Object Oriented Programming using Java

Prepared By:
Suyel, PhD
Assistant Professor
Dept. of CSE, NIT Patna

Outline

Collections Framework

- Collections framework is a hierarchy of interfaces and classes that provides state-of-the-art technology.
- It provides many interfaces (Set, List, etc.) and classes (ArrayList, LinkedList, etc.).
- Collections framework can achieve all the operations that we perform on data, such as searching, sorting, etc.
- java.util package contains a large number of interfaces and classes that support many functionalities.

Collections Interfaces

- Some are given below:

Interface	Description
Collection	Enables us to work with groups of objects. It is at top.
List	Extends Collection to handle sequences of list of objects.
Queue	Extends Collection to handle special type of List.
Deque	Extend Queue to handle double ended queue.
Set	Extends Collection to handle similar type of elements.
SortedSet	Extends Set to sort.
NavigableSet	Extends SortedSet to retrieve the elements based on closest-match searches.

Collections Interfaces (Cont...)

- **Collection Interface:**
  - The Collection interface is the foundation upon which the Collections Framework is built.
  - Collection extends the Iterable interface.
  - It declares the core methods that all collections have. Some are shown in the next slide.
  - These methods throw different kind of Exception (See Book).
  - Declaration:

    interface Collection <E>

    Here, E specifies the type of the objects that the collection can hold.

Collections Interfaces (Cont...)

Method	Description
boolean add(E *obj*)	Adds obj to the invoking collection. Returns true, if it is added.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of c to the invoking collection.
void clear( )	Removes all elements from the invoking collection.
boolean remove(Object *obj*)	Removes one instance of obj from the invoking collection.
boolean removeAll(Collection<?> *c*)	Removes all elements of c from the invoking collection.
boolean equals(Object *obj*)	Returns true, if the invoking collection and obj are equal.
boolean contains(Object *obj*)	Returns true, if obj is an element of the invoking collection.
boolean containsAll(Collection<?> *c*)	Returns true, if the invoking collection contains all elements of c.

Collections Interfaces (Cont...)

- **List Interface:**
  - List interface extends Collection and declares the behavior of a collection that stores a sequence of elements.
  - Element can be inserted or accessed by the position (index) in the list.
  - A list may contain duplicate elements.
  - Declaration:

    interface List <E>
  - There are many methods of List Interface (Next Slide).
  - A method throw UnsupportedOperationException, if the list cannot be modified.

Collections Interfaces (Cont...)

Method	Description
void add(int *index*, E *obj*)	Inserts obj into the invoking list at the particular index.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all the elements of c into the invoking list at the index passed in index.
E get(int *index*)	Returns the object stored at the specified index.
E remove(int *index*)	Removes the element at position index and returns the deleted element.
E set(int *index*, E *obj*)	Assigns obj to the specified index.
int indexOf(Object *obj*)	Returns the index of the first instance of obj. If obj is not an element of the list, –1 is returned.
ListIterator<E> listIterator()	Returns an iterator to the start of the invoking list.

Collections Interfaces (Cont...)

- **Set Interface:**
  - Set interface extends Collection and declares the behavior of a collection by defining a set.
  - It does not allow duplicate elements.
  - Here, add( ) method returns, false if an attempt is made to add duplicate elements to a set.
  - Set interface does not have any additional methods.
  - Declaration:

    interface Set<E>

Collections Interfaces (Cont...)

SortedSet Interface:

- SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.
- This interface has its own methods, which are shown in the next slide.
- SortedSet throw several exceptions.
- Declaration:

  interface SortedSet<E>

Collections Interfaces (Cont...)

Method	Declaration
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
E first( )	Returns the first element in the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a SortedSet containing those elements less than *end* that are contained in the invoking sorted set.
SortedSet<E> tailSet(E *start*)	Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a SortedSet that includes those elements between *start* and *end*–1.

Collections Interfaces (Cont…)

- **NavigableSet Interface:**
  - NavigableSet interface extends SortedSet.
  - It declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.
  - Dclaration:

    interface NavigableSet<E>
  - This interface has its own methods. Some are given in the next slide.

Collections Interfaces (Cont...)

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element e such that e >= obj. If such an element is found, it is returned. Otherwise, null is returned.
E floor(E *obj*)	Searches the set for the largest element e such that e <= obj.
E higher(E *obj*)	Searches the set for the largest element e such that e > obj.
E lower(E *obj*)	Searches the set for the largest element e such that e < obj.
NavigableSet<E> descendingSet( )	Returns a NavigableSet that is the reverse of the invoking set.
E pollFirst( )	Returns the first element, removing the element in the process.
E pollLast( )	Returns the last element, removing the element in the process.

Collections Interfaces (Cont...)

Queue Interface:

- Queue interface extends Collection interface.

- It declares the behavior of a queue in the first-in, first-out manner.

- Queue is a generic interface.

- Declaration:

  interface Queue<E>

- There are 5 methods, which are mentioned in the next slide.

Collections Interfaces (Cont...)

Method	Description
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns true, if obj is added.
E element( )	Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException, if queue is empty.
E peek( )	Returns the element at the head of the queue. The element is not removed. It returns null, if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException, if the queue is empty.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns null, if the queue is empty.

Collections Interfaces (Cont...)

- **Deque Interface:**
  - Deque interface extends Queue.
  - It can function as standard, first-in, first-out queues or as last-in, first-out stacks.
  - Declaration:

    interface Deque<E>
  - Along with the methods of Queue, Deque adds some methods. Some of them are mentioned in the next slide.

Collections Interfaces (Cont...)

Method	Description
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an IllegalStateException, if a capacity-restricted deque is out of space.
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an IllegalStateException, if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail or end of the deque. Throws an IllegalStateException, if a capacity-restricted deque is out of space.
E getFirst( )	Returns the first element of the deque. The object is not removed from the deque. It throws NoSuchElementException, if the deque is empty.
E getLast( )	Returns the last element of the deque. The object is not removed from the deque. It throws NoSuchElementException, if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws NoSuchElementException, if the deque is empty.

Collections Interfaces (Cont...)

- **Conditions to throw Exceptions:**

Exception	Description
ClassCastException	When an object is incompatible
NullPointerException	When an attempt is made to store a null object
IllegalArgumentException	When an invalid argument is used
IllegalStateException	When an attempt is made to add an element to a fixed-length queue
NoSuchElementException	When an attempt is made to remove an element from an empty queue
UnsupportedOperationException	When an attempt is made to change an unmodifiable list.

Collection Classes

- Collection classes implements the Interfaces discussed so far.
- Some of the classes provide full implementations that can be used as-is.
- However, some other are abstract, which provide skeletal implementations that are used as starting points for creating concrete collections.
- There are many Collection classes. Some of them are mentioned in the next slide.

Collection Classes (Cont…)

Class	Description
AbstractCollection	Implements most of the Collection interface.
AbstractList	It extends AbstractCollection and implements most of the List interface.
AbstractQueue	It extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	It extends AbstractList and uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending AbstractSequentialList.
ArrayList	Implements a dynamic array by extending AbstractList.
HashSet	Extends AbstractSet to support a hash table.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.

Collection Classes (Cont…)

- **ArrayList Class**
  - ArrayList class extends AbstractList and implements the List interface.
  - ArrayList is a generic class that has the declaration:

    class ArrayList<E>
  - ArrayList supports dynamic arrays that can grow as needed.
  - There are three constructor in ArrayList:

    ArrayList() //It is used to build an empty array list

    ArrayList(Collection<? extends E> c) //It is used to build an array list that is initialized with the elements of the collection c.

    ArrayList(int capacity) //Used to build an array list with initial capacity

Collection Classes (Cont…)

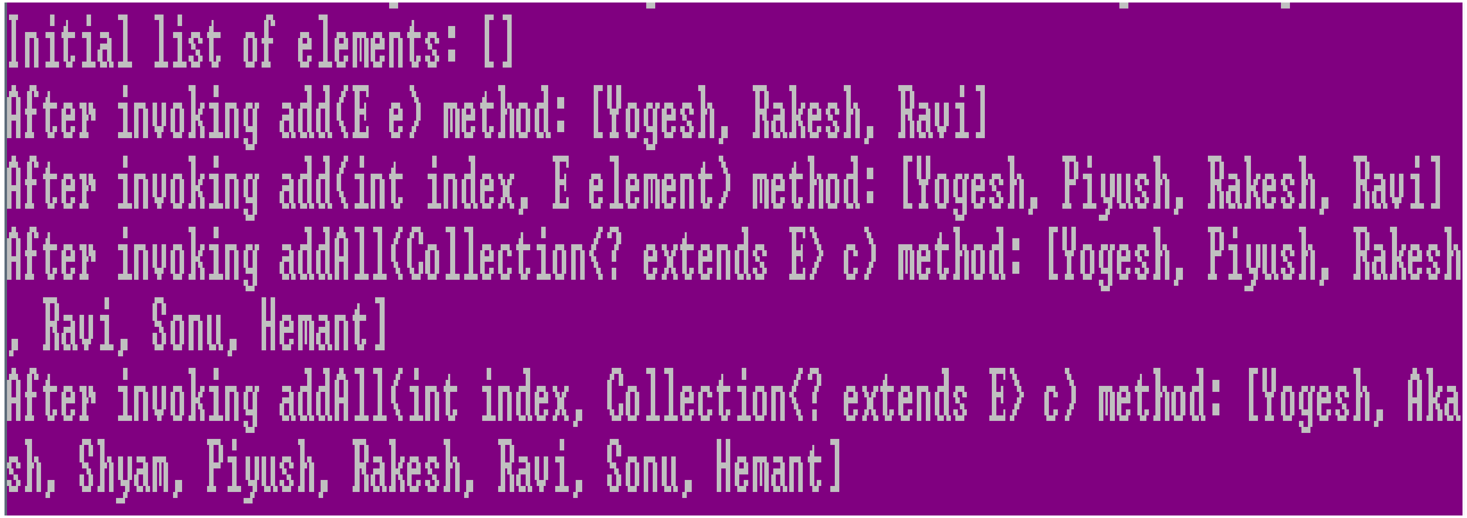
❑ ArrayList Class (Cont…)

```
import java.util.*;
class ArrayList1
{
  public static void main(String args[])
  {
    ArrayList<String> al1=new ArrayList<String>();
    System.out.println("Initial list of elements: "+al1);
    al1.add("Yogesh");      //Adding elements to the end of the list
    al1.add("Rakesh");
    al1.add("Ravi");
    System.out.println("After invoking add(E e) method: "+al1);

    al1.add(1, "Piyush");  //Adding an element at the specific position
    System.out.println("After invoking add(int index, E element) method: "+al1);

    ArrayList<String> al2=new ArrayList<String>();
    al2.add("Sonu");    //Adding elements to the end of the list
    al2.add("Hemant");

    al1.addAll(al2);      //Adding second list elements to the first list
    System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al1);

    ArrayList<String> al3=new ArrayList<String>();
    al3.add("Akash");     //Adding elements to the end of the list
    al3.add("Shyam");

    al1.addAll(1, al3);   //Adding second list elements to the first list at specific position
    System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+al1);
  }
}
```

Collection Classes (Cont…)

❑ **ArrayList Class (Cont…)**

Output



```
Initial list of elements: []
After invoking add(E e) method: [Yogesh, Rakesh, Ravi]
After invoking add(int index, E element) method: [Yogesh, Piyush, Rakesh, Ravi]
After invoking addAll(Collection<? extends E> c) method: [Yogesh, Piyush, Rakesh
, Ravi, Sonu, Hemant]
After invoking addAll(int index, Collection<? extends E> c) method: [Yogesh, Aka
sh, Shyam, Piyush, Rakesh, Ravi, Sonu, Hemant]
```

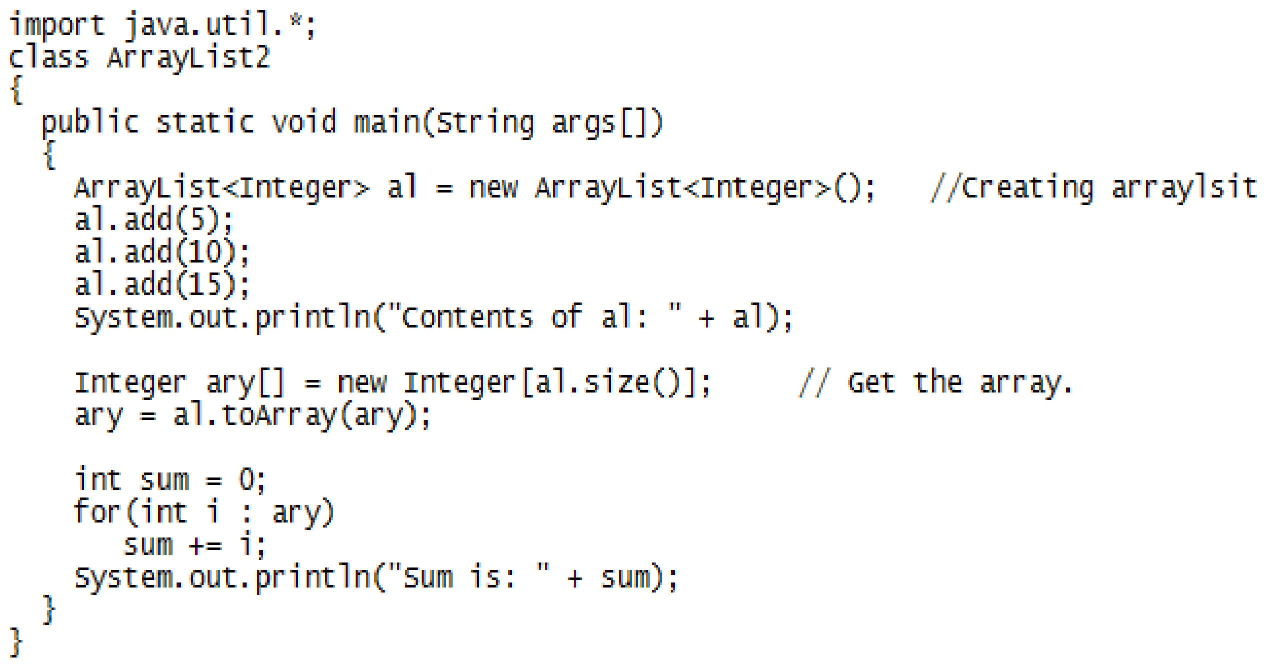
Collection Classes (Cont…)

- **ArrayList Class (Cont…)**
- **Obtaining an array from an ArrayList**
  - We can obtain an array that contains the contents of the list by using toArray( ).
  - There could be many reasons to convert a collection into an array:
    - To obtain faster processing times for certain operations.
    - To pass an array to a method that is not overloaded to accept a collection
  - There are two versions of toArray( ):
    - Object[ ] toArray( )
    - <T> T[ ] toArray(T *array[ ]*)

  The first form returns an array of Object. The second returns an array of elements that have the same type as T.

Collection Classes (Cont…)

- ArrayList Class (Cont…)
- Obtaining an array from an ArrayList (Cont…)



```
import java.util.*;
class ArrayList2
{
  public static void main(String args[])
  {
    ArrayList<Integer> al = new ArrayList<Integer>();   //Creating arraylsit
    al.add(5);
    al.add(10);
    al.add(15);
    System.out.println("Contents of al: " + al);

    Integer ary[] = new Integer[al.size()];     // Get the array.
    ary = al.toArray(ary);

    int sum = 0;
    for(int i : ary)
       sum += i;
    System.out.println("Sum is: " + sum);
  }
}
```

Collection Classes (Cont…)

- **ArrayList Class (Cont…)**
- **Obtaining an array from an ArrayList (Cont…)**
  - **Output**

```
Contents of al: [5, 10, 15]
Sum is: 30
```

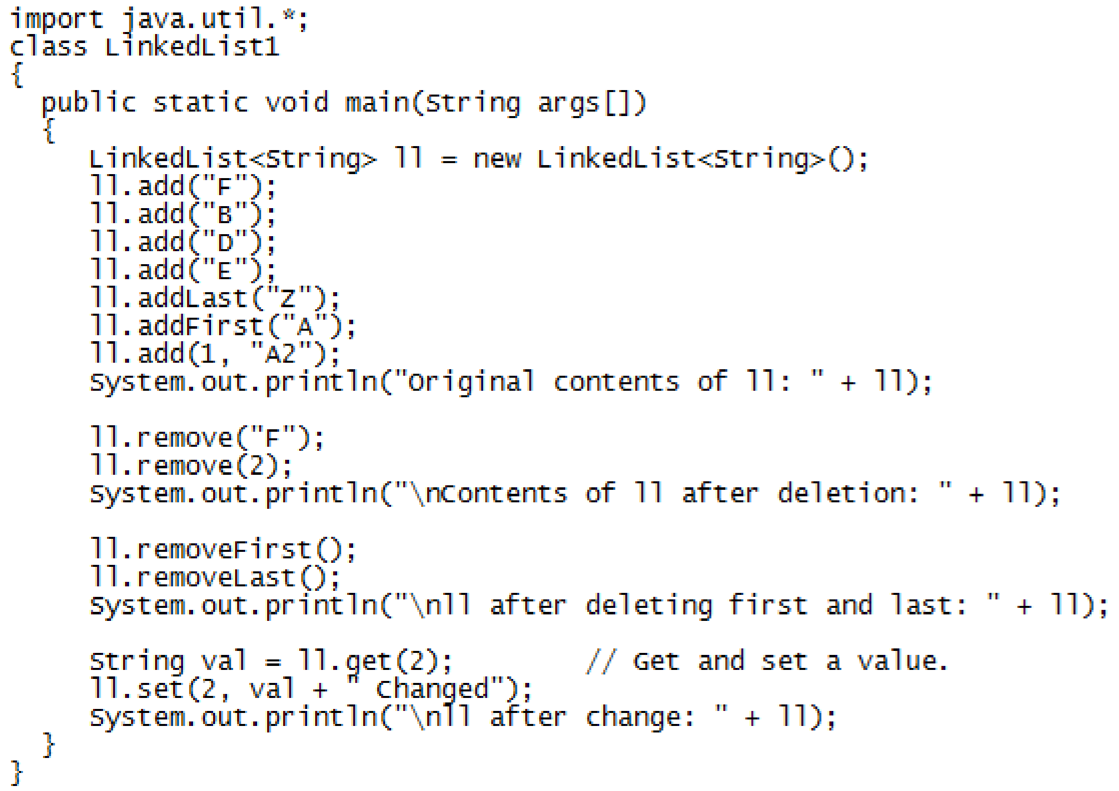
Collection Classes (Cont…)

- **LinkedList Class:**
  - LinkedList class extends AbstractSequentialList. It provides a linked-list data structure.
  - It implements List, Deque and Queue interfaces.
  - Declaration:

    class LinkedList<E>
  - There are two constructors:
    - LinkedList( ) //An empty linked list
    - LinkedList(Collection<? extends E> *c*) //A linked list that is initialized with the elements of the collection *c*.

Collection Classes (Cont…)

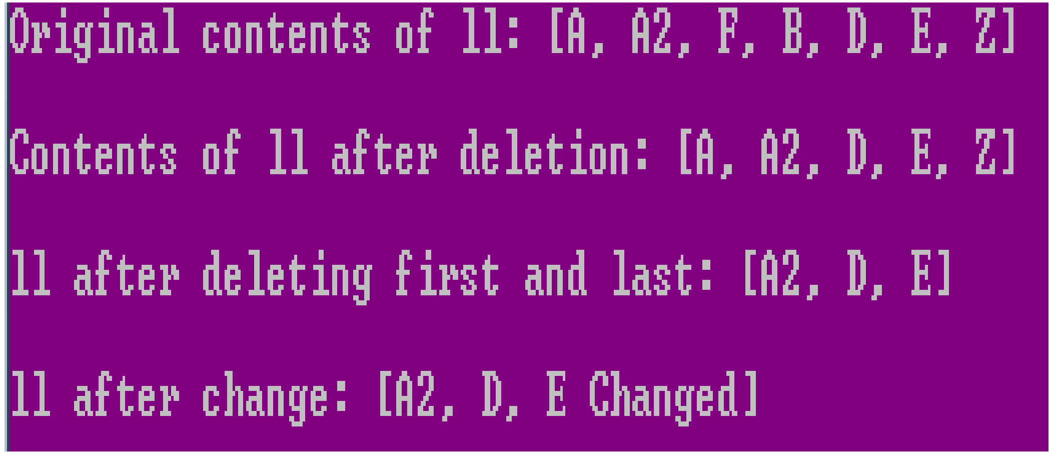
- **LinkedList Class (Cont…):**



```
import java.util.*;
class LinkedList1
{
  public static void main(String args[])
  {
     LinkedList<String> ll = new LinkedList<String>();
     ll.add("F");
     ll.add("B");
     ll.add("D");
     ll.add("E");
     ll.addLast("Z");
     ll.addFirst("A");
     ll.add(1, "A2");
     System.out.println("Original contents of ll: " + ll);

     ll.remove("F");
     ll.remove(2);
     System.out.println("\nContents of ll after deletion: " + ll);

     ll.removeFirst();
     ll.removeLast();
     System.out.println("\nll after deleting first and last: " + ll);

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

Collection Classes (Cont…)

- LinkedList Class (Cont…):
  - Output



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

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

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

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

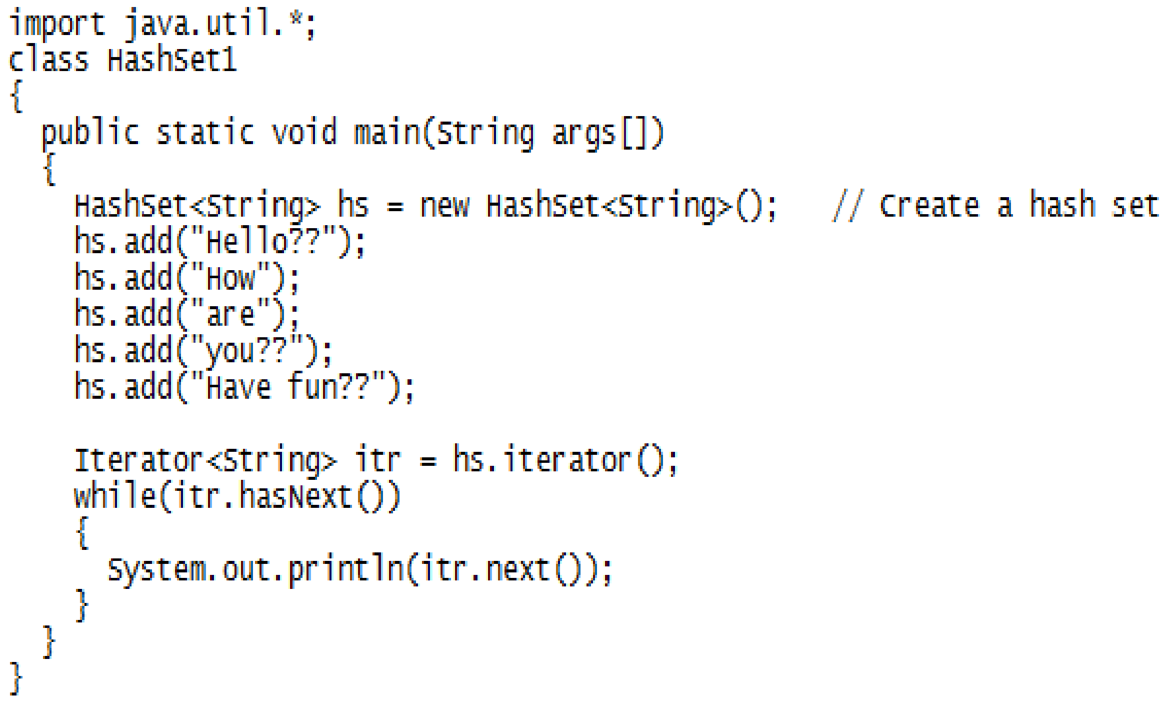
Collection Classes (Cont…)

- **HashSet Class:**
  - HashSet class extends AbstractSet, and it implements the Set interface.
  - It creates a collection that uses a hash table for storage. Declaration:

    class HashSet<E>
  - There are four constructors:
    - HashSet( ) //Default hash set
    - HashSet(Collection<? extends E> *c*) //Initializes by using element c
    - HashSet(int *capacity*) //Initializes the capacity (Default capacity is 16)
    - HashSet(int *capacity*, float *fillRatio*) //Initializes capacity and fill ratio (also called load capacity) from its arguments. The fill ratio must be between 0.0 and 1.0.

Collection Classes (Cont…)

- **HashSet Class (Cont…):**



```
import java.util.*;
class HashSet1
{
  public static void main(String args[])
  {
    HashSet<String> hs = new HashSet<String>();   // Create a hash set
    hs.add("Hello??");
    hs.add("How");
    hs.add("are");
    hs.add("you??");
    hs.add("Have fun??");

    Iterator<String> itr = hs.iterator();
    while(itr.hasNext())
    {
      System.out.println(itr.next());
    }
  }
}
```

Collection Classes (Cont…)

- **HashSet Class (Cont…):**
  - **Output**

```
How
you??
are
Have fun??
Hello??
```

Collection Classes (Cont…)

- **TreeSet Class:**
  - TreeSet extends AbstractSet and implements the NavigableSet interface. It creates a collection that uses a tree for storage.
  - Objects are stored in sorted, ascending order.
  - Access and retrieval times are quite fast. Declaration:

    class TreeSet<E>
  - There are four constructors:

    TreeSet( ) //Sorted tree with ascending order

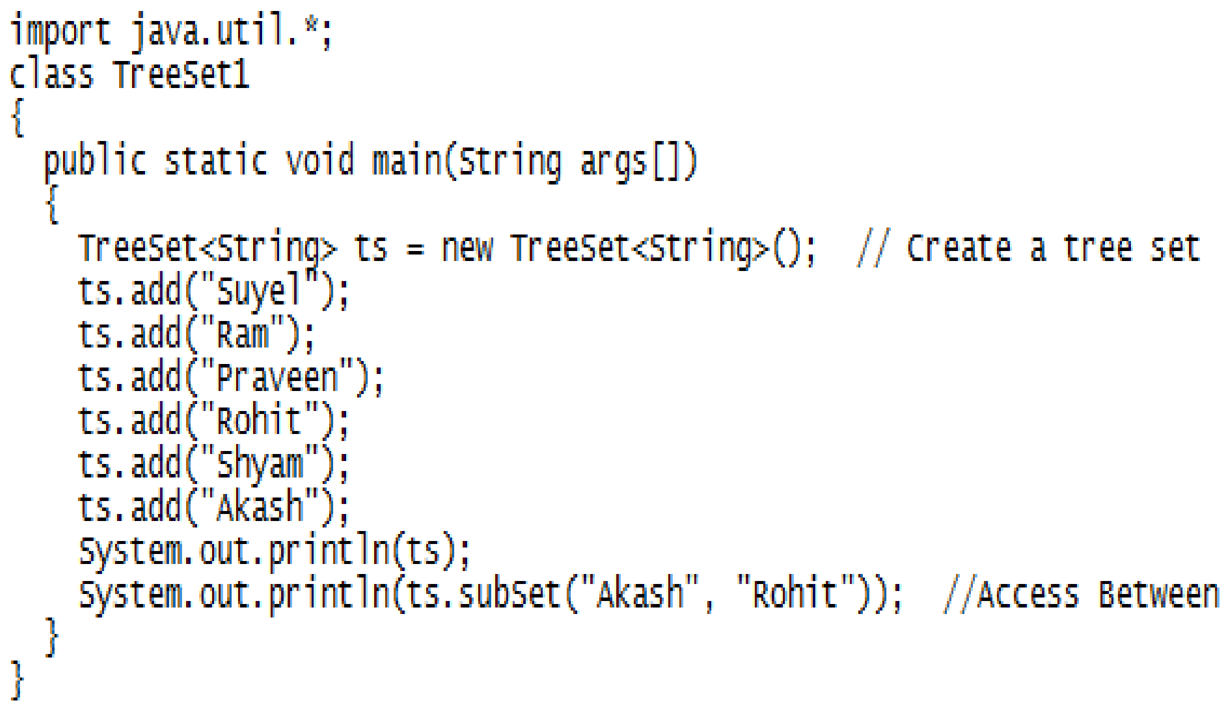
    TreeSet(Collection<? extends E> *c*) //Contains elements of *c*

    TreeSet(Comparator<? super E> *comp*) //Empty tree sorted according to the comparator

    TreeSet(SortedSet<E> *ss*) //A tree set that contains the elements of *ss*

Collection Classes (Cont…)

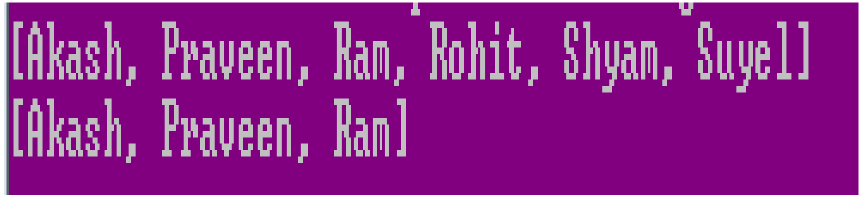
❑ TreeSet Class (Cont…):



```
import java.util.*;
class TreeSet1
{
  public static void main(String args[])
  {
    TreeSet<String> ts = new TreeSet<String>();  // Create a tree set
    ts.add("Suyel");
    ts.add("Ram");
    ts.add("Praveen");
    ts.add("Rohit");
    ts.add("Shyam");
    ts.add("Akash");
    System.out.println(ts);
    System.out.println(ts.subSet("Akash", "Rohit"));  //Access Between
  }
}
```

Collection Classes (Cont…)

- **TreeSet Class (Cont…):**
  - **Output**



```
[Akash, Praveen, Ram, Rohit, Shyam, Suyel]
[Akash, Praveen, Ram]
```

Collection Classes (Cont…)

- **PriorityQueue Class:**
  - PriorityQueue extends AbstractQueue and implements the Queue interface. PriorityQueue is a generic class that has the declaration:

    class PriorityQueue<E>
  - There are six constructors:

    PriorityQueue( ) //Builds an empty queue. Its starting capacity is 11.

    PriorityQueue(int *capacity*) //Builds a queue with specified *capacity*

    PriorityQueue(int *capacity*, Comparator<? super E> *comp*) //*Capacity* and *comparator*

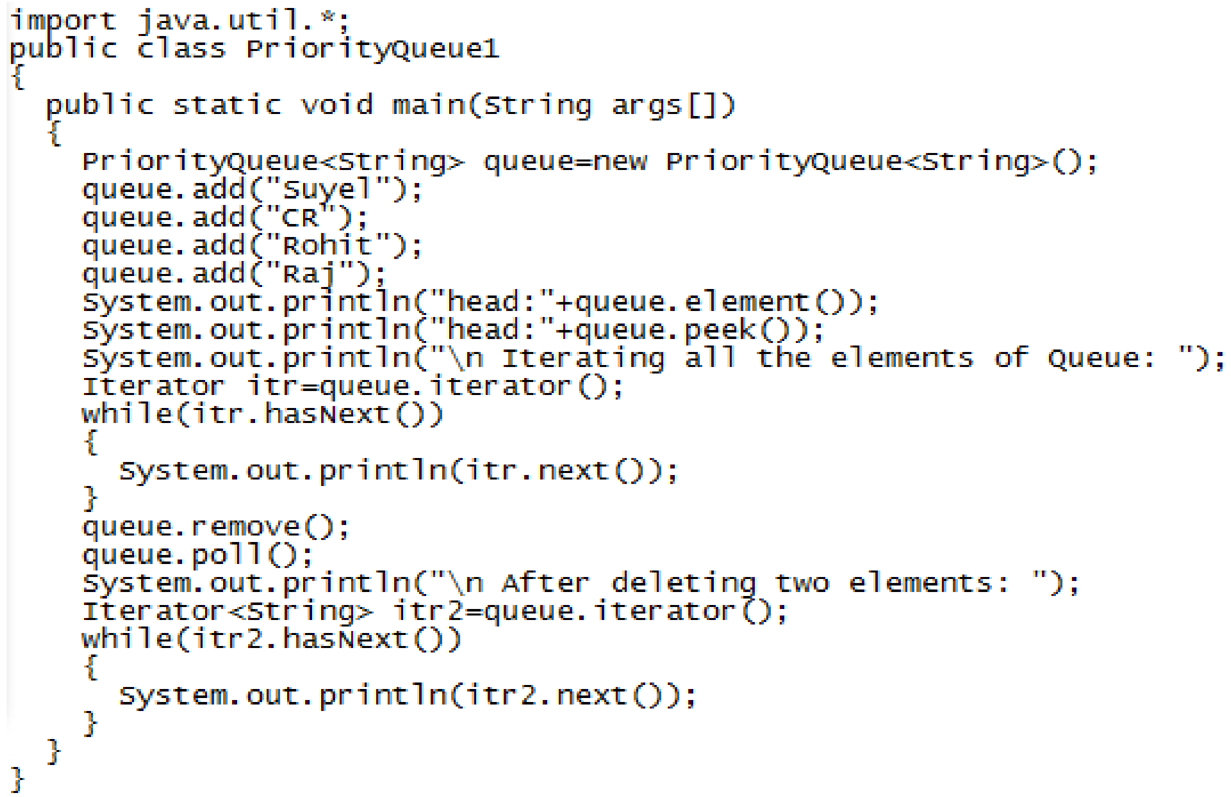
    PriorityQueue(Collection<? extends E> *c*) //Elements of the collection passes in *c*

    PriorityQueue(PriorityQueue<? extends E> *c*) //Elements of the collection passes in *c*

    PriorityQueue(SortedSet<? extends E> *c*) //Elements of the collection passes in *c*
  - It creates a queue that is prioritized based on the queue's comparator.

Collection Classes (Cont…)

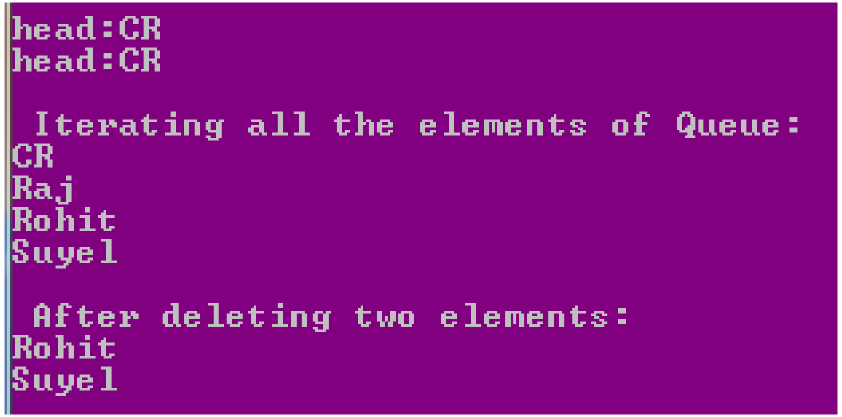
- **PriorityQueue Class (Cont…):**



```
import java.util.*;
public class PriorityQueue1
{
  public static void main(String args[])
  {
    PriorityQueue<String> queue=new PriorityQueue<String>();
    queue.add("Suyel");
    queue.add("CR");
    queue.add("Rohit");
    queue.add("Raj");
    System.out.println("head:"+queue.element());
    System.out.println("head:"+queue.peek());
    System.out.println("\n Iterating all the elements of Queue: ");
    Iterator itr=queue.iterator();
    while(itr.hasNext())
    {
      System.out.println(itr.next());
    }
    queue.remove();
    queue.poll();
    System.out.println("\n After deleting two elements: ");
    Iterator<String> itr2=queue.iterator();
    while(itr2.hasNext())
    {
      System.out.println(itr2.next());
    }
  }
}
```

Collection Classes (Cont…)

- **PriorityQueue Class (Cont…):**
  - **Output**



```
head:CR
head:CR

 Iterating all the elements of Queue:
CR
Raj
Rohit
Suyel

 After deleting two elements:
Rohit
Suyel
```

Collection Classes (Cont…)

- **ArrayDeque Class:**
  - ArrayDeque class extends AbstractCollection and implements the Deque interface.
  - It adds no methods of its own and creates a dynamic array with no capacity restrictions (Deque interface supports implementations that restrict capacity, but does not require such restrictions).
  - Declaration:

    class ArrayDeque<E>
  - There are three constructors:

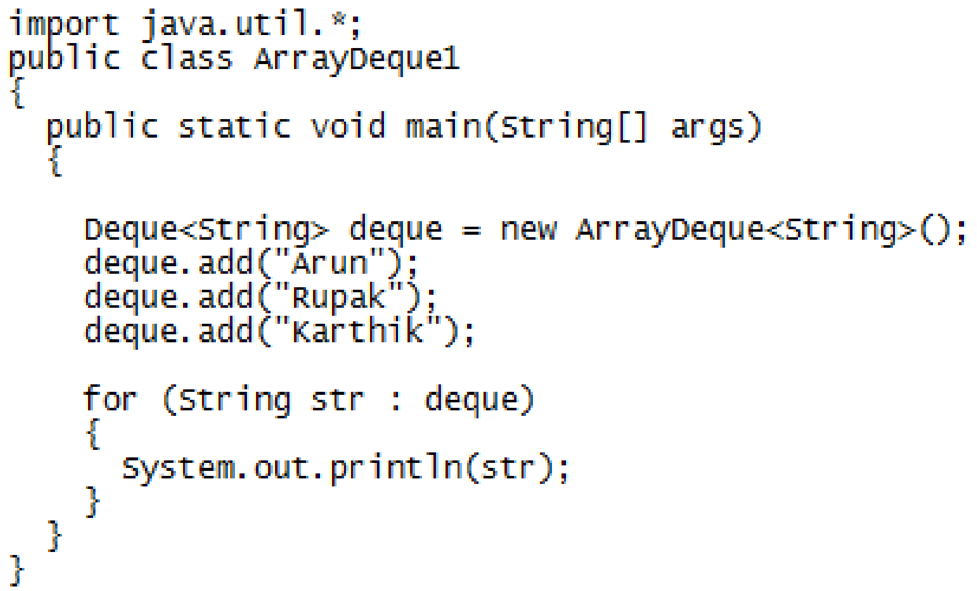
    ArrayDeque( ) //Builds an empty deque with starting capacity 16.

    ArrayDeque(int *size*) //Builds a deque with specified capacity

    ArrayDeque(Collection<? extends E> *c*) //Elements of the collection passes in *c*

Collection Classes (Cont…)

❑ **ArrayDeque Class (Cont…):**



```
import java.util.*;
public class ArrayDeque1
{
  public static void main(String[] args)
  {

    Deque<String> deque = new ArrayDeque<String>();
    deque.add("Arun");
    deque.add("Rupak");
    deque.add("Karthik");

    for (String str : deque)
    {
      System.out.println(str);
    }
  }
}
```

Collection Classes (Cont…)

- **ArrayDeque Class (Cont…):**
  - **Output**

```
Arun
Rupak
Karthik
```

Collection Classes (Cont…)

- **EnumSet Class:**
  - EnumSet is one of the specialized implementation of Set interface to use with the enumeration type.
  - It extends AbstractSet and implements Set Interface.
  - It is not synchronized and faster than HashSet.
  - This provides many static factory methods to create an instance of EnumSet e.g. EnumSet.of(...). Some of the methods are mentioned in the next slide.
  - Declaration:

    class EnumSet<E extends Enum<E>>

    Here, E specifies the elements and E must extend Enum<E>, which enforces the requirement that the elements must be of the specified enum type.

Collection Classes (Cont…)

Methods	**Descriptions**
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an EnumSet that contains the elements in the enumeration specified by *t*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Creates an EnumSet from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Creates an EnumSet from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Creates an EnumSet that contains *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Creates an EnumSet that contains *v1* and *v2*.

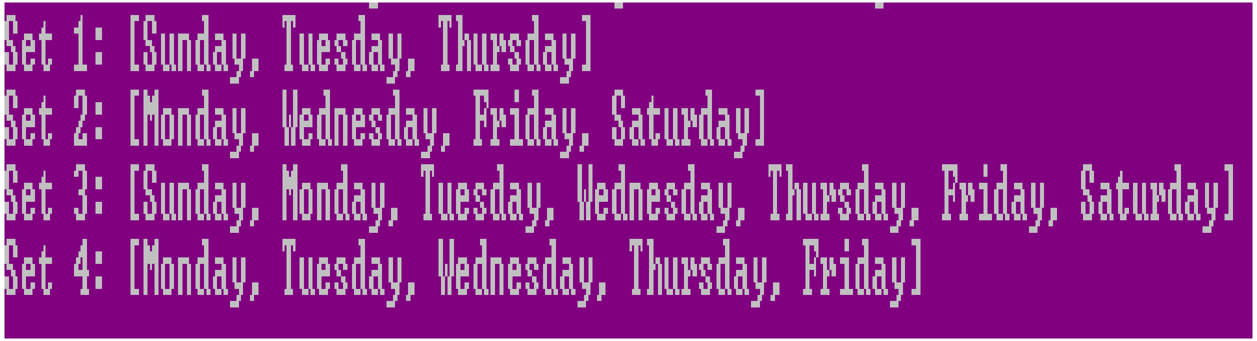
Collection Classes (Cont…)

❑ EnumSet Class (Cont…):

```
import java.util.*;

enum Days
{
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}
public class EnumSet1
{
  public static void main(String[] args)
  {
    EnumSet<Days> set1, set2, set3, set4;   //Creating a set

    set1 = EnumSet.of(Days.Sunday, Days.Tuesday, Days.Thursday);  //Adding elements
    set2 = EnumSet.complementOf(set1);
    set3 = EnumSet.allOf(Days.class);
    set4 = EnumSet.range(Days.Monday, Days.Friday);
    System.out.println("Set 1: " + set1);
    System.out.println("Set 2: " + set2);
    System.out.println("Set 3: " + set3);
    System.out.println("Set 4: " + set4);
  }
}
```

Collection Classes (Cont…)

- EnumSet Class (Cont…):
  - Output



```
Set 1: [Sunday, Tuesday, Thursday]
Set 2: [Monday, Wednesday, Friday, Saturday]
Set 3: [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]
Set 4: [Monday, Tuesday, Wednesday, Thursday, Friday]
```

Accessing a Collection via an Iterator

- Sometimes, we want to display the elements of a collection.
- One way to do so by employing an iterator.
  - It is an object that implements either the Iterator or ListIterator interface.
  - Iterator enables us to cycle through a collection, obtaining or removing elements.
  - ListIterator extends Iterator to allow bidirectional traversal of a list.
  - Both Iterator and ListIterator are generic interfaces, which are declared as:

    interface Iterator<E>

    interface ListIterator<E>
  - There are many methods of both Iterator and ListIterator.

Accessing a Collection via an Iterator (Cont…)

- **Methods of Iterator**

Methods	Descriptions
boolean hasNext( )	Returns true, if there are more elements. Otherwise, returns false.
E next( )	Returns the next element. Throws NoSuchElementException, if there is not a next element.
void remove( )	Removes the current element. Throws IllegalStateException, if an attempt is made to call remove( ) that is not preceded by a call to next( ).

Accessing a Collection via an Iterator (Cont…)

❑ Some Methods of ListIterator

Methods	Descriptions
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to next( ).
boolean hasNext( )	Returns true, if there is a next element. Otherwise, returns false.
boolean hasPrevious( )	Returns true, if there is a previous element.
E next( )	Returns the next element. A NoSuchElementException is thrown, if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A NoSuchElementException is thrown, if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns −1.

Accessing a Collection via an Iterator (Cont…)

- **Using an Iterator and ListIterator**
  - To use an iterator for the contents of a collection, there are these steps:

    1. Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.

    2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.

    3. Within the loop, obtain each element by calling next( ).
  - ListIterator is available only to those collections that implement the List interface.
  - ListIterator gives us the ability to access the collection in either the forward or backward direction and lets you modify an element.

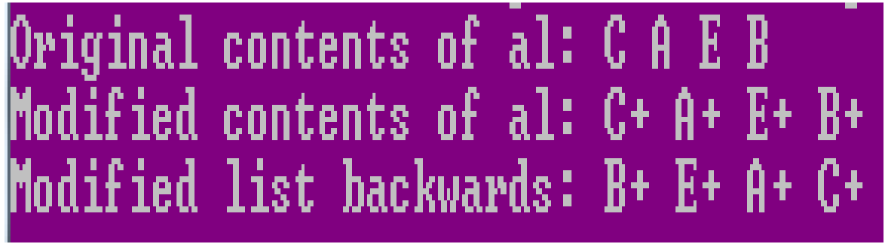
Accessing a Collection via an Iterator (Cont…)

- **Using an Iterator and ListIterator (Cont…)**

```
import java.util.*;
class ListIterator1
{
 public static void main(String args[])
 {
   ArrayList<String> al = new ArrayList<String>(); //Create an array list
   al.add("C");
   al.add("A");
   al.add("E");
   al.add("B");
   System.out.print("Original contents of al: ");  //Use iterator to display contents of al
   Iterator<String> itr = al.iterator();
   while(itr.hasNext())
   {
     String element = itr.next();
     System.out.print(element + " ");
   }
   System.out.println();
   ListIterator<String> litr = al.listIterator();  //Modify objects being iterated.
   while(litr.hasNext())
   {
     String element = litr.next();
     litr.set(element + "+");
   }
   System.out.print("Modified contents of al: ");
   itr = al.iterator();
   while(itr.hasNext())
   {
     String element = itr.next();
     System.out.print(element + " ");
   }
   System.out.println();
   System.out.print("Modified list backwards: ");  //Display the list backwards.
   while(litr.hasPrevious())
   {
     String element = litr.previous();
     System.out.print(element + " ");
   }
   System.out.println();
 }
}
```

Accessing a Collection via an Iterator (Cont…)

- Using an Iterator and ListIterator (Cont…)
  - Output



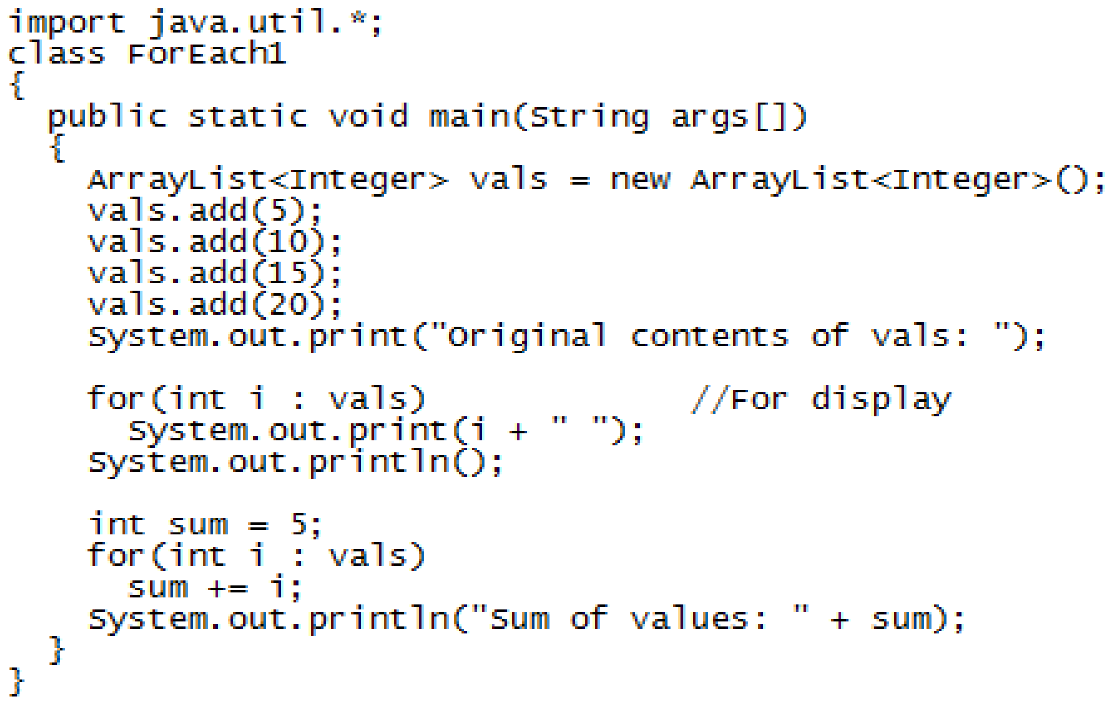
```
Original contents of al: C A E B
Modified contents of al: C+ A+ E+ B+
Modified list backwards: B+ E+ A+ C+
```

Accessing a Collection via an Iterator (Cont…)

- **For-Each Alternative to Iterators**
  - For each loop cannot operate in the backward direction.
  - All the Collection classes can be operated upon by the *for*.

Accessing a Collection via an Iterator (Cont…)

- **For-Each Alternative to Iterators (Cont…)**



```
import java.util.*;
class ForEach1
{
  public static void main(String args[])
  {
    ArrayList<Integer> vals = new ArrayList<Integer>();
    vals.add(5);
    vals.add(10);
    vals.add(15);
    vals.add(20);
    System.out.print("Original contents of vals: ");

    for(int i : vals)              //For display
      System.out.print(i + " ");
    System.out.println();

    int sum = 5;
    for(int i : vals)
      sum += i;
    System.out.println("Sum of values: " + sum);
  }
}
```

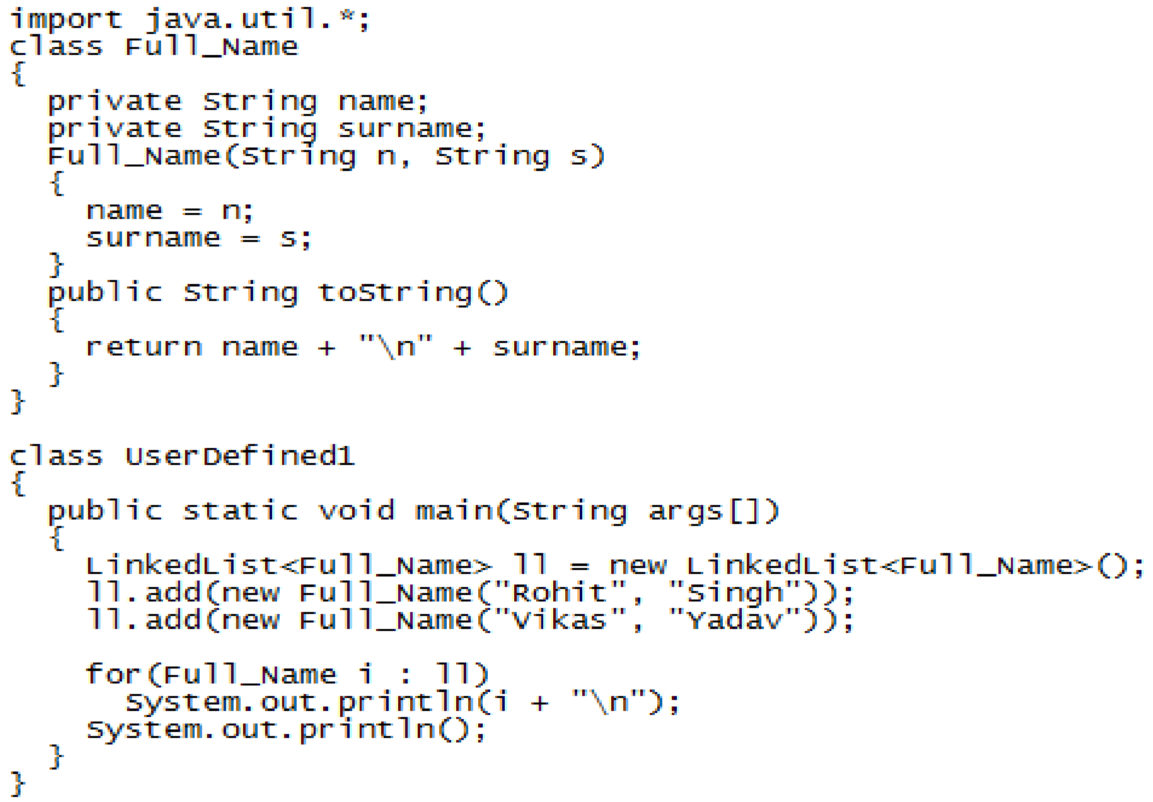
Accessing a Collection via an Iterator (Cont…)

- **For-Each Alternative to Iterators (Cont…)**
  - **Output**

```
Original contents of vals: 5 10 15 20
Sum of values: 55
```

User-Defined Classes in Collections

- We can create our own user defined classes in collection.



```
import java.util.*;
class Full_Name
{
  private String name;
  private String surname;
  Full_Name(String n, String s)
  {
    name = n;
    surname = s;
  }
  public String toString()
  {
    return name + "\n" + surname;
  }
}

class UserDefined1
{
  public static void main(String args[])
  {
    LinkedList<Full_Name> ll = new LinkedList<Full_Name>();
    ll.add(new Full_Name("Rohit", "Singh"));
    ll.add(new Full_Name("Vikas", "Yadav"));

    for(Full_Name i : ll)
      System.out.println(i + "\n");
    System.out.println();
  }
}
```

User-Defined Classes in Collections (Cont…)

- **Output**

```
Rohit
Singh


Vikas
Yadav
```

Working with Maps

- A map is an object that stores associations between keys and values or key/value pairs. Both keys and values are objects.
- The keys must be unique, but, the values may be duplicated.
- Maps don’t implement the Iterable interface (Collection interface), so we cannot cycle through a map using a for-each style for loop and cannot obtain an iterator to a map.
- We can obtain a collection-view of a map by entrySet( ), which allows the use of either for loop or an iterator.
- There are four interfaces of maps, namely **Map, Map.Entry NavigableMap and SortedMap.**

Map Interface

❑ Map Interface

- ❖ The Map interface maps unique keys to values.
- ❖ Given a key and a value, we can store the value in a Map object. After the value is stored, we can retrieve it by using its key.
- ❖ Map is generic and is declared as shown below:

  interface Map<K, V>

  Here, K specifies the type of keys, and V specifies the type of values.
- ❖ There are many methods declared by map. **Some** of them are mentioned in the next slide.
- ❖ Several methods throw a ClassCastException, when an object is incompatible with the elements in a map.

Map Interface (Cont…)

Methods	Descriptions
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object *k*)	Returns true, if the invoking map contains k as a key.
boolean containsValue(Object *v*)	Returns true, if the map contains *v* as a value.
Set<Map.Entry<K, V>> entrySet( )	Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. Thus, it provides a set-view.
V put(K *k*, V *v*)	Puts an entry in the invoking map. The key and value are *k* and *v*, respectively. Returns null, if the key did not already exist. Otherwise, the previous value linked to the key is returned.
V get(Object *k*)	Returns the value associated with the key *k*. Returns null, if the key is not found.

Map Interface (Cont…)

❑ SortedMap Interface

- ❖ SortedMap interface extends Map.
- ❖ It ensures that the entries are maintained in ascending order based on the keys.
- ❖ SortedMap is generic and is declared:

  interface SortedMap<K, V>
- ❖ **All** the methods of SortedMap are presented in the next slide.
- ❖ This interface allows very efficient manipulations of **submaps**, which are actually subsets of a map. To obtain a submap, we can use headMap( ), tailMap( ) or subMap( ) methods.

Map Interface (Cont…)

Methods	Descriptions
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, null is returned.
K firstKey( )	Returns the first key in the invoking map.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those entries with keys that are less than *end*.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

Map Interface (Cont…)

- **NavigableMap Interface**
  - NavigableMap interface extends SortedMap.
  - It declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
  - It is a generic interface that has the following declaration:

    interface NavigableMap<K,V>
  - It supports many additional methods. **Some** of them are shown in the next slide.
  - Several methods throw a ClassCastException, when an object is incompatible with the keys in the map.

Map Interface (Cont…)

Methods	Descriptions
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, null is returned.
NavigableSet<K> descendingKeySet( )	Returns a NavigableSet that contains the keys in the invoking map in reverse order. It returns a reverse set-view of the keys.
NavigableMap<K,V > descendingMap( )	Returns a NavigableMap that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, null is returned.

Map Interface (Cont…)

- **Map.Entry Interface**
  - The Map.Entry interface enables us to work with a map entry.
  - Please note that the entrySet( ) method declares by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.
  - Map.Entry is generic, and it is declared by:

    interface Map.Entry<K, V>
  - All the methods of this interface are mentioned in the next slide.
  - Many methods throw various exceptions.

Map Interface (Cont…)

Methods	Descriptions
boolean equals(Object *obj*)	Returns true, if *obj* is a Map.Entry, whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. ClassCastException is thrown, if *v* is not the correct type. IllegalArgumentException is thrown, if there is problem with *v*. NullPointerException is thrown, if *v* is null and the map does not permit null keys. An UnsupportedOperationException is thrown, if the map cannot be changed.

Map Class

- Several classes provide implementations of the map interfaces.
- There are many map classes, namely AbstractMap, EnumMap, HashMap, TreeMap, WeakHashMap, LinkedHashMap and IdentityHashMap.
- AbstractMap is a superclass for all concrete map implementations.
- WeakHashMap implements a map that uses “weak keys”. It allows an element in a map to be garbage-collected, when its key is unused.

Map Class (Cont…)

- **HashMap Class:**
  - HashMap class extends AbstractMap & implements the Map interface.
  - It uses a hash table to store the map. Declaration is given below:

    class HashMap<K, V>
  - There are four constructors as mentioned below:

    HashMap( ) //Default hash map

    HashMap(Map<? extends K, ? extends V> *m*) //Initializes with elements of m

    HashMap(int *capacity*) //Initializes with capacity

    HashMap(int *capacity*, float *fillRatio*) //Initializes both capacity and fill ratio. Default capacity and fill ration is 16 and 0.75, respectively.
  - This class does not have its own method.

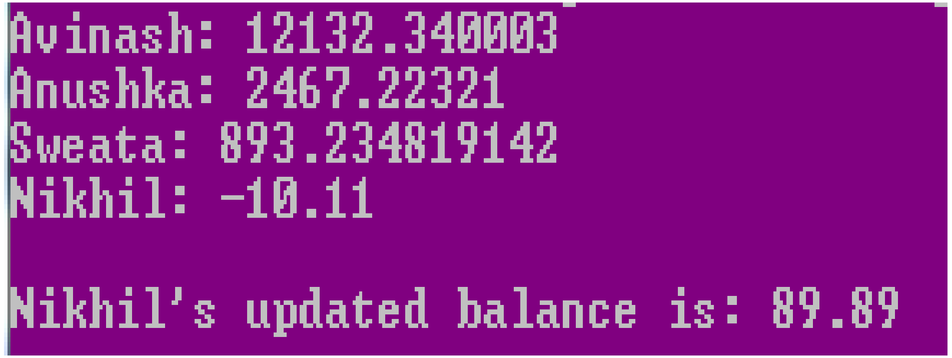
Map Class (Cont…)

- **HashMap Class (Cont…):**

```
import java.util.*;
class HashMap1
{
  public static void main(String args[])
  {
    HashMap<String, Double> hhmp = new HashMap<String, Double>();
    hhmp.put("Avinash", new Double(12132.340003));
    hhmp.put("Anushka", new Double(2467.22321));
    hhmp.put("Sweata", new Double(893.234819142));
    hhmp.put("Nikhil", new Double(-10.11));

    Set<Map.Entry<String, Double>> set = hhmp.entrySet();    //Get a set of the entries

    for(Map.Entry<String, Double> i : set)  //Display the set
    {
      System.out.print(i.getKey() + ": ");
      System.out.println(i.getValue());
    }
    System.out.println();

    double balance = hhmp.get("Nikhil");    //Deposit 100 into Nikhil
    hhmp.put("Nikhil", balance + 100);
    System.out.println("Nikhil's updated balance is: " + hhmp.get("Nikhil"));
  }
}
```

Map Class (Cont…)

- HashMap Class (Cont…):
  - Output



```
Avinash: 12132.340003
Anushka: 2467.22321
Sweata: 893.234819142
Nikhil: -10.11

Nikhil's updated balance is: 89.89
```

Map Class (Cont…)

- **TreeMap Class:**
  - TreeMap class extends AbstractMap and implements the NavigableMap interface.
  - TreeMap guarantees that its elements are sorted in the ascending key order. It is a generic class that has the below declaration:

    class TreeMap<K, V>
  - There are four constructors:

    TreeMap( )   //Construct an empty tree map (sorted by natural order of its key)

    TreeMap(Comparator<? super K> *comp*)   //Empty tree sorted by comparator

    TreeMap(Map<? extends K, ? extends V> *m*) //Initializes with the entries of *m* (sorted by using natural order)

    TreeMap(SortedMap<K, ? extends V> *sm*)   //Initializes with the entries of *sm* (sorted in the same order as *sm*).

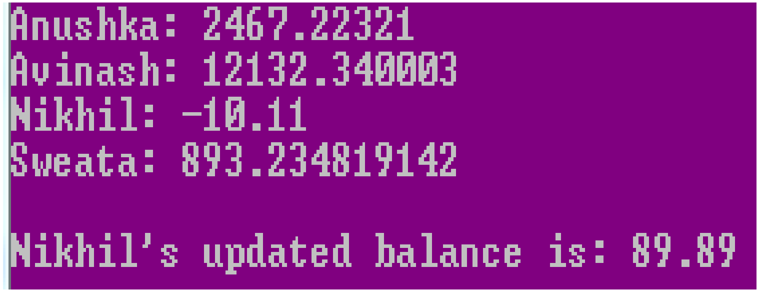
Map Class (Cont…)

- **TreeMap Class (Cont…):**

```
import java.util.*;
class TreeMap1
{
  public static void main(String args[])
  {
    TreeMap<String, Double> Trmp = new TreeMap<String, Double>();
    Trmp.put("Avinash", new Double(12132.340003));
    Trmp.put("Anushka", new Double(2467.22321));
    Trmp.put("Sweata", new Double(893.234819142));
    Trmp.put("Nikhil", new Double(-10.11));

    Set<Map.Entry<String, Double>> set = Trmp.entrySet();    //Get a set of the entries

    for(Map.Entry<String, Double> i : set)  //Display the set
    {
      System.out.print(i.getKey() + ": ");
      System.out.println(i.getValue());
    }
    System.out.println();

    double balance = Trmp.get("Nikhil");    //Deposit 100 into Nikhil
    Trmp.put("Nikhil", balance + 100);
    System.out.println("Nikhil's updated balance is: " + Trmp.get("Nikhil"));
  }
}
```

Map Class (Cont…)

- **TreeMap Class (Cont…):**
  - **Output**



```
Anushka: 2467.22321
Avinash: 12132.340003
Nikhil: -10.11
Sweata: 893.234819142

Nikhil's updated balance is: 89.89
```

Map Class (Cont…)

- **LinkedHashMap Class:**
  - LinkedHashMap class extends HashMap. It maintains a linked list of the entries in the map in the order in which they were inserted.
  - Here, the elements are returned in the order in which they are inserted.
  - We can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.
  - LinkedHashMap is a generic class that has the following declaration:

    class LinkedHashMap<K, V>
  - It adds only one method to those defined by HashMap:

    protected boolean removeEldestEntry(Map.Entry<K, V> *e*) //This method is called by put( ) and putAll( ). The oldest entry is passed in *e*. By default, it returns false and does nothing. However, if we override this method, then we can have the LinkedHashMap that removes the oldest entry in the map.

Map Class (Cont…)

- **LinkedHashMap Class (Cont…):**
  - There are five consrtuctors:

LinkedHashMap( ) //Constructs a default LinkedHashMap

LinkedHashMap(Map<? extends K, ? extends V> *m*) //Initializes with the elements of m

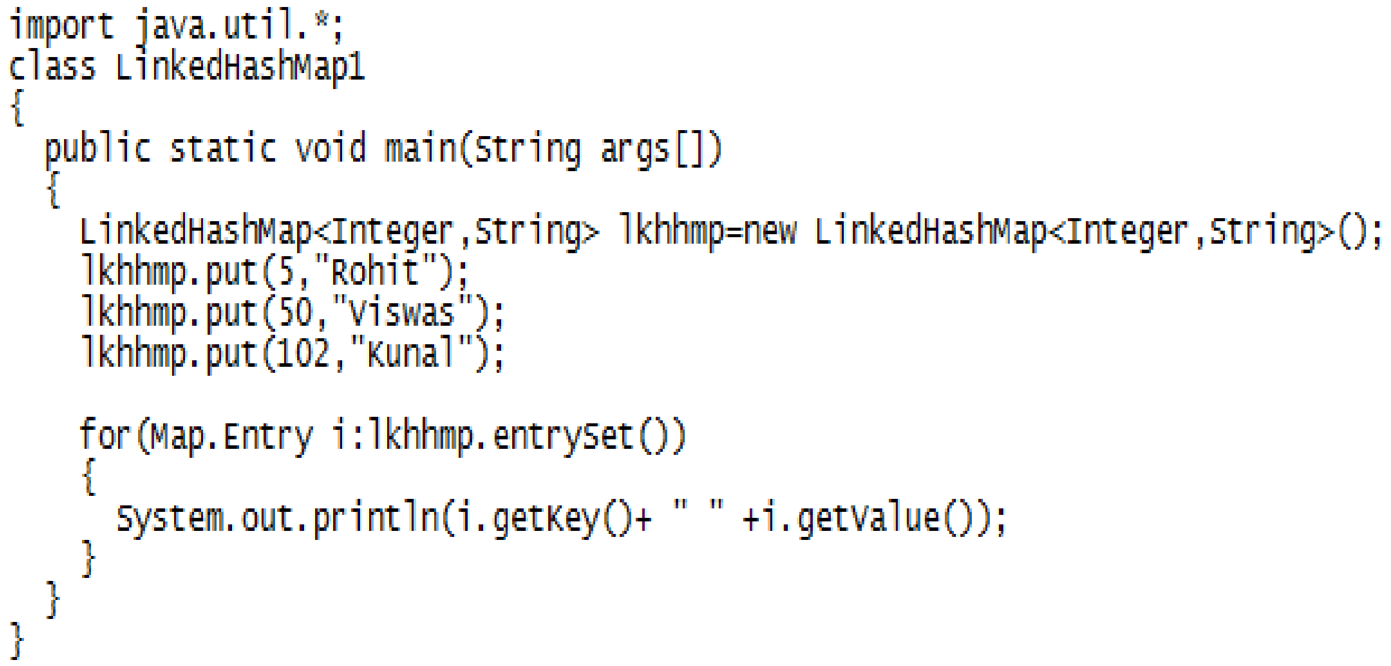
LinkedHashMap(int *capacity*) //Initializes with *capacity*

LinkedHashMap(int *capacity*, float *fillRatio*) //Initializes with *capacity* and *fill ratio*. Default capacity is 16 and fill ration is 0.75

LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*) //It allows to specify whether the elements will be stored in the linked list by insertion order or by order of last access. If *Order* is true, then access order is used. If *Order* is false, then insertion order is used.

Map Class (Cont…)

- **LinkedHashMap Class (Cont…):**



```
import java.util.*;
class LinkedHashMap1
{
  public static void main(String args[])
  {
    LinkedHashMap<Integer,String> lkhhmp=new LinkedHashMap<Integer,String>();
    lkhhmp.put(5,"Rohit");
    lkhhmp.put(50,"Viswas");
    lkhhmp.put(102,"Kunal");

    for(Map.Entry i:lkhhmp.entrySet())
    {
      System.out.println(i.getKey()+ " " +i.getValue());
    }
  }
}
```

Map Class (Cont…)

- **LinkedHashMap Class (Cont…):**
  - **Output**

```
5 Rohit
50 Viswas
102 Kunal
```

Map Class (Cont…)

- **IdentityHashMap Class:**
  - IdentityHashMap extends AbstractMap and implements the Map interface.
  - It is similar to HashMap except that it uses reference equality, when comparing elements.
  - IdentityHashMap is a generic class that has the following declaration:

    class IdentityHashMap<K, V>

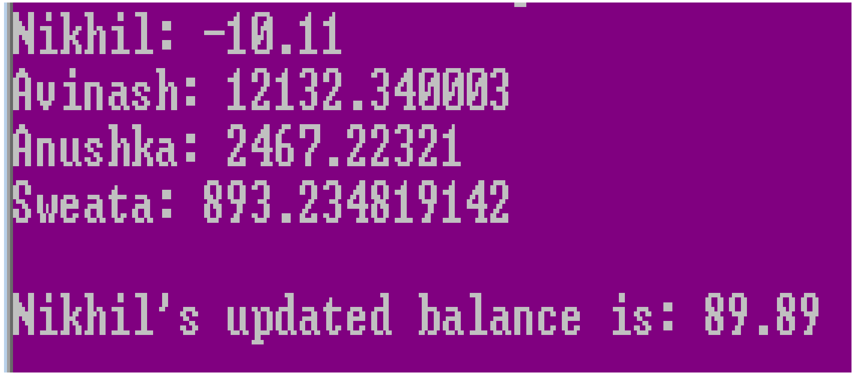
Map Class (Cont…)

- **IdentityHashMap Class (Cont…):**

```
import java.util.*;
class IdentityHashMap1
{
  public static void main(String args[])
  {
    IdentityHashMap IdHhMp = new IdentityHashMap();
    IdHhMp.put("Avinash", new Double(12132.340003));
    IdHhMp.put("Anushka", new Double(2467.22321));
    IdHhMp.put("Sweata", new Double(893.234819142));
    IdHhMp.put("Nikhil", new Double(-10.11));

    Set set = IdHhMp.entrySet();     //Get a set of the entries

    Iterator i = set.iterator();     //Get an iterator

    while(i.hasNext())  //Display the set
    {
      Map.Entry mpen = (Map.Entry)i.next();
      System.out.print(mpen.getKey() + ": ");
      System.out.println(mpen.getValue());
    }
    System.out.println();

    double balance = ((Double)IdHhMp.get("Nikhil")).doubleValue();    //Deposit 100 into Nikhil
    IdHhMp.put("Nikhil", new Double (balance + 100));
    System.out.println("Nikhil's updated balance is: " + IdHhMp.get("Nikhil"));
  }
}
```

Map Class (Cont…)

- IdentityHashMap Class (Cont…):
  - Output



```
Nikhil: -10.11
Avinash: 12132.340003
Anushka: 2467.22321
Sweata: 893.234819142

Nikhil's updated balance is: 89.89
```

Map Class (Cont…)

- **EnumMap Class:**
  - EnumMap extends AbstractMap and implements Map.
  - It is specifically for use with keys of an enum type. Declaration:

    class EnumMap<K extends Enum<K>, V>

    Here, K must extend Enum<K> that enforces the requirement that the keys must be of an enum type.
  - There are three constructors:

    EnumMap(Class<K> *kType*) //Constructor an empty EnumMap of type kType

    EnumMap(Map<K, ? extends V> *m*) //Creates an EnumMap from entries of *m*

    EnumMap(EnumMap<K, ? extends V> *em*) //Creates an EnumMap initialized with the values in *em*.
  - EnumMap defines no methods of its own.

Comparator

- Comparator supports us to order the elements of **Set or Map** based on our requirement or choice.
- Comparator is a generic interface that has this declaration:

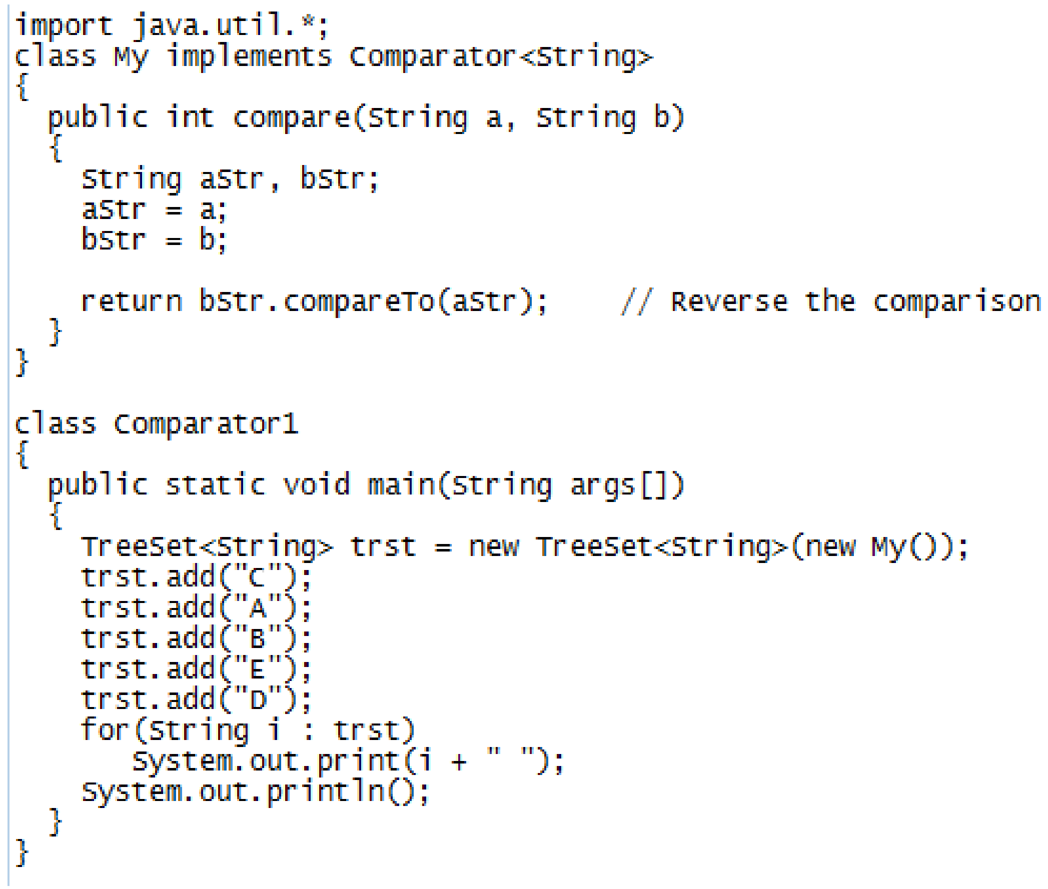
  interface Comparator<T>

  Here, T specifies the type of objects being compared.
- There are two methods, namely compare( ) and equals( ):

  int compare(T *obj1*, T *obj2*) //Returns a positive value, when *obj1* is greater than *obj2*. Otherwise, negative value is returned.

  boolean equals(Object *obj*) //It is used for checking equality. It returns true, if *obj* and the invoking object both are Comparator objects and use the same ordering. Otherwise, it returns false.
- By overriding compare( ), we can alter the order of the objects.

Comparator (Cont…)



```
import java.util.*;
class My implements Comparator<String>
{
  public int compare(String a, String b)
  {
    String aStr, bStr;
    aStr = a;
    bStr = b;

    return bStr.compareTo(aStr);    // Reverse the comparison
  }
}

class Comparator1
{
  public static void main(String args[])
  {
    TreeSet<String> trst = new TreeSet<String>(new My());
    trst.add("C");
    trst.add("A");
    trst.add("B");
    trst.add("E");
    trst.add("D");
    for(String i : trst)
       System.out.print(i + " ");
    System.out.println();
  }
}
```

Comparator (Cont…)

- **Output**

```
E D C B A
```

Collection Algorithms

- The Collections Framework defines many algorithms, which can be applied to collections and maps.
- All the algorithms are defined as static methods within the Collections class.
- In the next three slides, some of these methods are mentioned.
- Several of the methods can throw **ClassCastException**.
- checkedCollection( ) method return the API documentation refers to as a “dynamically typesafe view” of a collection.
- This view is a reference to the collection that monitors insertions into the collection for type compatibility at run time.

Collection Algorithms (Cont…)

Method	Description
static <T> boolean addAll (Collection <? super T> *c*, T ... *elements*)	Inserts the elements specified by elements into the collection specified by *c*. Returns true, if the elements are added. Otherwise, false.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*.
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for value in list ordered according to *c*. Returns the position of value in list or a negative value, if value is not found.
static <T> int binarySearch(List<? Extends Comparable<? super T>> *list*, T *value*)	Searches for value in list. The list must be sorted. Returns the position of value in list or a negative value, if value is not found.

Collection Algorithms (Cont…)

Method	Description
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a List. An attempt to insert an incompatible element is caused a ClassCastException.
static <E> List<E> checkedSet(Set<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a Set. An attempt to insert an incompatible element is caused a ClassCastException.
static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*)	Copies the elements of *list2* to *list1*.
static <T> List<T> emptyList( )	Returns an immutable, empty List object of the inferred type.
static <T> List<T> nCopies(int *num*, T *obj*)	Returns *num* copies of *obj* contained in an immutable list. num>=0.

Collection Algorithms (Cont…)

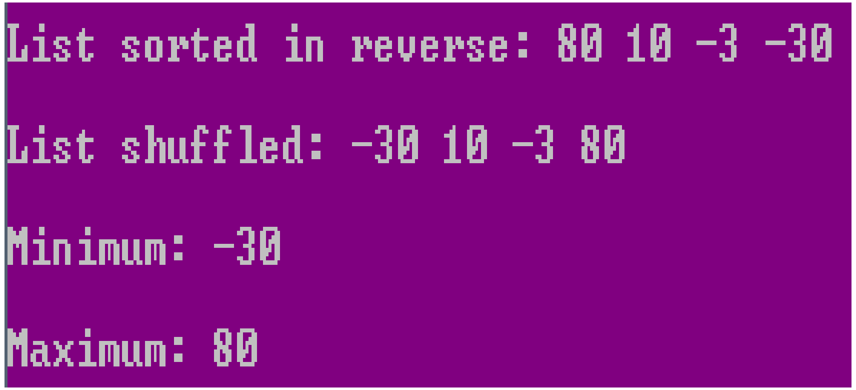
Method	Description
static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the maximum element in *c* as determined by *comp*.
static <T> Comparator<T> reverseOrder(Comparator<T> *comp*)	Returns a reverse comparator based on the one passed in *comp*. The returned comparator reverses the outcome of a comparison that uses *comp*.
static <K, V> Map<K, V> singletonMap(K *k*, V *v*)	Returns the key/value pair k/v as an immutable map. This is an easy way to convert a single key/value pair into a map.
static int indexOfSubList(List<?> *list*, List<?> *subList*)	Searches list for the first occurrence of *subList*. Returns the index of the first match or –1, if no match is found.

Collection Algorithms (Cont…)

```
import java.util.*;
class Algorithm1
{
  public static void main(String args[])
  {
    LinkedList<Integer> ll = new LinkedList<Integer>();
    ll.add(-3);
    ll.add(10);
    ll.add(-30);
    ll.add(80);

    Comparator<Integer> r = Collections.reverseOrder(); //Create a reverse order comparator

    Collections.sort(ll, r);  //Sort list by using the comparator

    System.out.print("\nList sorted in reverse: ");
    for(int i : ll)
      System.out.print(i+ " ");
    System.out.println();

    Collections.shuffle(ll);   //Shuffle list

    System.out.print("\nList shuffled: ");   //Display randomized list

    for(int i : ll)
      System.out.print(i + " ");
    System.out.println();

    System.out.println("\nMinimum: " + Collections.min(ll));
    System.out.println("\nMaximum: " + Collections.max(ll));
  }
}
```

Collection Algorithms (Cont…)

- **Output**



```
List sorted in reverse: 80 10 -3 -30

List shuffled: -30 10 -3 80

Minimum: -30

Maximum: 80
```

Arrays

- Arrays class provides various methods, which are useful to work with arrays.

- These methods help bridge the gap between collections and arrays.

- **asList( ) Method**
  - This method returns a **List,** supported by a specified array.
  - Here, both list and the array refer to the same location.

static <T> List asList(T ... *array*)   //array contains the data

Arrays (Cont…)

- **binarySearch( ) Method**
  - This method uses a binary search to find a specified value.
  - It must be applied to sorted arrays.
  - **Some** of its forms are:

    static int binarySearch(byte *array*[ ], byte *value*)

    static int binarySearch(char *array*[ ], char *value*)

    static int binarySearch(int *array*[ ], int *value*)

    static int binarySearch(long *array*[ ], long *value*)

    static int binarySearch(Object *array*[ ], Object *value*)

    static <T> int binarySearch(T[ ] *array*, T *value*, Comparator<? super T> *c*)
  - Here, *array* is the array to be searched, and *value* is the value to be located.

Arrays (Cont…)

copyOf( ) Method

- It returns a copy of an array**. Some versions** of this method are:

  static boolean[ ] copyOf(boolean[ ] *source*, int *len*)
  static double[ ] copyOf(double[ ] *source*, int *len*)
  static int[ ] copyOf(int[ ] *source*, int *len*)
  static <T> T[ ] copyOf(T[ ] *source*, int *len*)
  static <T, U> T[ ] copyOf(U[ ] *source*, int *len*, Class<? extends T[ ]> *resultT*)

- The original array is specified by *source*, and the length of the copy is specified by *len*.
- If the copy is longer than *source*, the copy is padded with zeros (for numeric arrays), nulls (for object arrays) or false (for boolean arrays). If the copy is shorter than *source*, then the copy is truncated.
- In the last form, type of *resultT* becomes type of the array returned.

Arrays (Cont…)

- **copyOfRange( ) Method**
  - This method returns a copy of a range within an array. **Some forms**:

    static boolean[ ] copyOfRange(boolean[ ] *source*, int *start*, int *end*)
    static char[ ] copyOfRange(char[ ] *source*, int *start*, int *end*)
    static float[ ] copyOfRange(float[ ] *source*, int *start*, int *end*)
    static int[ ] copyOfRange(int[ ] *source*, int *start*, int *end*)
    static <T,U> T[ ] copyOfRange(U[ ] *source*, int *start*, int *end*, Class<? extends T[ ]> *resultT*)
  - The original array is specified by *source*. The range to copy is specified by the indices passed via *start* and *end*.
  - The range runs from *start* to *end* – 1.
  - If the range is longer than *source*, the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays).

Arrays (Cont…)

- **equals( ) Method**
  - It returns true, if two arrays are equivalent. Otherwise, returns **false**.

    static boolean equals(boolean *array1*[ ], boolean *array2*[ ])
    static boolean equals(char *array1*[ ], char *array2*[ ])
    static boolean equals(float *array1*[ ], float *array2*[ ])
    static boolean equals(long *array1*[ ], long *array2*[ ])
    static boolean equals(Object *array1*[ ], Object *array2*[ ]) **//Some Forms**

- **deepEquals( ) Method**
  - It can be used to determine if two arrays, which might contain nested arrays, are equal. It has only one declaration: static boolean deepEquals(Object[ ] *a*, Object[ ] *b*)
  - It returns true, if the arrays passed in *a* and *b* contain the same elements. If *a* and *b* contain nested arrays, then the contents of those nested arrays are also checked. Otherwise, false.

Arrays (Cont…)

- **fill( ) Method:**
  - It assigns a value to all elements in an array.
  - fill( ) method has **two versions**. **Some** forms of the first version are:

    static void fill(byte *array*[ ], byte *value*)
    static void fill(int *array*[ ], int *value*)
    static void fill(Object *array*[ ], Object *value*)

    Here, *value* is assigned to all elements in *array*.
  - Some of the forms of the second version are:

    static void fill(boolean *array*[ ], int *start*, int *end*, boolean *value*)
    static void fill(char *array*[ ], int *start*, int *end*, char *value*)

    Here, *value* is assigned to the elements in *array* from position *start* to position *end*– 1.

Arrays (Cont…)

- **sort( ) Method**
  - It sorts an array in ascending order.
  - This method has **two versions**. **Some** forms of the first version are:

    static void sort(byte *array*[ ])
    static void sort(char *array*[ ])
    static void sort(int *array*[ ])
    static <T> void sort(T *array*[ ], Comparator<? super T> *c*)

    In the last form, *c* is a Comparator used to order the elements of *array*.
  - **Some** of the second version of sort( ) are:

    static void sort(char *array*[ ], int *start*, int *end*)
    static void sort(Object *array*[ ], int *start*, int *end*)

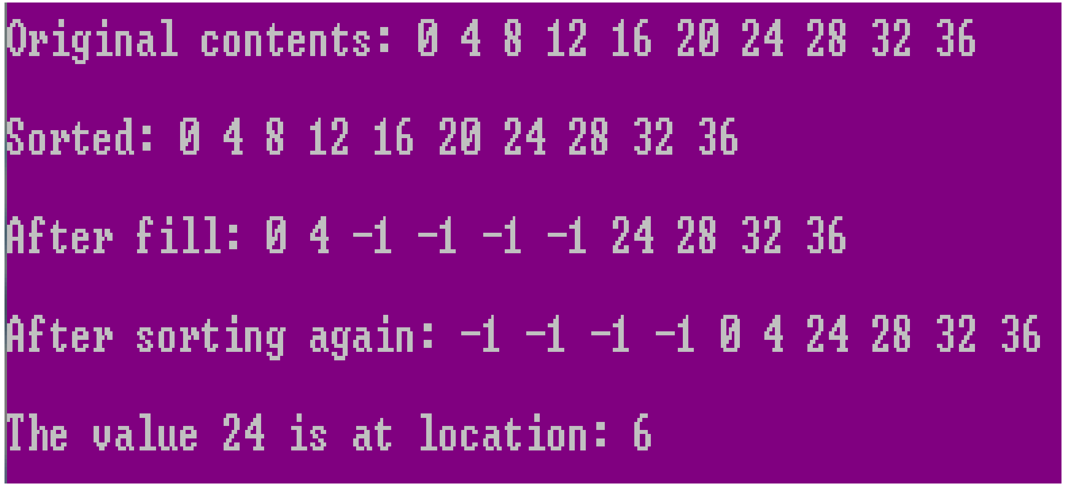
    Here, the range beginning at *start* and running through *end*– 1 within *array* will be sorted. All of these methods can throw exceptions

Arrays (Cont…)

```
import java.util.*;
class Arrays1
{
  public static void main(String args[])
  {

    int array[] = new int[10];    //Allocate and initialize array
    for(int i = 0; i < 10; i++)
      array[i] = 4 * i;

    System.out.print("\nOriginal contents: ");   //Display, sort, and display the array
    display(array);
    Arrays.sort(array);
    System.out.print("\nSorted: ");

    display(array);
    Arrays.fill(array, 2, 6, -1);    //Fill and display the array
    System.out.print("\nAfter fill: ");

    display(array);
    Arrays.sort(array);     //Sort and display the array
    System.out.print("\nAfter sorting again: ");

    display(array);
    System.out.print("\nThe value 24 is at location: ");   //Binary search for 24
    int index = Arrays.binarySearch(array, 24);
    System.out.println(index);
  }
  static void display(int array[])
  {
    for(int i: array)
      System.out.print(i + " ");
    System.out.println();
  }
}
```

Arrays (Cont…)

- **Output**



```
Original contents: 0 4 8 12 16 20 24 28 32 36

Sorted: 0 4 8 12 16 20 24 28 32 36

After fill: 0 4 -1 -1 -1 -1 24 28 32 36

After sorting again: -1 -1 -1 -1 0 4 24 28 32 36

The value 24 is at location: 6
```

Generic Collections

- Collections Framework is arguably the single most important use of generics in the Java API.
- The reason for this is that generics add type safety to the Collections Framework.
- Prior to generics, all collections stored references of type object. This allowed any type of reference to be stored in the collection.
- Unfortunately, the fact that a pre-generics collection stored object references could easily lead to errors.
- First, it requires that rather than the compiler, ensure that only objects of the proper type be stored in a specific collection.

Generic Collections (Cont…)

- The second problem with pre-generics collections is that when we retrieve a reference from the collection, we must manually cast that reference into the proper type.
- The addition of generics fundamentally improves the usability and safety of collections because:
  - Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection always contains references of a known type.
  - Eliminates the need to cast a reference retrieved from a collection. Instead, a reference retrieved from a collection is automatically cast into the proper type. This prevents run-time errors due to invalid casts and avoids an entire category of errors.

Legacy Classes and Interfaces

- None of the collection classes are synchronized. However, we can make them synchronized by using some algorithms provided by Collections.

- However, all the legacy classes are synchronized.

- There are one Legacy interface, namely Enumeration.

- Five Legacy classes are there, namely Dictionary, Hashtable, Properties, Stack and Vector

Legacy Classes and Interfaces (Cont…)

❑ **Enumeration Interface:**

- Enumeration interface defines the methods, so that we can enumerate (obtain one at a time) the elements in a collection of objects.
- It is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (Vector and Properties), and used by several other API classes. It has the below declaration: interface Enumeration<E>

  Here, E specifies the type of element being enumerated.
- There are two methods:

  boolean hasMoreElements( ) //Must return true while there are still more elements to extract, and false when all the elements have been enumerated.

  E nextElement( ) //It returns the next object in the enumeration. That is, each call to nextElement( ) obtains the next object in the enumeration.

Legacy Classes and Interfaces (Cont…)

- **Vector Class:**
  - Vector class is similar to ArrayList, but with two differences: (1) Vector is synchronized, and (2) it contains many legacy methods that are not part of the Framework.
  - It extends AbstractList and implements the List interface and Iterable.
  - Vector is fully compatible with collections. Declaration:

    class Vector<E>   //E specifies the type of element that is stored
  - There are four constructors:

    Vector( )   //Creates a default Vector with initial size 10.
    Vector(int *size*)   //Creates a Vector with initial capacity *size*.
    Vector(int *size*, int *incr*)   //Initializes with capacity and a increment. Increment is the number of elements to allocate each time that a vector is resized.
    Vector(Collection<? extends E> *c*)   //Initializes the elements of collection *c*.

Legacy Classes and Interfaces (Cont…)

- **Vector Class (Cont…):**
  - Vector defines these protected data members:

    int capacityIncrement; //Increment value is stored in capacityIncrement

    int elementCount; //Number of element currently in the Vector is stored in elementCount

    Object[ ] elementData; //The array that holds the Vector is stored in elementData.

  - There are many legacy methods in Vector. **Some** of the methods are mentioned in the next slide.

Legacy Classes and Interfaces (Cont…)

Method	Description
void addElement(E *element*)	The object specified by element is added to the vector.
int capacity( )	Returns the capacity of the vector.
E elementAt(int *index*)	Returns the element at the location specified by index.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to size.
E firstElement( )	Returns the first element in the vector.
E lastElement( )	Returns the last element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of element. If the object is not in the vector, –1 is returned.

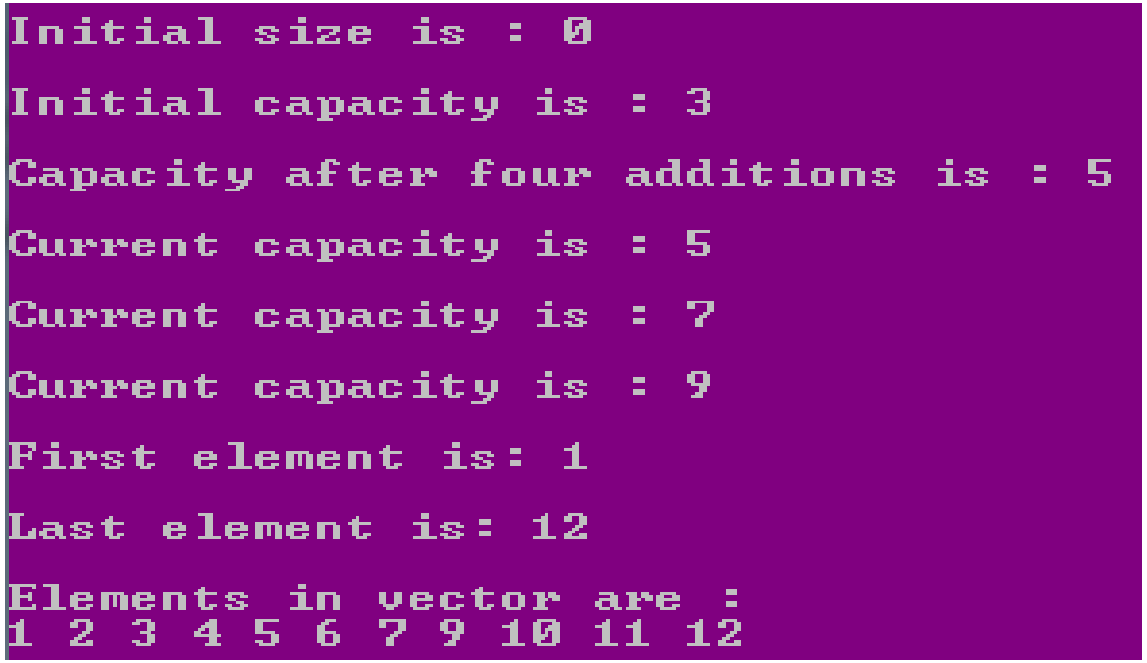
Legacy Classes and Interfaces (Cont…)

❑ Vector Class (Cont…):

```
import java.util.*;
class Vector1
{
  public static void main(String args[])
  {
    Vector<Integer> v = new Vector<Integer>(3, 2); //Initial size is 3 & increment is 2
    System.out.println("\nInitial size is : " + v.size());
    System.out.println("\nInitial capacity is : " + v.capacity());

    v.addElement(1);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);
    System.out.println("\nCapacity after four additions is : " + v.capacity());

    v.addElement(5);
    System.out.println("\nCurrent capacity is : " + v.capacity());

    v.addElement(6);
    v.addElement(7);
    System.out.println("\nCurrent capacity is : " + v.capacity());

    v.addElement(9);
    v.addElement(10);
    System.out.println("\nCurrent capacity is : " + v.capacity());

    v.addElement(11);
    v.addElement(12);
    System.out.println("\nFirst element is: " + v.firstElement());
    System.out.println("\nLast element is: " + v.lastElement());

    Enumeration vEnum = v.elements();   //Enumerate the elements in the vector
    System.out.println("\nElements in vector are : ");   //We can use Iterator or for-each For Loop
    while(vEnum.hasMoreElements())
       System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

Legacy Classes and Interfaces (Cont…)

- **Vector Class (Cont…):**



```
Initial size is : 0
Initial capacity is : 3
Capacity after four additions is : 5
Current capacity is : 5
Current capacity is : 7
Current capacity is : 9
First element is: 1
Last element is: 12
Elements in vector are :
1 2 3 4 5 6 7 9 10 11 12
```

Legacy Classes and Interfaces (Cont…)

- **Stack:**
  - Stack is a subclass of Vector that implements a standard last-in, first-out stack.
  - Stack only defines the default constructor that creates an empty stack.

    class Stack<E>

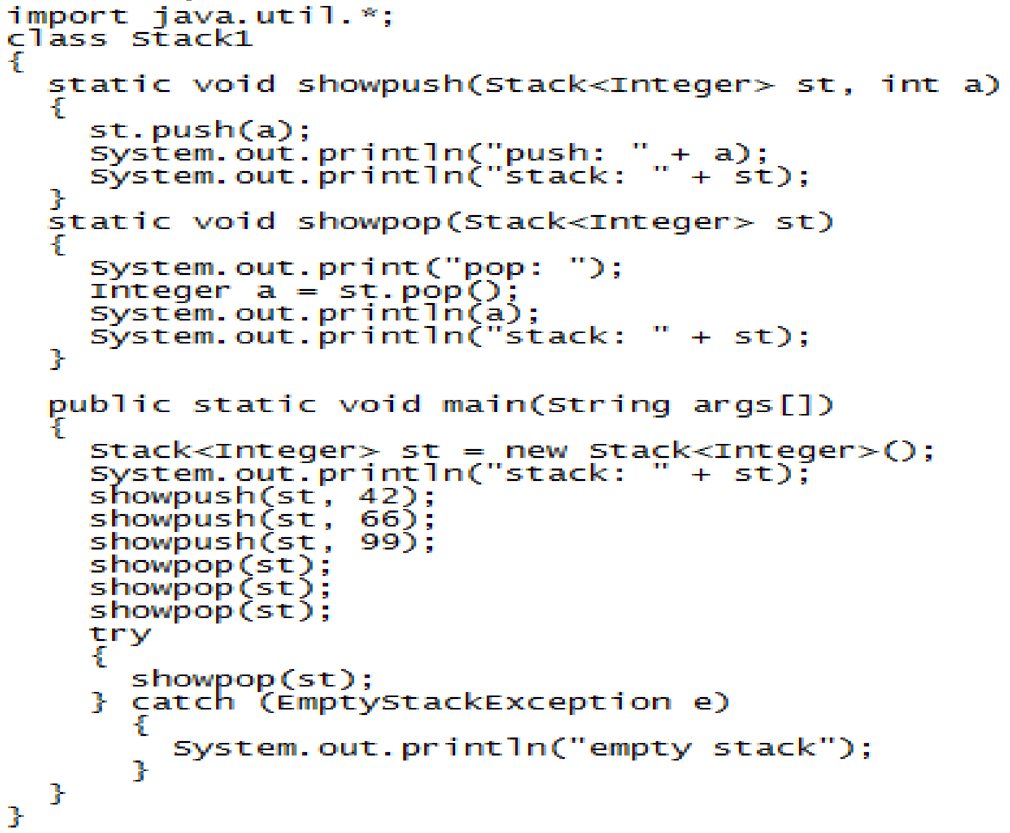
    Here, E specifies the type of element stored in the stack.
  - Stack includes all the methods defined by Vector and adds several of its own shown in the next slide.

Legacy Classes and Interfaces (Cont…)

Methods	Descriptions
boolean empty( )	Returns true, if the stack is empty. Otherwise, returns false, if the stack contains elements.
E peek( )	Returns the element on the top of the stack, but does not remove it.
E pop( )	Returns the element on the top of the stack, removing it in the process.
E push(E *element*)	Pushes element onto the stack and element is also returned.
int search(Object *element*)	Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

Legacy Classes and Interfaces (Cont…)

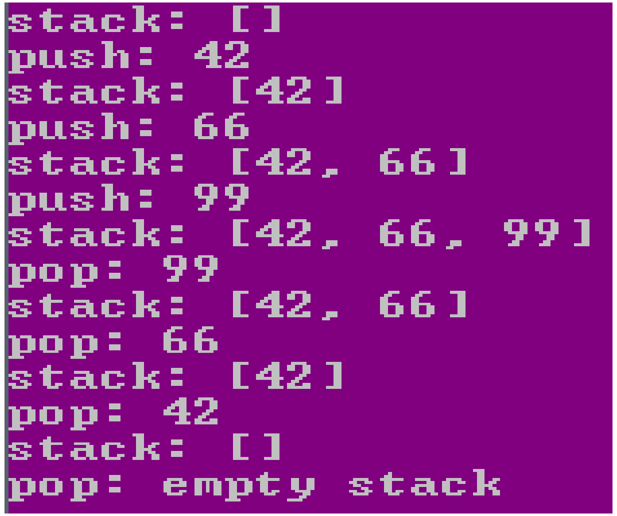
- Stack (Cont…)



```
import java.util.*;
class Stack1
{
   static void showpush(Stack<Integer> st, int a)
   {
      st.push(a);
      System.out.println("push: " + a);
      System.out.println("stack: " + st);
   }
   static void showpop(Stack<Integer> st)
   {
      System.out.print("pop: ");
      Integer a = st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

   public static void main(String args[])
   {
      Stack<Integer> st = new Stack<Integer>();
      System.out.println("stack: " + st);
      showpush(st, 42);
      showpush(st, 66);
      showpush(st, 99);
      showpop(st);
      showpop(st);
      showpop(st);
      try
      {
         showpop(st);
      } catch (EmptyStackException e)
         {
            System.out.println("empty stack");
         }
   }
}
```

Legacy Classes and Interfaces (Cont…)

- Stack (Cont…)
  - Output



```
stack: []
push: 42
stack: [42]
push: 66
stack: [42, 66]
push: 99
stack: [42, 66, 99]
pop: 99
stack: [42, 66]
pop: 66
stack: [42]
pop: 42
stack: []
pop: empty stack
```

Legacy Classes and Interfaces (Cont…)

- **Dictionary Class:**
  - Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
  - Given a key and value, we can store the value in a Dictionary object. Once the value is stored, we can retrieve it by using its key.
  - However, it is better to use Map.
  - Dictionary is generic and it is declared as:

    class Dictionary<K, V>

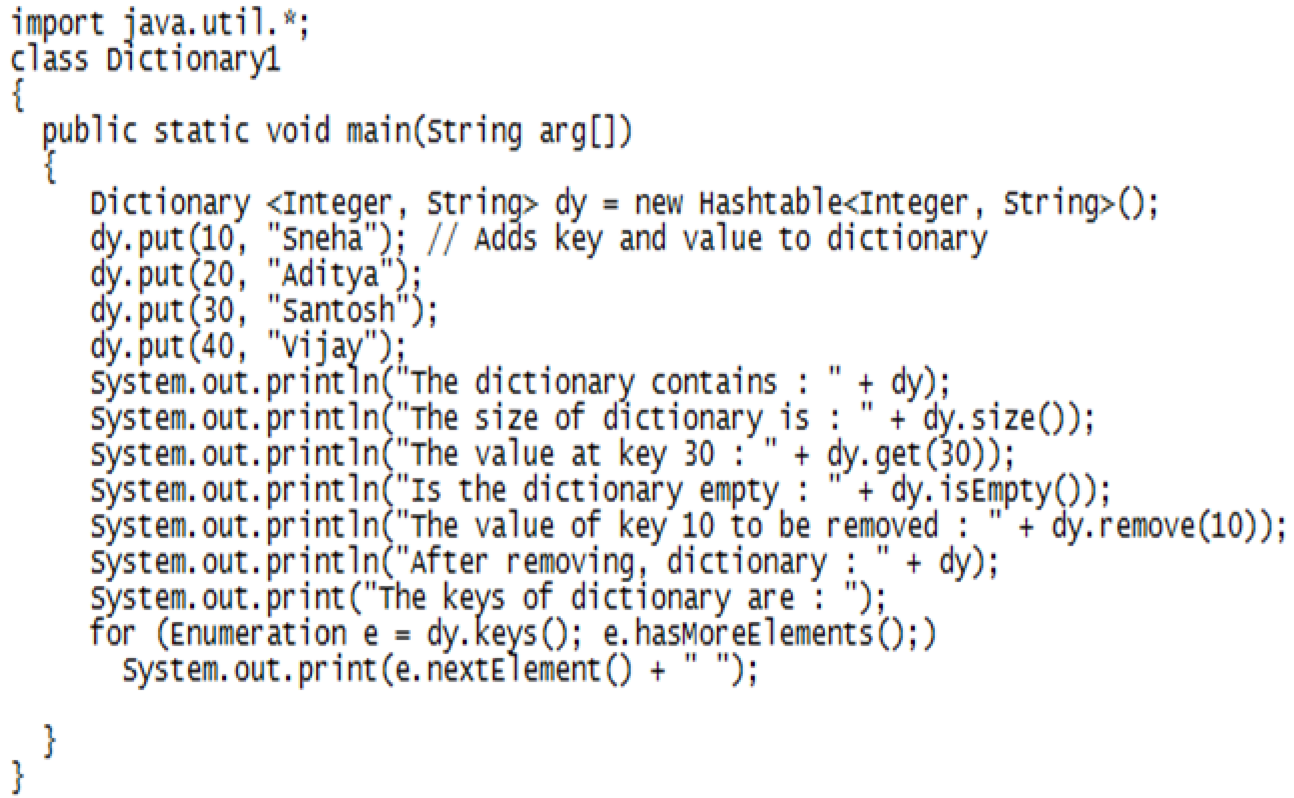
    Here, K specifies the type of keys, and V specifies the type of values.
  - The methods defined by Dictionary are mentioned in the next slide.

Legacy Classes and Interfaces (Cont…)

Methods	Descriptions
Enumeration<V> elements( )	Returns an enumeration of the values contained in the dictionary.
V get(Object *key*)	Returns the object that contains the value associated with key. If key is not in the dictionary, a null object is returned.
boolean isEmpty( )	Returns true, if the dictionary is empty. Otherwise, returns false.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the dictionary.
V put(K *key*, V *value*)	Inserts a key and its value into the dictionary. Returns null, if key is not already in the dictionary. It returns the previous value associated with key, if key is already in the dictionary.
V remove(Object *key*)	Removes key and its value. Returns the value associated with key. If key is not in the dictionary, null is returned.
int size( )	Returns the number of entries in the dictionary.

Legacy Classes and Interfaces (Cont…)

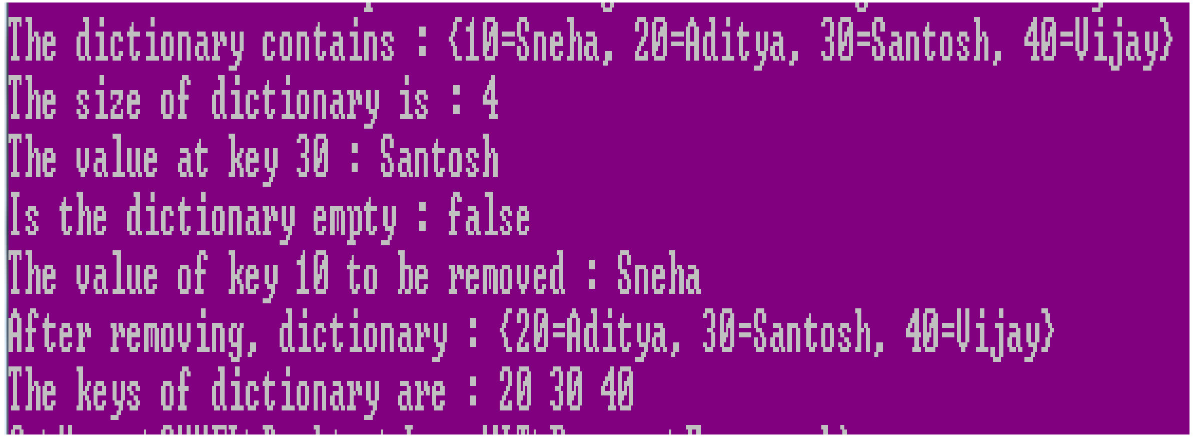
- **Dictionary Class (Cont…):**



```
import java.util.*;
class Dictionary1
{
  public static void main(String arg[])
  {
     Dictionary <Integer, String> dy = new Hashtable<Integer, String>();
     dy.put(10, "Sneha"); // Adds key and value to dictionary
     dy.put(20, "Aditya");
     dy.put(30, "Santosh");
     dy.put(40, "Vijay");
     System.out.println("The dictionary contains : " + dy);
     System.out.println("The size of dictionary is : " + dy.size());
     System.out.println("The value at key 30 : " + dy.get(30));
     System.out.println("Is the dictionary empty : " + dy.isEmpty());
     System.out.println("The value of key 10 to be removed : " + dy.remove(10));
     System.out.println("After removing, dictionary : " + dy);
     System.out.print("The keys of dictionary are : ");
     for (Enumeration e = dy.keys(); e.hasMoreElements();)
       System.out.print(e.nextElement() + " ");

  }
}
```

Legacy Classes and Interfaces (Cont…)

- **Dictionary Class (Cont…):**
  - **Output**



```
The dictionary contains : {10=Sneha, 20=Aditya, 30=Santosh, 40=Vijay}
The size of dictionary is : 4
The value at key 30 : Santosh
Is the dictionary empty : false
The value of key 10 to be removed : Sneha
After removing, dictionary : {20=Aditya, 30=Santosh, 40=Vijay}
The keys of dictionary are : 20 30 40
```

Legacy Classes and Interfaces (Cont…)

- **Hashtable Class:**
  - Hashtable class is similar to **HashMap**, but it is synchronized.
  - **Hashtable** implements the **Map** interface. So, it is integrated into the Collections Framework.
  - Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, here neither keys nor values can be **null**.
  - When using a **Hashtable**, we specify an object that is used as a key, and the value that we want linked to that key.
  - The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
  - **Hashtable** was made generic by JDK 5 and it is declared like below:

```
class Hashtable<K, V>
```

Legacy Classes and Interfaces (Cont…)

- **Hashtable Class (Cont…):**
  - A hash table can only store objects that override the hashCode( ) and equals( ) (Compare two objects) methods that are defined by Object.
  - hashCode( ) must compute and return the hash code for the object.
  - Many built-in classes implement the hashCode( ) (Ex: String object).
  - There are four Hashtable constructors:

    Hashtable( ) //It is default constructor

    Hashtable(int *size*) //Creates with the initial *size* (Default size is 11)

    Hashtable(int *size*, float *fillRatio*) //Fill ratio must be 0.0 to 1.0. It determines how full the hash table can be before it is resized upward. When the number of elements is greater than the capacity of the hash table, multiply by its fill ratio. If we do not specify a fill ratio, then 0.75 is used.

    Hashtable(Map<? extends K, ? extends V> *m*) //Initializes with elements in *m*

Legacy Classes and Interfaces (Cont…)

- **Hashtable Class (Cont…):**
  - Some methods of Hashtable are mentioned below:

Methods	Descriptions
void clear( )	Resets and empties the hash table.
Enumeration<V> elements( )	Returns an enumeration of the values contained in the hash table.
V get(Object *key*)	Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.
V put(K *key*, V *value*)	Inserts a key and a value into the hash table. Returns null, if key isn't already in the hash table. It returns the previous value associated with key, if key is already in the hash table.
int size( )	Returns the number of entries in the hash table.

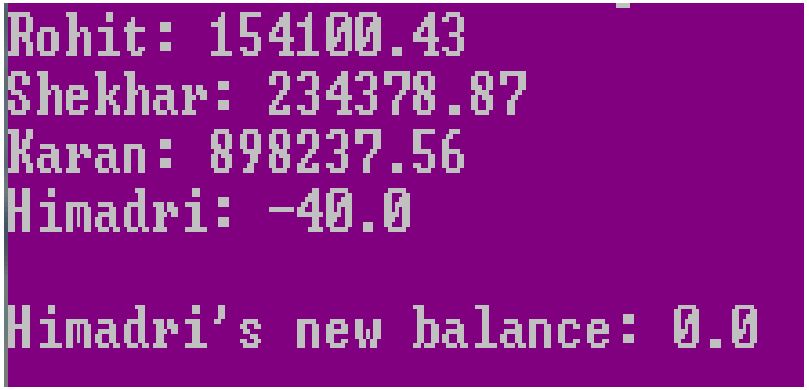
Legacy Classes and Interfaces (Cont…)

- **Hashtable Class (Cont…):**

```
import java.util.*;
class Hashtable1
{
  public static void main(String args[])
  {
    Hashtable<String, Double> balance = new Hashtable<String, Double>();
    Enumeration<String> names;
    String str;
    double bal;
    balance.put("Rohit", 154100.43);
    balance.put("Karan", 898237.56);
    balance.put("Shekhar", 234378.87);
    balance.put("Himadri", -40.00);

    names = balance.keys();     //Show all balances of the hashtable
    while(names.hasMoreElements())
    {
      str = names.nextElement();
      System.out.println(str + ": " + balance.get(str));
    }
    System.out.println();

    bal = balance.get("Himadri");     //Deposit 40 into Himadri's account
    balance.put("Himadri", bal+40);
    System.out.println("Himadri's new balance: " + balance.get("Himadri"));
  }
}
```

Legacy Classes and Interfaces (Cont…)

- **Hashtable Class (Cont…):**
  - **Output**



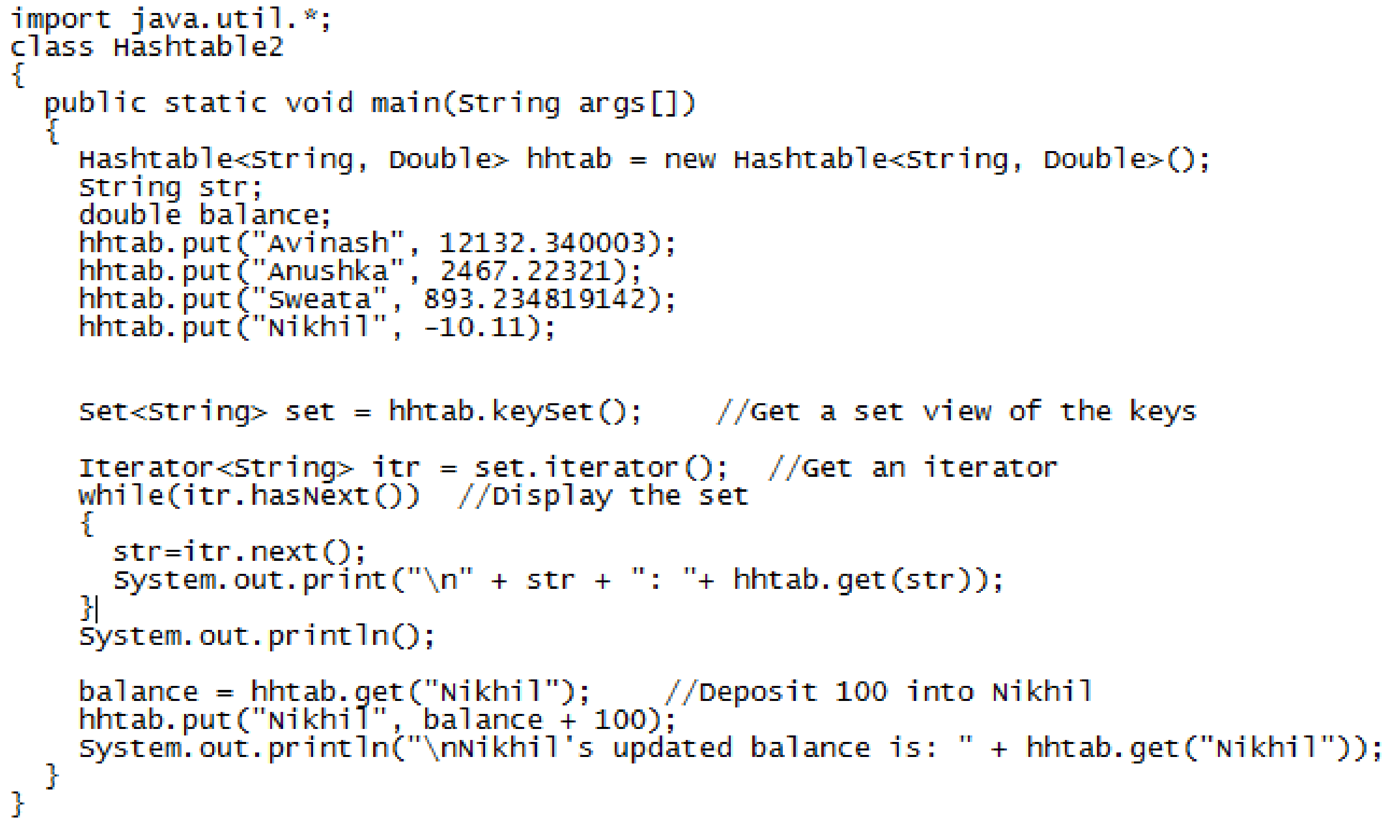
```
Rohit: 154100.43
Shekhar: 234378.87
Karan: 898237.56
Himadri: -40.0

Himadri's new balance: 0.0
```

Legacy Classes and Interfaces (Cont…)

- **Hashtable Class (Cont…):**
  - Like the map classes, Hashtable does not directly support iterators.
  - So, an enumeration is used in the **last program** to display the contents of balance.
  - However, we can obtain set-views of the hash table, which permits the use of iterators. To do so, we simply use one of the collection-view methods defined by Map, such as entrySet( ) or keySet( ).
  - For an example, we can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced for loop.
  - In the next slide, an example is shown to depict how to use iterator.

Legacy Classes and Interfaces (Cont…)

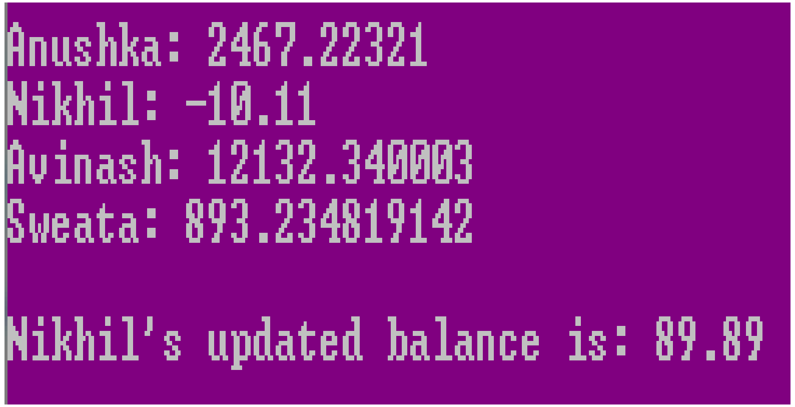
- **Hashtable Class (Cont…):**



```
import java.util.*;
class Hashtable2
{
  public static void main(String args[])
  {
    Hashtable<String, Double> hhtab = new Hashtable<String, Double>();
    String str;
    double balance;
    hhtab.put("Avinash", 12132.340003);
    hhtab.put("Anushka", 2467.22321);
    hhtab.put("Sweata", 893.234819142);
    hhtab.put("Nikhil", -10.11);


    Set<String> set = hhtab.keySet();    //Get a set view of the keys

    Iterator<String> itr = set.iterator();  //Get an iterator
    while(itr.hasNext())  //Display the set
    {
      str=itr.next();
      System.out.print("\n" + str + ": "+ hhtab.get(str));
    }
    System.out.println();

    balance = hhtab.get("Nikhil");    //Deposit 100 into Nikhil
    hhtab.put("Nikhil", balance + 100);
    System.out.println("\nNikhil's updated balance is: " + hhtab.get("Nikhil"));
  }
}
```

Legacy Classes and Interfaces (Cont…)

- **Hashtable Class (Cont…):**
  - **Output**



```
Anushka: 2467.22321
Nikhil: -10.11
Avinash: 12132.340003
Sweata: 893.234819142

Nikhil's updated balance is: 89.89
```

Legacy Classes and Interfaces (Cont…)

- **Properties Class:**
  - Properties is a subclass of Hashtable.
  - It is used to maintain lists of values in which the key is a String and the value is also a String.
  - Properties class is not generic. However, several of its methods are generic.
  - Properties defines the following instance variable:

    Properties defaults;
  - Properties has two constructors:

    Properties( ) //Creates a properties object with no default values

    Properties(Properties *propDefault*) //Creates an object that uses *propDefault* for its default values. In both cases, the property list is empty.

Legacy Classes and Interfaces (Cont…)

- **Properties Class (Cont…):**
  - **Some** of the methods of this class are mentioned in the next slide.
  - Here, a default value an be specified along with the key in the getProperty( ) method, such as getProperty("name", "default value"). If the "name" value is not found, then "default value" is returned.
  - When we construct a Properties object, we can pass another instance of Properties to be used as the default properties for the new instance.
  - For an example: if we call getProperty("foo") on a given Properties object, and "foo" does not exist, Java looks for "foo" in the default Properties object. This allows for arbitrary nesting of levels of default properties.

Legacy Classes and Interfaces (Cont…)

Methods	Descriptions
String getProperty(String *key*)	Returns the value associated with key. A null object is returned, if key is neither in the list nor in the default property list.
String getProperty(String *key*, String *defaultProperty*)	Returns the value associated with key. defaultProperty is returned, if key is neither in the list nor in the default property list.
void list(PrintStream *streamOut*)	Sends the property list to the output stream linked to streamOut.
void list(PrintWriter *streamOut*)	Sends the property list to the output stream linked to streamOut.
Object setProperty(String *key*, String *value*)	Associates value with key. Returns the previous value associated with key. Returns null, if no such association exists.
Set<String> stringPropertyNames( )	Returns a set of keys.

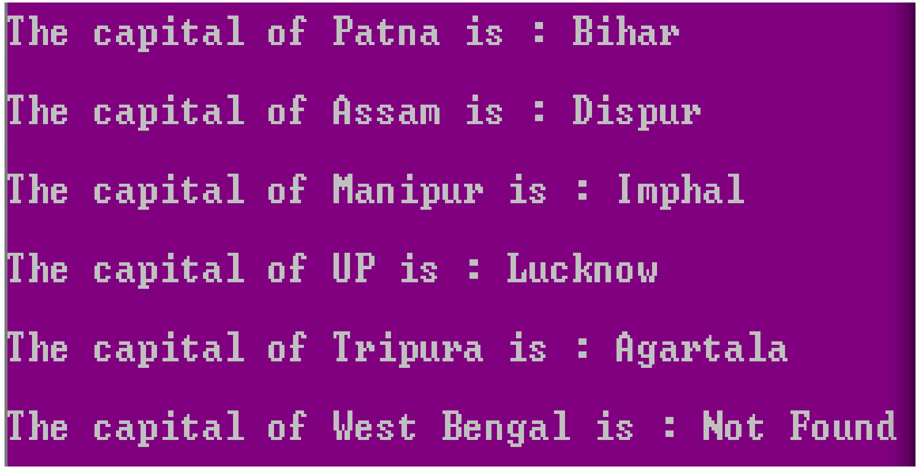
Legacy Classes and Interfaces (Cont…)

- **Properties Class (Cont…):**

```
import java.util.*;
class Properties1
{
  public static void main(String args[])
  {
    Properties capitals = new Properties();
    capitals.put("Tripura", "Agartala");
    capitals.put("Assam", "Dispur");
    capitals.put("Manipur", "Imphal");
    capitals.put("UP", "Lucknow");
    capitals.put("Patna", "Bihar");

    Set states = capitals.keySet();     //Get a set-view of the keys

    for(Object name : states)    // Show all of the states and capitals
      System.out.println("\nThe capital of " + name + " is : " + capitals.getProperty((String)name));
    System.out.println();

    String str = capitals.getProperty("West Bengal", "Not Found");   //Look for state not in the list-Set default
    System.out.println("The capital of West Bengal is : " + str);
  }
}
```

Legacy Classes and Interfaces (Cont…)

- **Properties Class (Cont…):**
  - **Output**



```
The capital of Patna is : Bihar
The capital of Assam is : Dispur
The capital of Manipur is : Imphal
The capital of UP is : Lucknow
The capital of Tripura is : Agartala
The capital of West Bengal is : Not Found
```

Legacy Classes and Interfaces (Cont…)

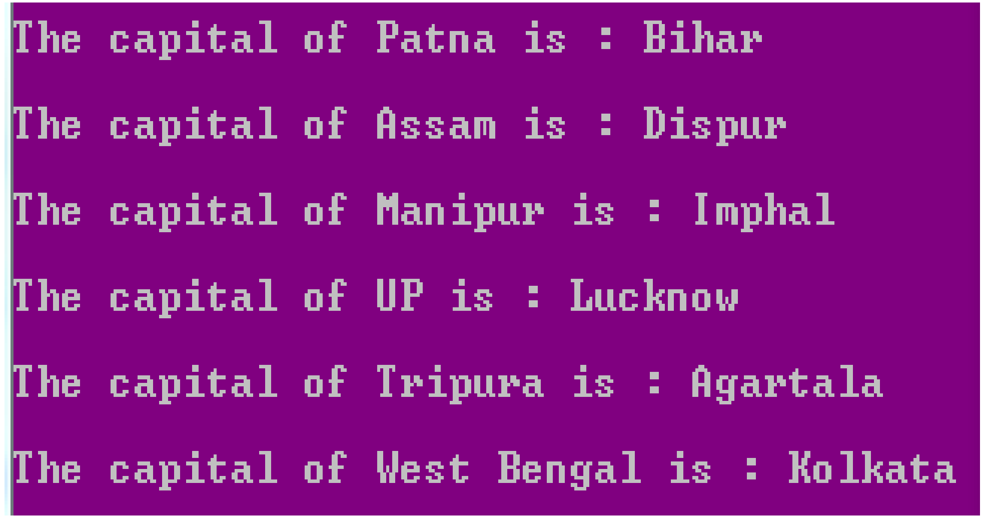
❑ Properties Class (Cont…):

❖ Another way to write the **Previous Program**

```
//Another way to write the Previous Program
import java.util.*;
class Properties2
{
  public static void main(String args[])
  {
    Properties defList = new Properties();
    defList.put("West Bengal", "Kolkata");
    defList.put("Karnataka", "Bengaluru");

    Properties capitals = new Properties(defList);
    capitals.put("Tripura", "Agartala");
    capitals.put("Assam", "Dispur");
    capitals.put("Manipur", "Imphal");
    capitals.put("UP", "Lucknow");
    capitals.put("Patna", "Bihar");

    Set states = capitals.keySet();     //Get a set-view of the keys

    for(Object name : states)    // Show all of the states and capitals
      System.out.println("\nThe capital of " + name + " is : " + capitals.getProperty((String)name));
    System.out.println();

    String str = capitals.getProperty("West Bengal");   //West Bengal is in the default list
    System.out.println("The capital of West Bengal is : " + str);
  }
}
```

Legacy Classes and Interfaces (Cont…)

- **Properties Class (Cont…):**
  - **Output**



```
The capital of Patna is : Bihar
The capital of Assam is : Dispur
The capital of Manipur is : Imphal
The capital of UP is : Lucknow
The capital of Tripura is : Agartala
The capital of West Bengal is : Kolkata
```

thank
you . . .

Slides are prepared from various sources, such as Book, Internet Links and many more.