* is the keyword and *InterfaceName* is any valid java variable

Example:

```
Interface Item
{
        static final int code = 1001;
        static final String name = "Fan";
        void display();
}
```

1.2 Implementing Interface

- *Once it is defined, any number of classes can implement an interface.*
- *Also, one class can implement any number of interface*
- To implement an interface, a class must create the complete set of methods defined by the interface.
- To implement an interface, include the *implements clause* in a class definition, and then create the methods defined by the interface.
- General form of a class that includes the implements clause looks like

```
Class ClassName [extends SuperClass]
        [implements Interface1[,... Interface N]]
{
        // class body
}
```

Example

```
Class A Extends B Implements I1,I2
{

}
```

- i.e if a class implements more than one interface, the interfaces are separated with a comma.
- Interface Methods must be declared as public,.
- Also the type signature of the implementing method must match exactly the type signature specified in the interface definition
- It is both permissible and common for classes that implement interfaces to define additional members of their own.

1.3 Interfaces can be Extended

- Like classes, interface can also be extended.
- i.e an interface can be subinterfaced from other interfaces.
- The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses.
- This is achieved using the keyword "extends".
- General form of extending interfaces is

```
Interface NameNew extends name1[,...nameN]
{
```

```
                Body of Interface
        }
```

Example

```
        interface A
        {
                void meth1();
                void meth2();
        }
        interface B extends A
        {
                void meth3();
        }
        Class  MyClass implements B
        {
                public void meth1()
                {
                        System.out.println("implementing meth1().....");
                }
                public void meth2()
                {
                        System.out.println("Implementing meth2()....");
                }
                public void meth3()
                {
                        System.out.println("Implementing meth3()....");
                }

        }
        Class InterfaceDemo
        {
                Public static void main(string args[])
                {
                        MyClass obj = new MyClass();
                        obj.meth1();
                        obj.meth2();
                        obj.meth3();
                }
        }
```

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Note : if a class that implements an interface and the class does not give implementations to all the methods of the interface, then the class becomes an abstract class and cannot be instantiated.

Interface Example

Interface Area

```
{
        final static float PI = 3.145;
        float compute(float x);
}
Class Square implements Area
{
        public float compute(float x)
        {
                return (x*x);
        }
}
Class Circle implements Area
{
        public float compute(float x)
        {
                return (PI*x*x);
        }
}
Class InterfaceDemo
{
        Public static void main(String args[])
        {
                Square s = new Square();
                double asqr = s.compute(10);
                Circle c = new Circle();
                double acir = c.compute(10);
                System.out.println("Area of Square:"+asqr);
                System.out.println("Area of Circle:"+acir);
        }
}
```

2. COLLECTION FRAMEWORK

2.1 COLLECTION OVERVIEW

A collection object or a container object is an object which can store a group of other objects. We are using a collection object to store 4 objects. A collection object has a class called as "collection class" or "container class". All the collection classes are available in the java.util package.

Collection Framework:

A group of collection classes is called a "collection framework" (or) A collection framework is a class library to handle group of objects. Collection framework is implemented in java.util package.

All the collection classes in java.util package are the implementation classes of different interfaces as shown in the table.

Interface type	Implementation classes
Set<T>	HashSet<T>
List<T>	Stack<T> LinkedList<T> ArrayList<T> Vector<T>
Queue<T>	LinkedList<T>
Map<K,V>	HashMap<K,V> Hashtable<K,V>

Sets:A set represents a group of elements arranged just like an array. The set will grow dynamically when the elements are stored into it. A set will not allow duplicate elements. If we try to pass the same element that is already available in the set,then it is not stored into the set.
Lists: Lists are like sets. They store a group of element, but lists allow duplicate values to be stored.
Queues: A Queue represents arrangement of elements in FIFIO(First In First Out) order. This means that an element that is stored as a first element into the queue will be removed first from the queue.
Maps: Maps store elements in the form of key and value pairs. If the key is provided then its corresponding value can be obtained. Of course, the keys should have unique values.

2.2 COLLECTION INTERFACES

Collection framework defines several interfaces. This will provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes.

The List Interface:

The **List Interface** extends **Collection** and declares the behaviour of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list.

Some of the methods of List Interface are

METHOD	DESCRIPTION
Void add(int index, E obj)	Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
Boolean addAll(int index, collection<? Extends E> c)	Inserts all elements of c into the invoking list at the index passed in index. Returns true , if the invoking list changes and returns false otherwise.

E get(int index)	Returns the object stored at the specified index within the invoking collection.
Int indexOf(object obj)	Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
Int lastIndexOf(Object obj)	Returns the index of the last index of obj in the invoking list. If obj is not an element of the list ,-1 is returned.
E removed(int index)	Removes the element at the position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.

The Set Interface:

- **Set** is an **interface** which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.
- Basically, **Set** is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Set Interface declaration
  public interface Set<E> **extends** Collection<E>
- **Set** has various **methods** to add, remove clear, size, etc to enhance the usage of this **interface**.

2.3 COLLECTION CLASSES

Some of the collection classes provide full implementations that can be used as - is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. The standard collection classes are summarised in the following table.

Class	Description
HashSet	Extends AbstractSet for use with the Hash table.
LinkedHashSet	Extends Hashset to allow insertion order itereations.
LinkedList	Implements a LinkedList by extending AbstractSequentialList
ArrayList	Implements a dynamic array by extending AbstractList.
Stack	Implements List interface
Vector	Implement List Interface

❖ **The ArrayList Class:** An ArrayList is like an array, which can grow in memory dynamically. It means that when we stored elements into the ArrayList, depending on the

number of elements, the memory is dynamically allotted and re-allotted to accommodate all the elements. ArrayList is not synchronized. This means that when more than one thread acts simultaneously on the ArrayList object, the results may incorrect in some cases.

ArrayList can be written as

class ArrayList<E>

For Example, to store String type elements, we can create an objects to ArrayList as:

ArrayList<String>arl = new ArrayList<String>();

ArrayList class includes the following methods:

- **boolean add(element obj):** This method add an element to the array list. It returns true if the element is added successfully.
- **void add(intposition, element obj):** This method inserts the specified elements at the specified position in the ArrayList
- **element remove(int position):**This method removes an element at the specified position in array list.
- **boolean remove (Object obj):** This method removes the first occurrence of the specified element obj from the ArrayList, if it present.
- **void clear():** This method removes all the elements from the ArrayList.
- **element set(int position, element obj):** This method replaces an element at the specified position in the ArrayList with the specified element obj.
- **boolean contains(Object obj):** This method returns the element available at the specified position in the ArrayList.
- **element get(int position):** this method returns the element available at the specified position in the ArrayList.
- **intindexOf(Object obj):** This method returns the position of the first occurrence of the specified element obj in the list, or -1 if the element is bot found in the list.
- **intlastIndexOf(Object obj):** This method returns the position of the last occurrence of the specified element obj in the list, or -1 if the element is bot found in the list.
- **int size():** This method returns the number of elements present in the ArrayList.
- **Object[] toArray():** This method returns an Object class type array containing all the elements in the ArrayList in proper sequence.

Program:

```
import  java.util.*;
Class ArraylistDemo
{
        public static void main(String args[])
        {
                ArrayList<String>arl=new ArrayList<String();
                arl.add("JAVA");
                arl.add("CN");
                arl.add("OS");
                arl.add("PSQT");
                arl.add("AFM");
                System.out.println("Element in ArrayList are" +arl);
                //remove two objects
```

```
                arl.remove(3);
                boolean b=arl.remove("JAVA");
                //display again
                System.out.println("Element in Array list after removing are "+arl);
                Iterator it =arl.iterator();
                while(it.hasNext())
                {
                        String s=(String)it.next();
                        System.out.println(s);
                }
        }
}
```

OUTPUT:
Element in ArrayList are[JAVA, CN, OS, PSQT, AFM]
Element in Array list after removing are [CN, OS, AFM]
CN
OS
AFM

- **LinkedList Class:** A Linked list contains a group of elements in the form of nodes. Each node will have three fields- the data field contains data and the link fields contain references to previous and next nodes.
  Linked list is very convenient to store data. Inserting the elements into the linked list and removing the elements from the linked list is done quickly and takes the same amount of time. A linked list is written in the form of: **class LinkedList<E>**
  For Example, to store String type elements, we can create an objects to LinkedList as:
  LinkedList<String>ll = new LinkedList<String>();
  LinkedList class includes the following methods;
  - **boolean add(element obj):** This method add an element to the linked list. It returns true if the element is added successfully.
  - **void add(int position, element obj):** This method inserts the specified elements at the specified position in the linked List
  - **voidaddFirst(element obj):** This method adds the element obj at the first position of the linked list.
  - **void addLast(element obj):** This method appends the specified element to the end of the linked list
  - **elementremoveFirst():** This method removes the first element from the linked list and returns it.
  - **element removeLast():** This method removes the last element from the linked list and returns
  - **element remove(int position):** This method removes an element at the specified position in the linked list
  - **void clear():** This method removes all the elements from the linked list.
  - **element get(int position):** This method returns the element at the specified position in the linked list
  - **element getFirst():** This method returns the first element from the list.

- **Element getLast():** This method returns the last element from the list.
- **int indexOf(Object obj):** This method returns the position of the first occurrence of the specified element obj in the list, or -1 if the element is not found in the list.
- **intlastIndexOf(Object obj):** This method returns the position of the last occurrence of the specified element obj in the list, or -1 if the element is not found in the list.
- **int size():** This method returns the number of elements present in the linked List.
- **Object[] toArray():** This method converts the linked list into an array of Object class type. All the elements of the linked list will be stored into the array in the same sequence.

Program:

```
//Demonstrate Linkedlist class
import java.util.*;
class  LLDemo
{
        public static void main(String args[])
        {
                LinkedList<String>ll= new LinkedList<String>();
                ll.add("Java");
                ll.add("Computer Network");
                ll.add("Operating System");
                ll.add("Accounts");
                ll.add("Statistics");
                System.out.println("LinkedList elements are:");
                for(String i:ll);
                {
                        System.out.println(i);
                }
        }
}
```

OUTPUT:

```
LinkedList Elements are:
Java
Computer Network
Operating System
Accounts
Statistics
```

❖ **HashSet Class:** A HashSet represents a set of elements (objects). It does not guarantee the order of elements. Also it does not allow the duplicate elements to be stored.A HashSet is written in the form of: **class HashSet<E>**

For Example, to store String type elements, we can create an objects toHashSetas:

```
HashSet<String>hs= new HashSet<String>();
```

HashSet class includes the following methods.

- **boolean add(obj):** This method add an element obj to the HashSet. It returns true if the element is removed successfully,else it return false.

- **boolean remove(obj):** This methods removes the element obj from the HashSet. If it is present. It returns true if the element is removed successfully otherwise false.
- **void clear():** This removes all the elements from the HashSet.
- **boolean contains(obj):** This returns true if the HashSet contains the specified element obj.
- **booleanisEmpty():** This returns true if the HashSet contains no elements.
- **int size():** This returns the number of elements present in the Hashset.

Program// Hashset Demo

```
import java.util.*;
class  HSDemo
{
        public static void main(String args[])
        {
                HashSet<string> hs= new HashSet<String>();
                hs.add("Java");
                hs.add("Computer Network");
                hs.add("Operating System");
                hs.add("Accounts");
                hs.add('Statistics");
                System.out.println("HashSet elements are:");
                for(String i:hs);
                {
                        System.out.println(i);
                }
        }
}
```

OUTPUT:

Hash elements are:
Java
Computer Network
Operating System
Accounts
Statistics

❖ **LinkedHashSet:**This is a subclass of HashSet class and does not contain any additional members on its own. It is a generic class that has the declaration: **class LinkedList<E>**
For Example, to store String type elements, we can create an objects toLinkedHashSetas:

```
LinkedHashSet<String>lh= new LinkedHashSet<String>();
```

LinkedHashSet class includes the following methods.

- **boolean add(obj):** This method tests whether the stack is empty or not. If the stack is empty then true is returned otherwise false.
- **element peek():** This method returns the top-most object from the stack without removing it.

- **element pop():** This method pops the top-most element form the stack and returns it
- **element push(element obj):** This method pushes an element obj onto the top of the stack and returns that element.
  - **int search(Object obj):** This method returns the positionof an element obj form the top of the stack. If the element is not found in the stack then returns -1

Program:

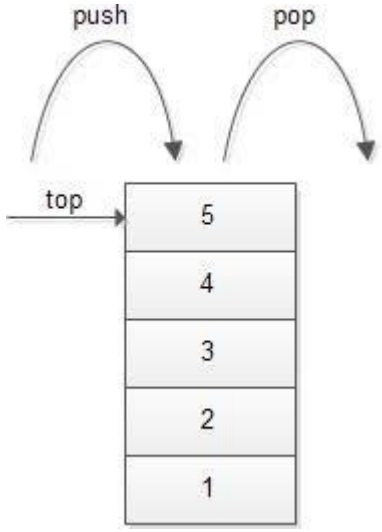
```
//Demonstrate linkedHashset Class
import java.util.*;
class  LinkedHashDemo
{
    public static void main(String args[])
    {
            LinkedHashSet<String> lhs= new LinkedHashSet<String>();
            lhs.add("Mango");
            lhs.add("Banana");
            lhs.add("Apple");
            lhs.add("Grapes");
            lhs.add("Orange");
            System.out.println("LinkedHashSet Elements are:");
            iterator it=lhs.iterator();
            while(it.hasNext())
            {
                    String s=(String)it.next();
                    System.out.println(s);
            }
    }
}
```

OUTPUT:

```
LinkedHashSet Elements are:
Mango
Banana
Apple
Grapes
Orange
```

❖ Stack class:

A stack represents a group of elements stored in LIFO(Last In First Out) order. This means that the element which is stored as a last element into the stack will be first element to be removed from the stack. Inserting elements (objects) into the stack is called "push operation" and removing elements from stack is called "pop operation". Searching for an element in the stack is called a side of the stack, called 'top' of the stack, as shown in the following figure:



A pile of places in a cafeteria where the lastly washed plate will be coming out first can be taken as an example for a stack. We can write a stack class as:

class Stack<E>

where E stands for element type. Suppose, we want to create a stack object that contains Integer objects, we can do so as shown here:

Stack<Integer> obj= new Stack<Integer>( );

Stack class includes the following methods:

i. boolean empty( ): This method tests whether the stack is empty or not. If the stack is empty then true is returned otherwise false.

ii. element peek( ): This method returns the top-most object from the stack without removing it.

iii. element pop( ): This method pops the top-most element from the stack and returns it.

iv. element push(element obj): This method pushes an element obj onto the top of the stack and returns that element.

v. int search(Object obj): This method returns the position of an element obj from the top of the stack. If the element(object) is not found in the stack then it returns

Example of using the Stack class:

```
import java.util.*;
public class Stackex
{
    Public static void main(String args[])
{
    Stack st=new Stack( );
```

```
        st.push("Java");
        st.push("Latest");
        st.push("Edition");
        st.push("Fifth");
        System.out.println("The elements in the stack:"+st);
        System.out.println("The element at the top:"+st.peek( ));
        System.out.println("The element popped out of the stack:"st.pop( ));
        System.out.println("The element in a stack after pop out anelement:"+st);
        System.out.println("The result of searching:"+st.search("r" "u"));
    }
}
```

Output:

```
The elements in the stack:[Java, Latest, Edition, -Fifth]
The element at the top: -Fifth
The element popped out of the stack: -Fifth
The element in a stack after pop out anelement: :[Java, Latest, Edition]
The result of searching: -1
```

❖ **Vector class:**

- A vector also stores elements(objects) similar to ArrayList, but vector is synchronized.
- It means even if several threads act on vector object simultaneously, the results will be reliable.
- We can write a Vector class as:
- **class Vector<E>**
- where E, represents the type of elements stored into the vector.
- For example, if we want to create an empty Vector that can be used to store Float type objects, we can write as:

```
Vector<Float> v= new Vector<Float>( );
```

vector class includes the following methods

1) **void add(int index, object element)**
   Inserts the specified element at the specified position in this vector
2) **boolean add(Object 0)**
   Appends the specified element to the end of this vector.
3) **Boolean addAll(Collection c)**
   Appends all of the elements in the specified Collection to the end of this Vector.
4) **void addElement(Object obj)**
   Adds the specified component to the end of this vector, increasing its size by one.
5) **boolean contains(Object elem)**
   Tests if the specified object is a component in this vector.
6) **Object elementAt(int index)**
   Returns the component at the specified index.
7) **int indexOf(Object elem)**

Searches for the first occurence of the given argument, testing for equality using the equals method.

8) **boolean isEmpty()**

Tests if this vector has no components.

Example of using the Vector class:

/* PROGRAM TO ILLUSTRATE Vector CLASS*/

```
import java.util.*;
import java.awt.*;
class VectorDemo
{
	public static void main(String args[])
	{
		Vector<Integer> V=new Vector<Integer>();
		int x[ ]={22,20,10,40,15,60};
		for(int i=0;i<x.length;i++)
		{
			V.add(x[i]);
		}
		System.out.println("Vector Elements:");
		for(int i=0;i<V.size();i++)
		{
			System.out.println(V.get(i));
		}
		System.out.println("Retrieving Elements using ListIterator Interface:");
		ListIterator Lit= V.listIterator();
		System.out.println(" In Forward direction:");
		while(Lit.hasNext())
		System.out.print(Lit.next()+"\t");
		System.out.println("\n In BackWard direction:");
		while(Lit.hasPrevious())
		System.out.print(Lit.previous()+"\t");
	}
}
```

OUTPUT:

```
Z:\>javac VectorDemo.java
Z:\>java VectorDemo
Vector Elements:
22
20
10
```

```
40
15
60
Retrieving  Elements using ListIterator Interface
 In Forward direction:
22      20      10      40      15      60
 In BackWard direction:
60      15      40      10      20      22
```

2.4 Retrieving elements from Collections

Following are the 4 ways to retrieve any elements form a collection object:

- **Using for-each loop.**
- **Using Iterator interface.**
- **Using ListIterator interface.**
- **Using Enumeration interface.**

For-each Loop: for-each loop is like for loop which repeatedly executes a group of statements for each element of the collection. The format is

```
for(variable: collection-object)
{
Statements;
}
```

Here, the variable assumes each element of the collection-object and the loop is executed as many times as there are number of elements in the collection-object. If collection-object has n elements the loop is executed exactly n times and the variable stores each element in each step.

Iterator Interface : Iterator is an interface that contains methods to retrieve the elements one by one from a collection object. It has 3 methods**:**

- **booleanhasNext():** This method returns true if the iterator has more elements.
- **element next**(): This method returns the next element in the iterator.
- **void remove**(): This method removes form the collection the last element returned by the iterator

ListIterator Interface: ListIterator is an interface that contains methods to retrieve the elements from a collection object, both in forward and reverse directions. It has the following methods

- **boolean hasNext():** This returns true if the ListIterator has more elements when traversing the list in the forward direction.
- **boolean hasPrevious():** This returns true if the ListIterator has more elements when traversing the list in the reverse direction.
- **element next():** This returns the next element in the list.
- **element previous():** This returns the previous elemnt.
- **void remove():** This removes from the list the last element that was returned by the next() or previous() methods.

Note : Difference between Iterator and ListIterator : Both are useful to retrieve elements from collection. Iterator can retrieve the elements only in forward direction. But ListIterator can retrieve the elements in forward and backward direction also. So ListIterator is preffered to iterator.

3. UTILITY CLASSES

3.1 StringTokenizer Class

This class is useful to break a string into pieces, called 'tokens'. These tokens are then stored in the StringTokenizer object form where the can be retrieved. The code to create an object to StringTokenizer class is:

StringTokenizerst= new StringTokenizer (str, "delimiter"0;

The actual string str is broken into pieces at the positions marked by a group of characters, called 'delimiters'. For example, to break the string wherever a comma is found, we can write:

StringTokenizerst =new StringTokenizer ("Keep, Smiling, Always");

StringTokenizer Class Methods

StringTokenizer class includes the following methods:

- **IntcountTokens():** This method counts and returns the number of tokens available in a StringTokenizer object.
- **Boolean hasMoretokens():** This method test if there are more tokens available in the StringTokenizer object or not. If next token is there then it returns true.
- **String nextToken():**This method returns the next token form the StringTokenizer.

Program:

```
//Demonstrate StringTokenizer class
import  java.util.*;
class StringTokenizerDemo
{
        public static void main(String args())
        {
                int sum=0,c=0;
                String str ="Keep Smiling Always";
                StringTokenizer st =new StringTokenizer(str);
                System.out.println("The given string is: "+str);
                c=st.CountTokens();
                While(st.hasMoreTokens())
                {
                        Stringtoken=st.nextToken();
                        System.out.println(token);
                }
                System.out.println(" Total No. of tokens in the given string is "+c);
        }
}
```

Output

```
Z:>\javac  StringTokenizerDemo.java
Z:\javac StringTokenizerDemo
```
The given string is: Keep Smiling Always
Keep
Smiling
Always
Total no. of tokenin the given string is 3

3.2 Scanner

Scanner is the complement of Formatter. Added by JDK 5, Scanner reads formatted input and converts it into its binary form. It is used to read input from different sources like disk file , keyboard etc. It makes easy to read all types of numeric values, strings and other types of data, whether it comes from a disk file keyboard, another source.
When the scanner class receives the input into it breaks the input into several tokens.
Scanner can be created for a String, an InputStream, a File, or any object that implements the Readable or ReadableByteChannel interfaces. The following sequence creates a Scanner that reads the file Test.txt:

```
FileReader fin =new FileReader("Test.txt"0;
Scanner src=new Scanner(fin);
```

The tokens can be retrieve from the scanner object using the following methods:

- **next():** to read a string
- **nextByte():** to read byte value
- **nextInt():** to read an integer value
- **nextFloat():** to read a float value
- **nextLong():** to read long value
- **nextDouble:**(): to read double value

Program:

```
//Demonstrate Scanner Class
import java.util.Scanner;
class  ScannerDemo
{
public static void main(String args[])
        {
                System.out.println("Enter ID, Name,Course,Salary");
                Scanner sc=new Scanner(System.in);
                int id=sc.nextInt();
                String name=sc.next();
                String course=sc.next();
                float salary=sc.nextFloat();
                System.out.println("ID of Employee:" +id)
                System.out.println("Name of the Employee:" +name);
                System.out.println("Qualification: +course);
                System.out.println("Salary: " +salary);
        }
}
```

Output:

```
Z:\>javac ScannerDemo.java
Z:\>java ScannerDemo
```

Enter ID, Name, Course,Salary : 001 GoldSmith MCA 15000
ID of Employee: 001
Name of the Employee: GoldSmith
Qualification: MCA
Salary: 15000

Setting Delimiters:

Scanner breaks input into tokens based on default delimiters i.e white space. This default delimiters can change using methods

```
Scanner useDelimiters(String pattern)
Scanner useDelimiters(pattern pattern)
```

Here pattern is a regular expression that specifies the delimiters set.

Other Scanner Features:

findInLine Method is used to search for the specified pattern within the next line of text. If the pattern is found, matching token is consuming and returned otherwise null is returned.

Program:

```
import java.util.*;
class FindlnLineDemo
{
        public static void main(String args())
        {
                String instr=”Name: Chinna Age:21 ID:13”;
                Scanner conin=new Scanner(instr);
                Conin.findInLIne(“Age”);
                if(conin.hasNext())
                System.out.println(conin.next());
                else
                System.out.println(“Error!”);
        }
}
```

Note: The Output is **21**the above program in the line is used to findInLine method and occurrence of the pattern Age. Once found the next token read, which is the age.