Collections and Utilities

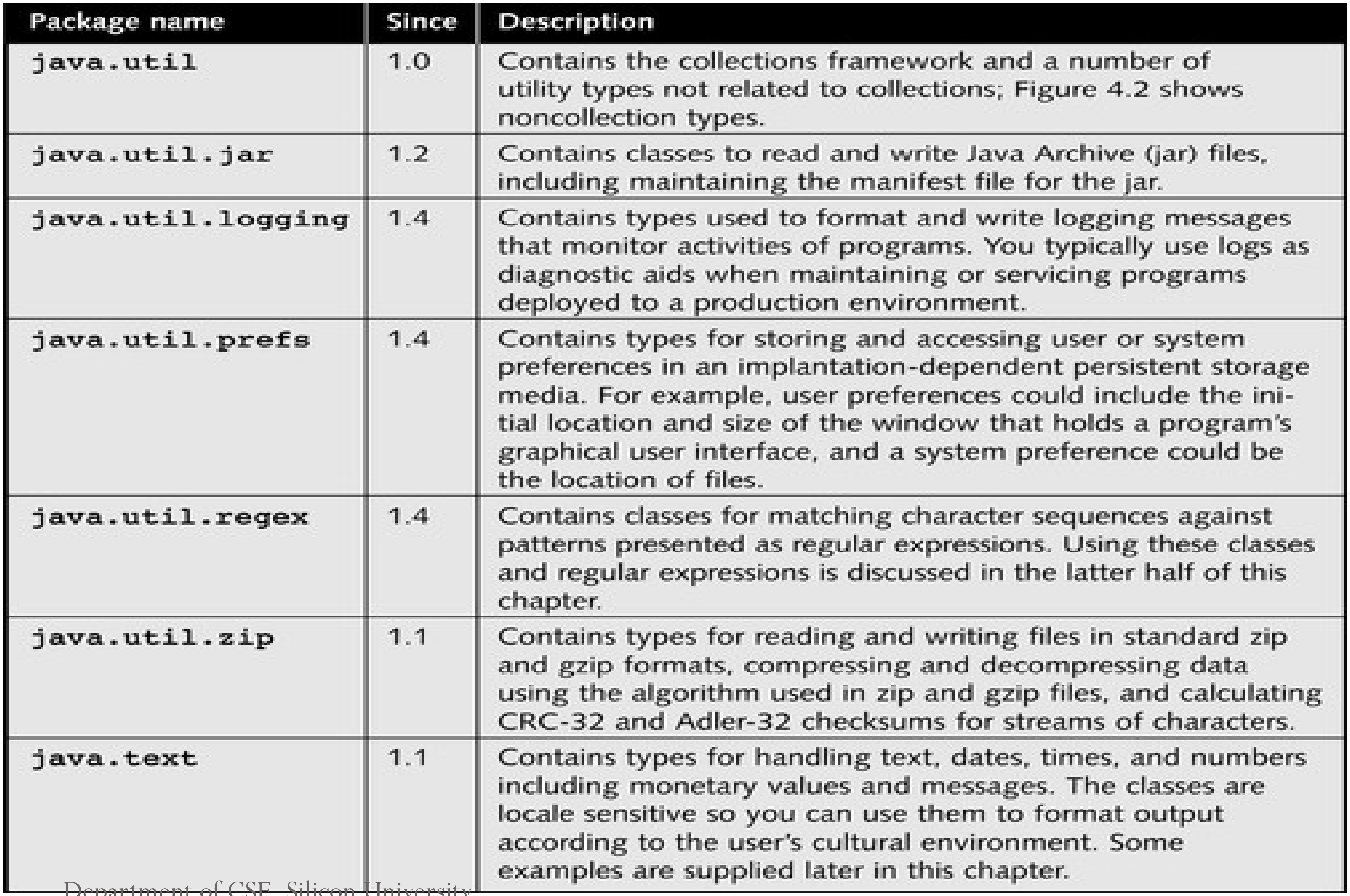
Prepared by

Dr. Rajesh Kumar Ojha
Asst. Prof., CSE, Silicon University

The Utility Packages

- In the first version of Java, the java.util package contained general purpose utility classes and interfaces
- Utility packages are in the java.util package and the packages inside it
- The java.util package contains the collections framework and a number of utility types not related to collections

The Utility Packages

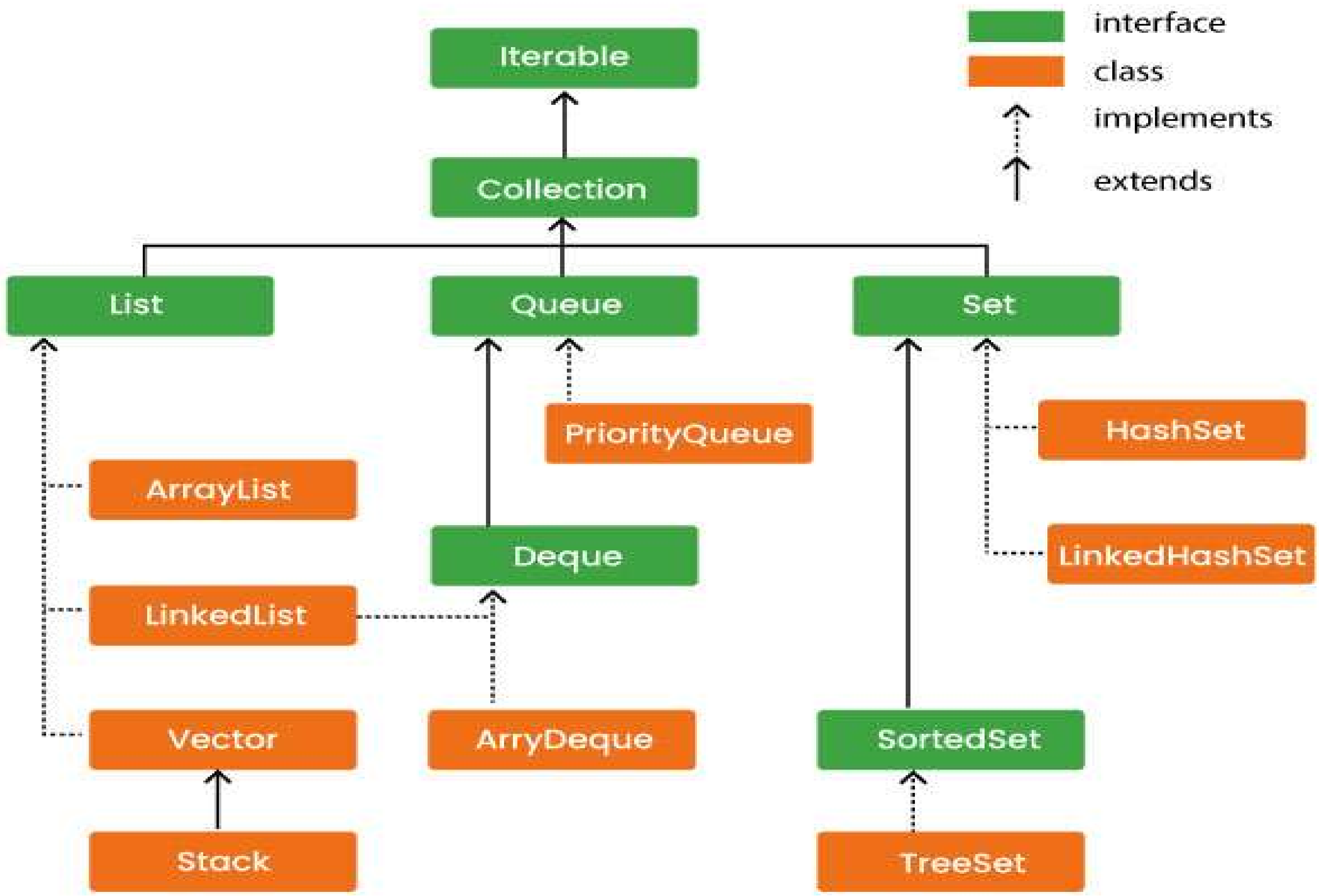


Package name	Since	Description
`java.util`	1.0	Contains the collections framework and a number of utility types not related to collections; Figure 4.2 shows noncollection types.
`java.util.jar`	1.2	Contains classes to read and write Java Archive (jar) files, including maintaining the manifest file for the jar.
`java.util.logging`	1.4	Contains types used to format and write logging messages that monitor activities of programs. You typically use logs as diagnostic aids when maintaining or servicing programs deployed to a production environment.
`java.util.prefs`	1.4	Contains types for storing and accessing user or system preferences in an implantation-dependent persistent storage media. For example, user preferences could include the initial location and size of the window that holds a program's graphical user interface, and a system preference could be the location of files.
`java.util.regex`	1.4	Contains classes for matching character sequences against patterns presented as regular expressions. Using these classes and regular expressions is discussed in the latter half of this chapter.
`java.util.zip`	1.1	Contains types for reading and writing files in standard zip and gzip formats, compressing and decompressing data using the algorithm used in zip and gzip files, and calculating CRC-32 and Adler-32 checksums for streams of characters.
`java.text`	1.1	Contains types for handling text, dates, times, and numbers including monetary values and messages. The classes are locale sensitive so you can use them to format output according to the user's cultural environment. Some examples are supplied later in this chapter.

The Collections Framework

The collections framework consists of:

1. Interfaces that define the behavior of collections
2. Concrete classes that provide general-purpose implementations of the interfaces that you can use directly
3. Abstract classes that implement the interfaces of the collections framework that you can extend to create collections for specialized data structures


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArryDeque
SortedSet
Stack
TreeSet

The goals of the collections framework

1. Reduce the programming effort of developers by providing the most common data structures.
2. Provide a set of types that are easy to use, extend, and understand.
3. Increase flexibility by defining a standard set of interfaces for collections to implement.
4. Improve program quality through the reuse of tested software components.

The Classes of Collections Framework

1. Bag: a group of elements
2. Iterator: a cursor, a helper object that clients of the collections use to access elements one at a time
3. List: a group of elements that are accessed in order
4. Set: a group of unique elements
5. Tree: a group of elements with a hierarchical structure

Collection Classes

ArrayList class

ArrayList class

1. The Java ArrayList class provides a resizable-array and implements the **List interface**.
2. It implements all optional list operations and it also permits all elements, includes null.
3. It provides methods to manipulate the size of the array that is used internally to store the list.
4. The **ArrayList** class extends **AbstractList** and implements the **List interface**.
5. ArrayList supports dynamic arrays that can grow as needed.
6. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

public class ArrayList<E>   extends AbstractList<E>
   implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Where <E> represents an Element. For example, if you're building an array list of Integers then you'd initialize it as:

ArrayList<Integer> list = new ArrayList<Integer>();

Constructors of ArrayList

Sr.No.	Constructor & Description
1	**ArrayList()** This constructor is used to create an empty list with an initial capacity sufficient to hold 10 elements.
2	**ArrayList(Collection<? extends E> c)** This constructor is used to create a list containing the elements of the specified collection.
3	**ArrayList(int initialCapacity)** This constructor is used to create an empty list with an initial capacity.

Methods of ArrayList

Sr.No.	Method & Description
1	boolean add(E e)This method appends the specified element to the end of this list.
2	boolean addAll(Collection<? extends E> c)This method appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator
3	void clear()This method removes all of the elements from this list.
4	Object clone()This method returns a shallow copy of this ArrayList instance.
5	boolean contains(Object o)This method returns true if this list contains the specified element.
6	void ensureCapacity(int minCapacity)This increases the capacity of this ArrayList.
7	E get(int index)This method returns the element at the specified position in this list.

8	**int indexOf(Object o)**This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
9	**boolean isEmpty()**This method returns true if this list contains no elements.
10	**Iterator<E> iterator()**This method returns an iterator over the elements in this list in proper sequence.
11	**int lastIndexOf(Object o)**This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
12	**ListIterator<E> listIterator()**This method returns an list iterator over the elements in this list in proper sequence.
13	**E remove(int index)**This method removes the element at the specified position in this list.
14	**boolean removeAll(Collection<?> c)**Removes from this list all of its elements that are contained in the specified collection.
15	**protected void removeIf(int fromIndex, int toIndex)**This method Removes all of the elements of this collection that satisfy the given predicate.

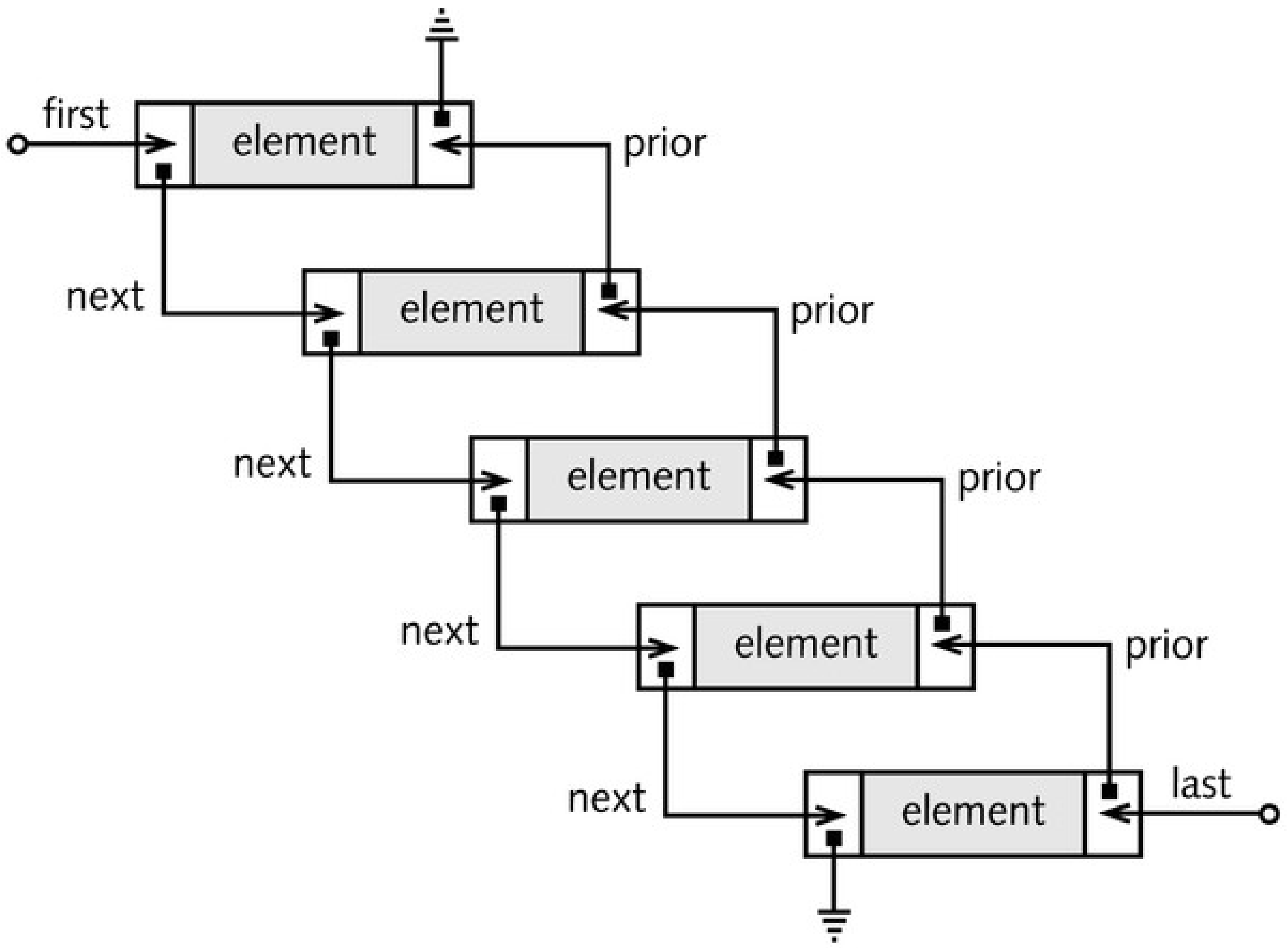
16	**boolean retainAll(Collection<?> c)**Retains from this list all of its elements that are contained in the specified collection.
17	**E set(int index, E element)**This method replaces the element at the specified position in this list with the specified element.
18	**int size()**This method returns the number of elements in this list.
19	**Spliterator<E> spliterator()**This method creates a late-binding and fail-fast Spliterator over the elements in this list.
20	**List<E> subList(int fromIndex, int toIndex)**This method returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
21	**Object[] toArray()**This method returns an array containing all of the elements in this list in proper sequence (from first to last element).
22	**void trimToSize()**This method trims the capacity of this ArrayList instance to be the list's current size.

```
import java.util.*;

class Main {
   public static void main(String args[]){

        ArrayList<String> al = new ArrayList<>(); // Creating an Array of string type
        al.add("Hi");                     // Adding elements to ArrayList at the end
        al.add("There");
        System.out.println("Orignal List : "+al);
        al.add(1, "I am");            // Adding Elements at the specific index
        System.out.println("After Adding element at index 1 : "+ al);
        al.remove(0);                    // Removing Element using index
        System.out.println("Element removed from index 0 : "+ al);
        al.remove("Hi");              // Removing Element using the value
        System.out.println("Element Geeks removed : "+ al);
        al.set(0, "Hello");           // Updating value at index 0
        // Printing all the elements in an ArrayList
        System.out.println("List after updation of value : "+al);
}
}
```

Collection Classes

LinkedList class

A Doubly Linked List

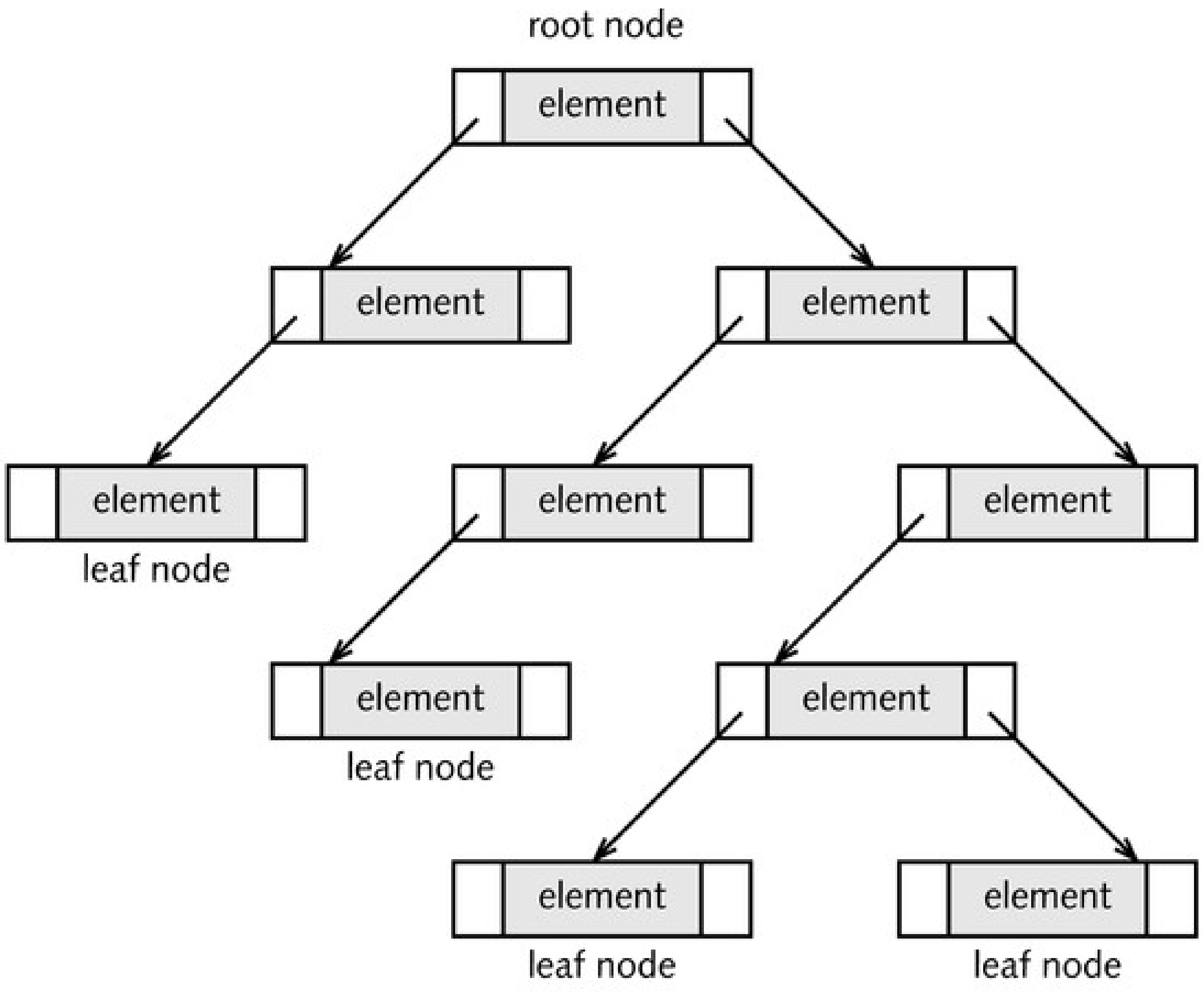


Methods of Linked List class

Method	Description
add(int index, E element)	This method Inserts the specified element at the specified position in this list.
add(E e)	This method Appends the specified element to the end of this list.
addAll(int index, Collection<E> c)	This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
addAll(Collection<E> c)	This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection’s iterator.
addFirst(E e)	This method Inserts the specified element at the beginning of this list.
addLast(E e)	This method Appends the specified element to the end of this list.
clear()	This method removes all of the elements from this list.
clone()	This method returns a shallow copy of this LinkedList.
contains(Object o)	This method returns true if this list contains the specified element.

```
import java.util.*;

public class LinkedListExample
{
   public static void main(String args[])
   {
      // Creating object of the class linked list
      LinkedList<String> ll = new LinkedList<String>();
      // Adding elements to the linked list
      ll.add("A");
      ll.add("B");
      ll.addLast("C");
      ll.addFirst("D");
      ll.add(2, "E");
      System.out.println(ll);
      ll.remove("B");
      ll.remove(3);
      ll.removeFirst();
      ll.removeLast();
      System.out.println(ll);
   }
}
```

A Binary Tree



Example Tree class

```
class tree{
 String val;
 tree left=null;
 tree right=null;
 tree (String s){ val=s;}// constructor
 static tree ins(tree t,String s){
        if (t==null) return new tree(s);
        if( t.val.compareTo(s)<0)
             t.right=ins(t.right,s);
        if(t.val.compareTo(s)>=0)
             t.left=ins(t.left,s);
        return t;
  }
```

More of tree class

```
  void ins(String s){ ins(this,s);}
  void visit(Visitor v){
        v.act(val);
        if(left!=null) left.visit(v);
        if(right!=null) right.visit(v);
  }
  class printer implements visitor{
   void act(Object
  o){System.out.println(o.toString());}
   }
  void printTree() { visit(new Printer());}
}
interface visitor{ void act(Object o) ;}
```

Three Key Interfaces in the collections Framework

The root interface of the framework is Collection

- The Set interface:
  - Extends the Collection interface
  - Defines standard behavior for a collection that does not allow duplicate elements
- The List interface:
  - Extends the Collection interface
  - Defines the standard behavior for ordered collections (*sequences*)

The Collection Interface

Methods:

1. boolean add(Object *element*)
2. boolean addAll(Collection *c*)
3. void clear()
4. boolean contains(Object *element*)
5. boolean containsAll(Collection *c*)
6. boolean equals(Object *o*)
7. int hashCode()
8. boolean isEmpty()

The Collection Interface (Cont.)

Methods:

1. Iterator iterator()
2. boolean remove(Object *element*)
3. boolean removeAll(Collection *c*)
4. boolean retainAll(Collection *c*)
5. int size()
6. Object[] toArray()
7. Object[] toArray(Object[] *a*)

The Set Interface

Methods**:**

- The boolean add(Object *element*) method:
  - Ensures collection contains the specified element
  - Returns false if element is already part of the set
- The boolean addAll(Collection *c*) method:
  - Adds all the elements in the specified collection to this collection
  - Returns false if all the elements in the specified collection are already part of the set

The List Interface

Methods:

1. boolean add(Object *element*)
2. void add(int index, Object *element*)
3. boolean addAll(Collection *c*)
4. boolean addAll(int *index*, Collection *c*)
5. Object get(int *index*)
6. int indexOf(Object *element*)
7. int lastIndexOf(Object *element*)

Note that the elements of the list have numbered positions – their indices

The List Interface (Cont.)

Methods:

1. ListIterator listIterator()
2. ListIterator listIterator(int *index*)
3. boolean remove(Object *element*)
4. Object remove(int *index*)
5. Object set(int *index*, Object *element*)
6. List subList(int *beginIndex*, int *endIndex*)

Traversing Collections with Iterators , 1

Iterator interface methods:

- boolean hasNext()
- Object next()
- void remove()

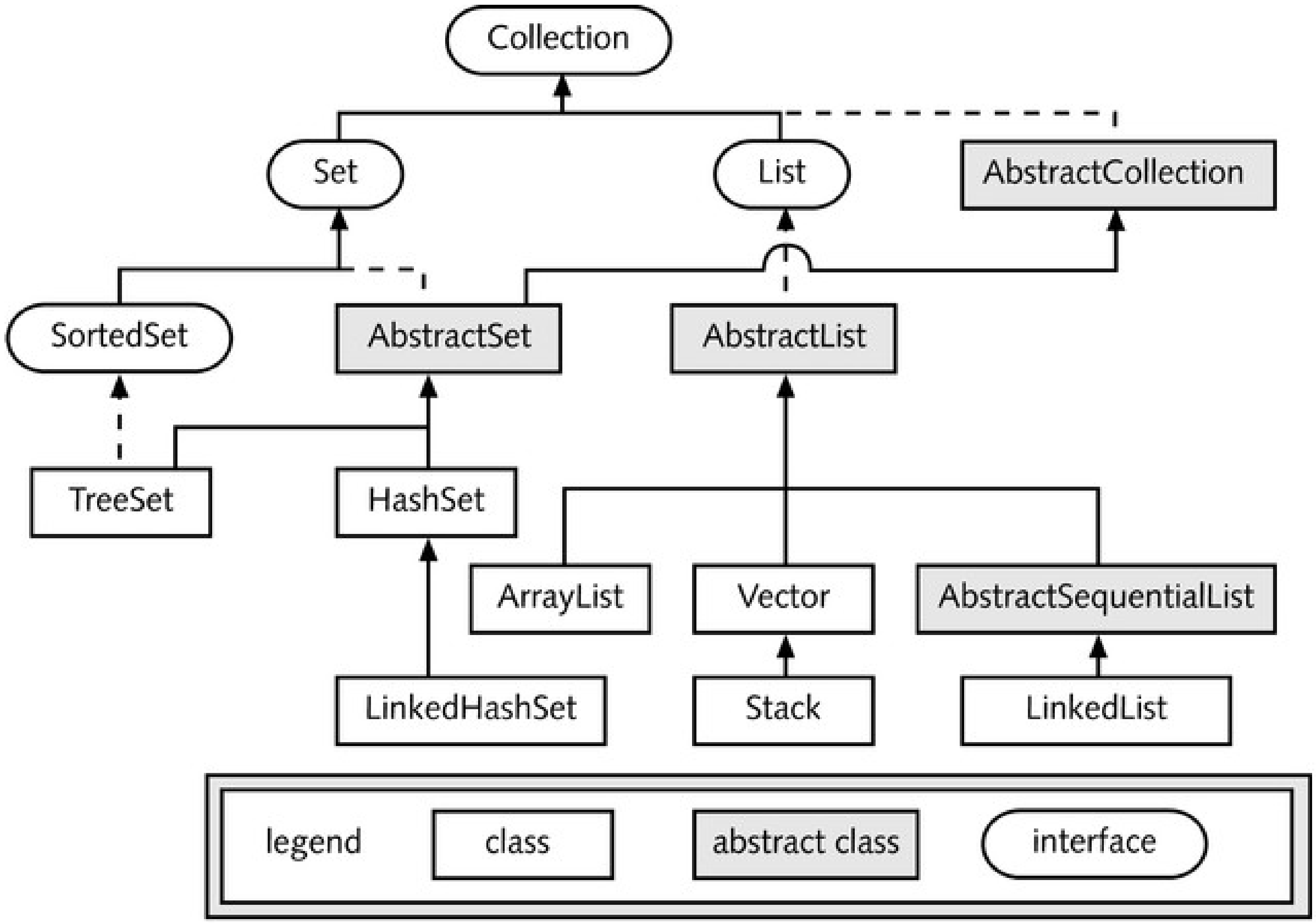
```
Iterator i = s.iterator();
 while (i.hasNext()){
   Object n = i.next();
   … do something with n
}
```

Traversing Collections with Iterators, 2

ListIterator interface methods:

- void add(Object *element*)
  - The element is inserted immediately before the next element that would be returned by next,
- boolean hasPrevious()
- int nextIndex()
- Object previous()
- int previousIndex()
- void set(Object *element*)

General Purpose Implementations



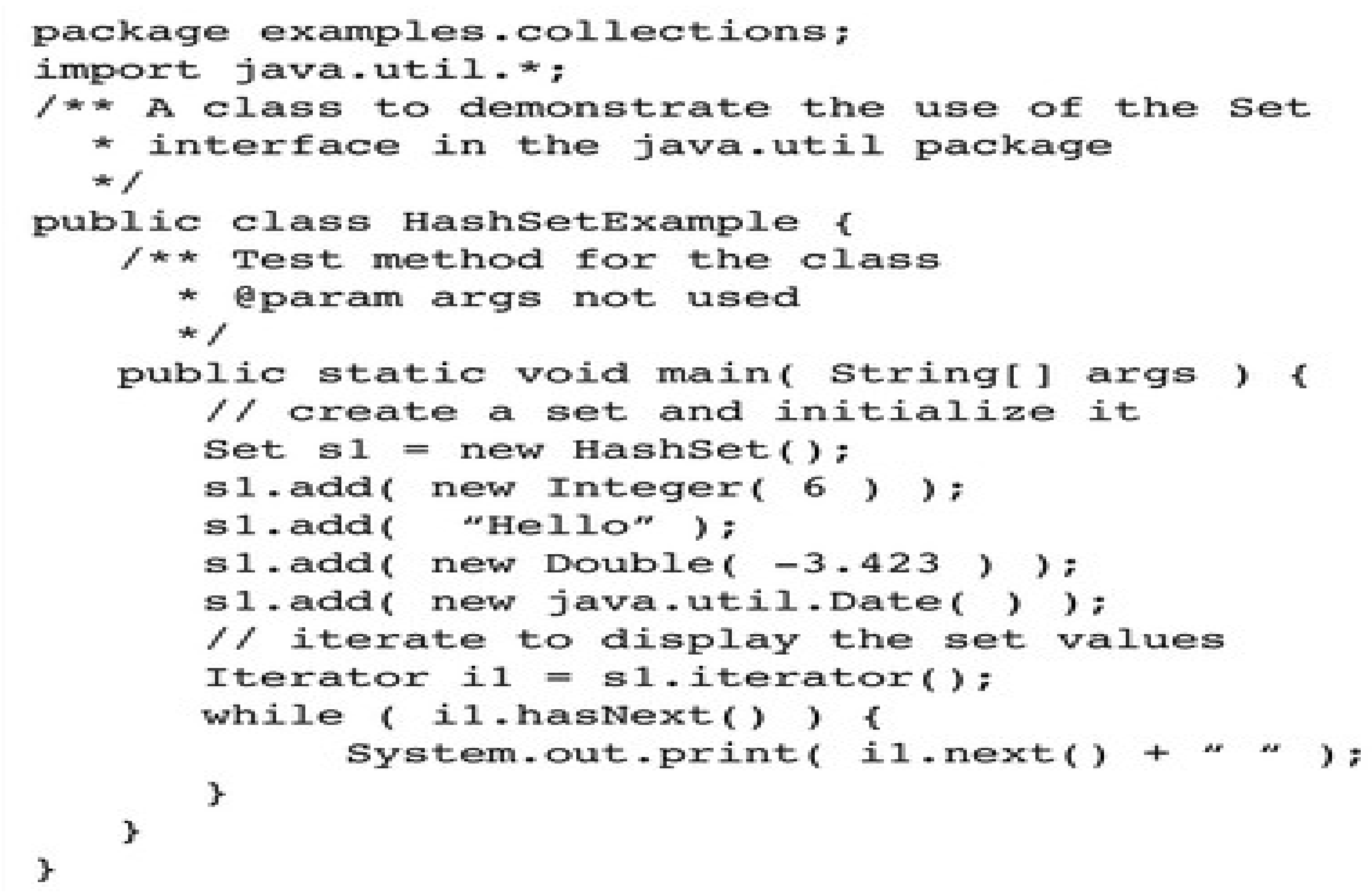
General Purpose Sets

Three framework classes implement the Set interface:

- HashSet
- TreeSet
- LinkedHashSet

Note that it is possible to add elements of different classes to the same set as the following example illustrates.

Sample Class Using a HashSet Collection

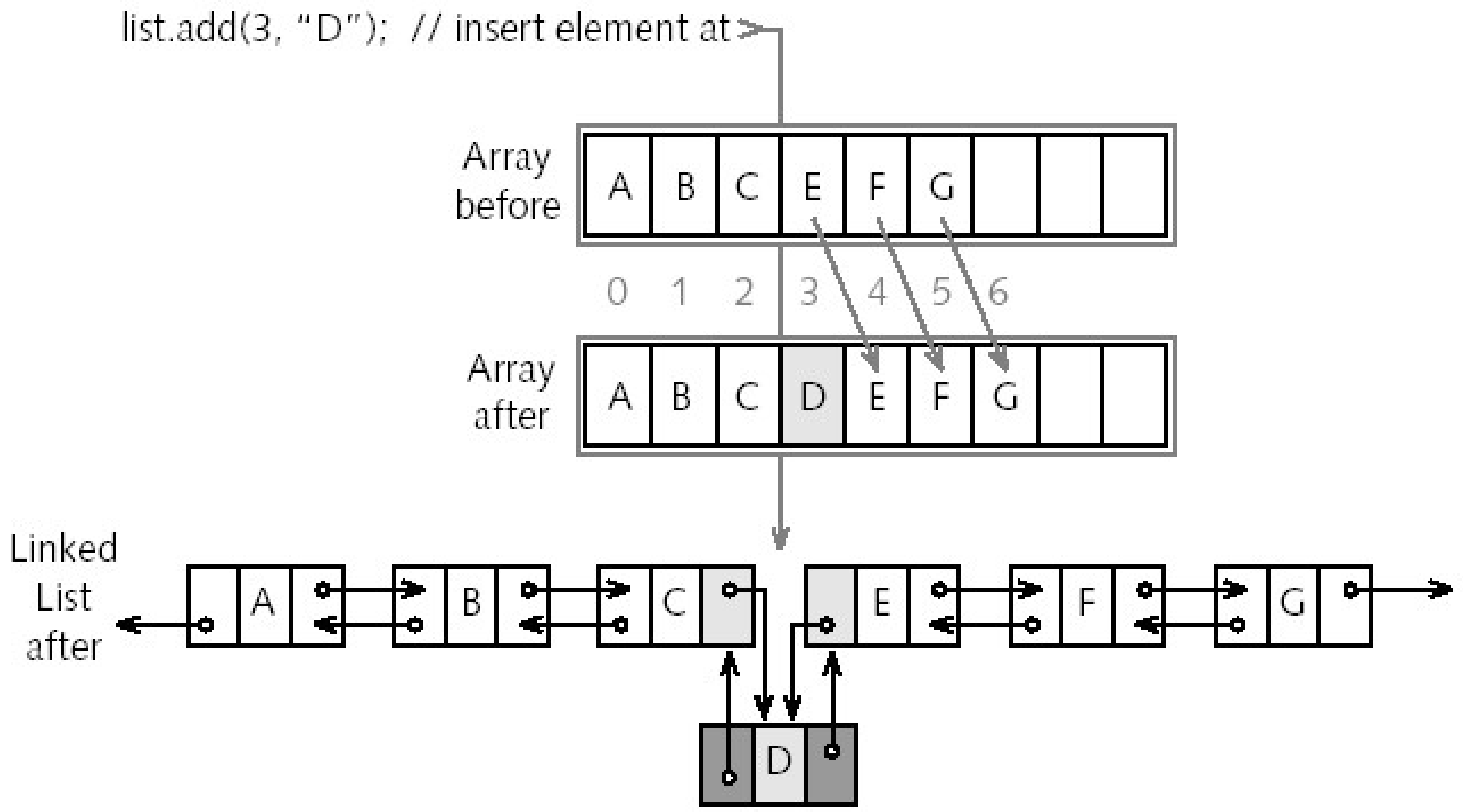


```
package examples.collections;
import java.util.*;
/** A class to demonstrate the use of the Set
   * interface in the java.util package
   */
public class HashSetExample {
    /** Test method for the class
       * @param args not used
       */
    public static void main( String[] args ) {
        // create a set and initialize it
        Set s1 = new HashSet();
        s1.add( new Integer( 6 ) );
        s1.add(  "Hello" );
        s1.add( new Double( -3.423 ) );
        s1.add( new java.util.Date( ) );
        // iterate to display the set values
        Iterator i1 = s1.iterator();
        while ( i1.hasNext() ) {
                System.out.print( i1.next() + " " );
        }
    }
}
```

General Purpose Lists

- Four concrete classes in the framework are implementations of the List interface:
  - ArrayList
  - LinkedList
  - Vector
  - Stack

Comparing Insert on an ArrayList and a Linked List



Array lists versus linked lists

- Array lists are more compact and thus use less memory and may impose less of a garbage collection overhead
- Linked lists are more efficient however, when insertions and deletions are common, as they do not require shifting of existing elements

Arrays as Collections

- toArray: converts a Collection object into an array
- java.util.Arrays: provides static methods that operate on array objects
- Arrays class: useful when integrating your programs with APIs that
  - Require array arguments
  - Return arrays

Sorted Collections

SortedSet adds methods to the Set interface:

1. Comparator comparator()
2. Object first()
3. SortedSet headSet(Object *element*)
4. Object last()
5. SortedSet subSet(int *beginElement*, int *endElement*)
6. SortedSet tailSet(Object *element*)

Maps

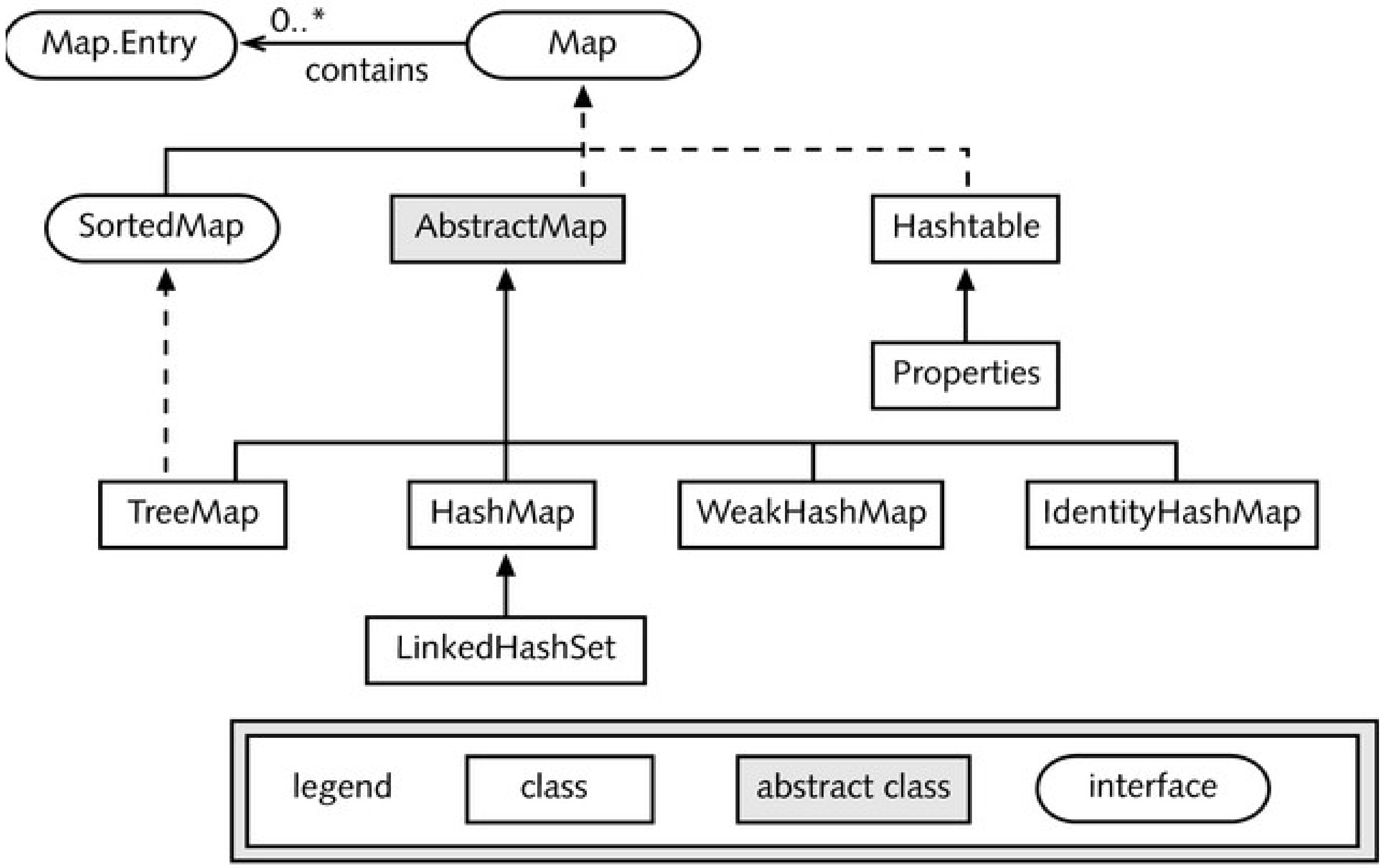
- A map is an abstraction for an aggregation of key-value, or name-value, pairs
- Two interfaces in the collections framework define the behavior of maps: Map and SortedMap
- A third interface, Map.Entry, defines the behavior of elements extracted from Map

Maps (Cont.)

Seven concrete framework classes implement the Map interface:

1. HashMap
2. IdentityHashMap
3. LinkedHashMap
4. TreeMap
5. WeakHashMap
6. Hashtable
7. Properties

The Map Types



Thank you