Collections

Collection

```
public class Name {
    private String firstName,lastName;
    public Name(String firstName, String lastName) {
        this.firstName = firstName; this.lastName = lastName;
    }
    public String getFirstName() {  return firstName;   }
    public String getLastName() {   return lastName;   }
    public String toString() {  return firstName + " " + lastName;  }
}
```

```
public class Test {
    public static void main(String arg[]) {
        Name name1 = new Name("f1","l1");
        Name name2 = new Name("f2","l2");
        Name name3 = new Name("f3","l3");
                    … … …
    }
}
```

Collection

- A collection represents a group of objects known as its elements.
  - A collection holds references to objects
  - But we say informally that it “holds objects”.

What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate.

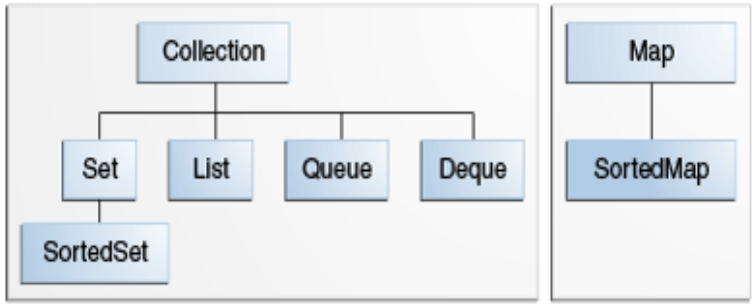
Benefits of the Java Collections Framework

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth.

Benefits of the Java Collections Framework

- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output.
- **Reduces effort to design new APIs:** Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Interface



- The *core collection interfaces* encapsulate different types of collections
- These interfaces allow collections to be manipulated independently of the details of their representation.
- Core collection interfaces are the foundation of the Java Collections Framework.

Interface

- Note that all the core collection interfaces are generic.
- For example, this is the declaration of the Collection interface.

  public interface Collection<E>...

  The <E> syntax tells you that the interface is generic.
- When you declare a Collection instance you can and should specify the type of object contained in the collection.
- Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

Collection

- Collection — the root of the collection hierarchy.
- A collection can contain references to two equal objects (a.equals (b)) as well as two references to the same object (a == b).
- An object can belong to several collections.
- An object can change while in a collection (unless it is immutable).
- The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired.
- Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered.

Collection

- Starting with Java 5, a collection holds objects of a specified type. A collection class's or interface's definition takes object type as a parameter:
  - *Collection*<***E***>
  - *List*<***E***>
  - Stack<***E***>
  - *Set*<***E***>
- A map takes two object type parameters:
  - Map<***K***,***V***>

Because collections work with different types, these are called *generic collections* or *generics*

Set

- Set — a collection that cannot contain duplicate elements.
  - This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- SortedSet — a Set that maintains its elements in ascending order.
  - Several additional operations are provided to take advantage of the ordering.
  - Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

List

- List — an ordered collection (sometimes called a sequence).
  - Lists can contain duplicate elements.
  - The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

Queue

- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.
- Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering.
- Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules.
- Every Queue implementation must specify its ordering properties.

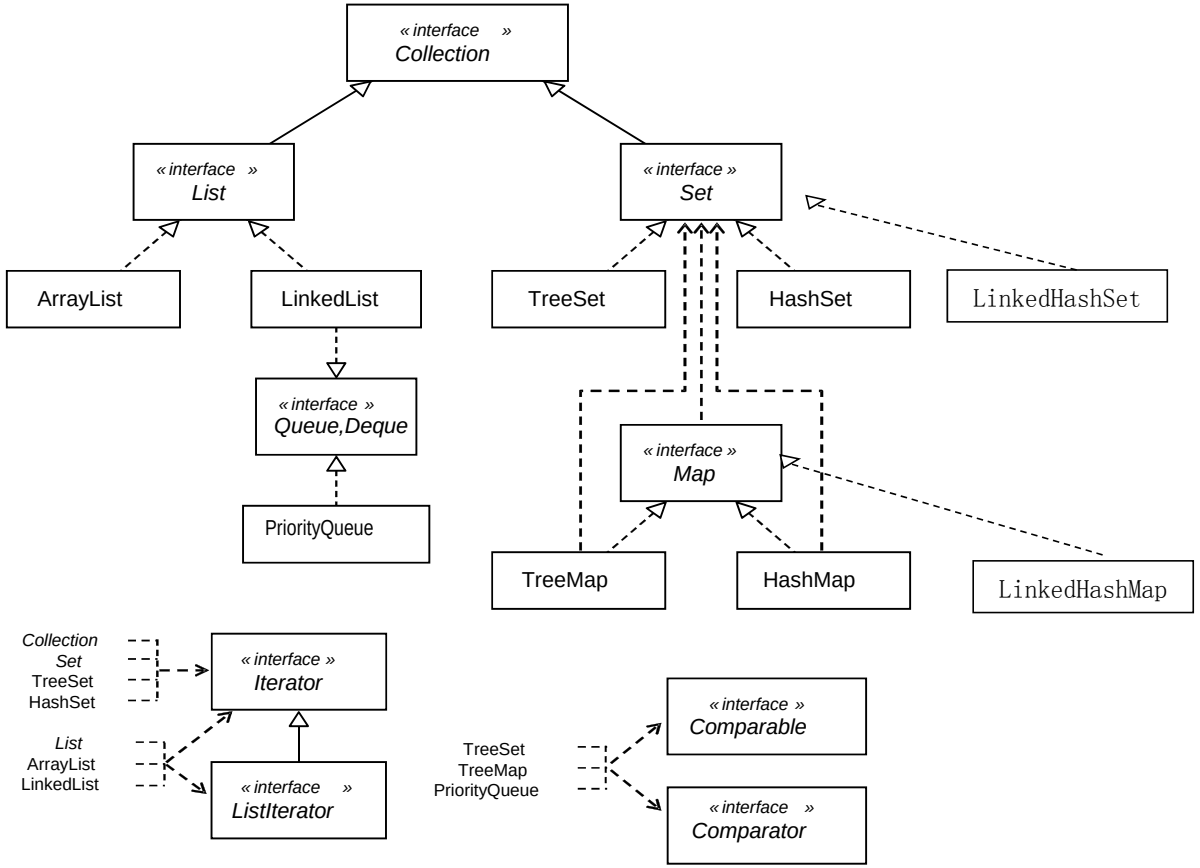
Deque

- Deque — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.
- Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out).
- In a deque all new elements can be inserted, retrieved and removed at both ends.

Map

- Map — an object that maps keys to values.
- A Map cannot contain duplicate keys; each key can map to at most one value.
- SortedMap — a Map that maintains its mappings in ascending key order.
  - This is the Map analog of SortedSet.
  - Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

Collection Interface and Implementation

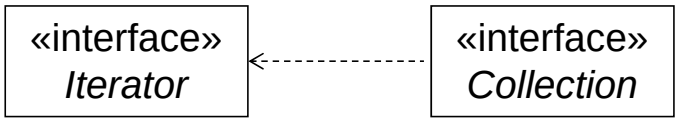


Collection Interface and Implementation

- *Collection*, *Iterator*
- Lists, *ListIterator*
  - *List*
  - ArrayList
  - LinkedList
- *Queue*, *Deque*, PriorityQueue
- Sets
  - *Set*
  - TreeSet
  - HashSet
  - LinkedHashSet
- Maps
  - *Map*
  - TreeMap
  - HashMap
  - LinkedHashMap

All these interfaces and classes are part of the **java.util** package. Names of interfaces are in italics.

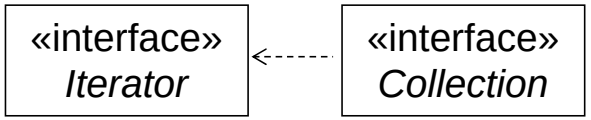
Collection, Iterator



- *Collection* interface represents any collection.
- An iterator is an object that helps to traverse the collection (process all its elements in sequence).
- A collection supplies its own iterator(s), (returned by collection's iterator method); the traversal sequence depends on the collection.

Collection<E> Methods

```
boolean isEmpty ();
int size ();
void clear();
boolean contains (Object obj);
boolean add (E e);
boolean addAll(Collection<? extends E> c)
boolean remove(Object o)
Iterator<E> iterator ();
boolean containsAll(Collection<?> c)
boolean removeAll(Collection<?> c) ;
boolean retainAll(Collection<?> c);
Object[] toArray() ;
//   ... other methods
```



Supplies an iterator for this collection

Example

```
import java.util.*;

public class BasicContainer {
   public static void main(String[] args) {
      Collection c = new HashSet();
      c.add("hello");
      c.add(new Name("f1","l1"));
      c.add(new Integer(100));
      c.remove("hello");
      c.remove(new Integer(100));
      System.out.println(c.remove(new Name("f1","l1")));
      System.out.println(c);
   }
}
```

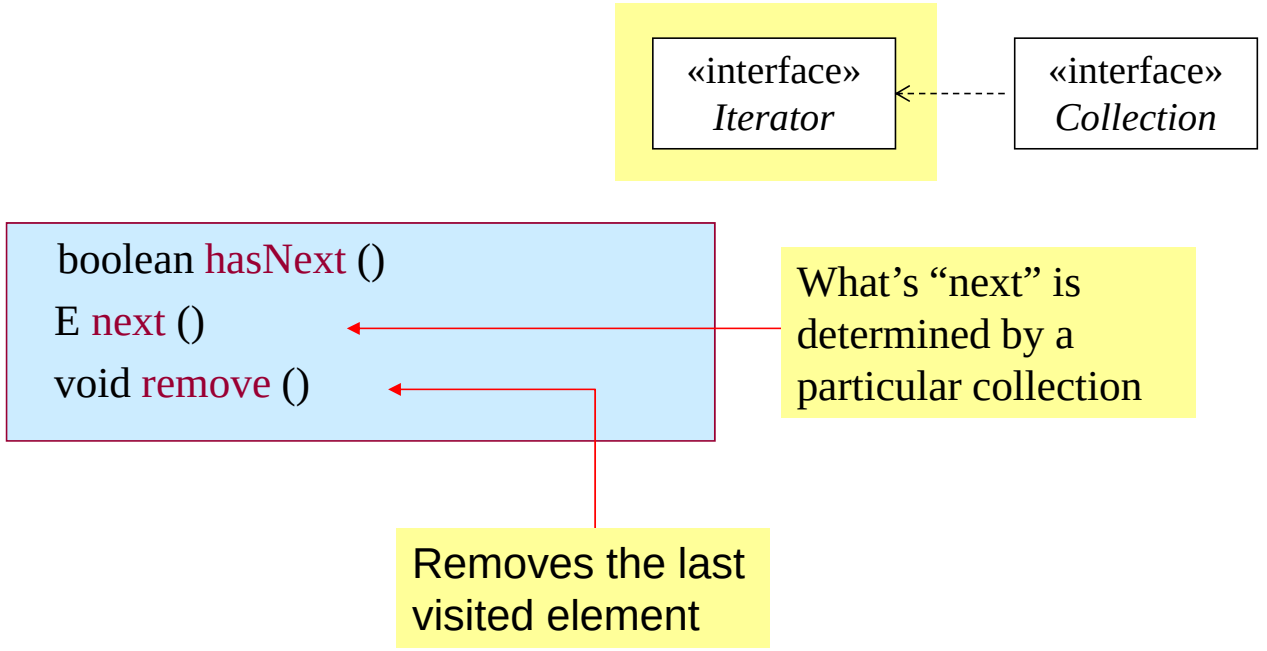
Example

```
class Name implements Comparable {
   private String firstName,lastName;
   public Name(String firstName, String lastName) {
      this.firstName = firstName; this.lastName = lastName;
   }

   public String getFirstName() {  return firstName;   }
   public String getLastName() {   return lastName;   }
   public String toString() {  return firstName + " " + lastName;  }

   public boolean equals(Object obj) {
         if (obj instanceof Name) {
            Name name = (Name) obj;
            return (firstName.equals(name.firstName)) && lastName.equals(name.lastName));
         }
         return super.equals(obj);
  }

  public int hashCode() {
         return firstName.hashCode();
  }
```

Example

```
   public int compareTo(Object o) {
      Name n = (Name)o;
      int lastCmp =
         lastName.compareTo(n.lastName);
      return
          (lastCmp!=0 ? lastCmp : firstName.compareTo(n.firstName));
   }
}
```

Iterator<E> Methods



Iterator ⇔ “For Each” Loop

```
Collection<String> words = new ArrayList<String>();
...
```

```
for (String word : words)
{
    < ... process word >
}
```

```
iterator<String> iter =words.iterator();
while (iter.hasNext ())
{
    String word = iter.next ();
    < ... process word >
}
```

A “for each” loop is a syntactic shortcut that replaces an iterator

for-each Construct

- The for-each construct allows you to concisely traverse a collection or array using a for loop.
- The following code uses the for-each construct to print out each element of a collection on a separate line.

```
for (Object o : collection) System.out.println(o);
```

Iterators

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively.
- The remove method removes the last element that was returned by next from the underlying Collection.
- The remove method may be called only once per call to next and throws an exception if this rule is violated.
- Note that Iterator.remove is the *only* safe way to modify a collection during iteration.
- The behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.
  - for(int i=0;i<list.size();i++){
    list.remove(...);
    }

Iterators

```
                    … … …
Collection c = new HashSet();
c.add(new Name("fff1","lll1"));
c.add(new Name("f2","l2"));
c.add(new Name("fff3","lll3"));
for(Iterator i = c.iterator();i.hasNext();) {
   Name name =(Name)i.next();
   if(name.getFirstName().length()<3){
      i.remove();
      //如果换成 c.remove(name); 会产生例外
   }
}
System.out.println(c);
```

Output：

```
[fff3 lll3, fff1 lll1]
```

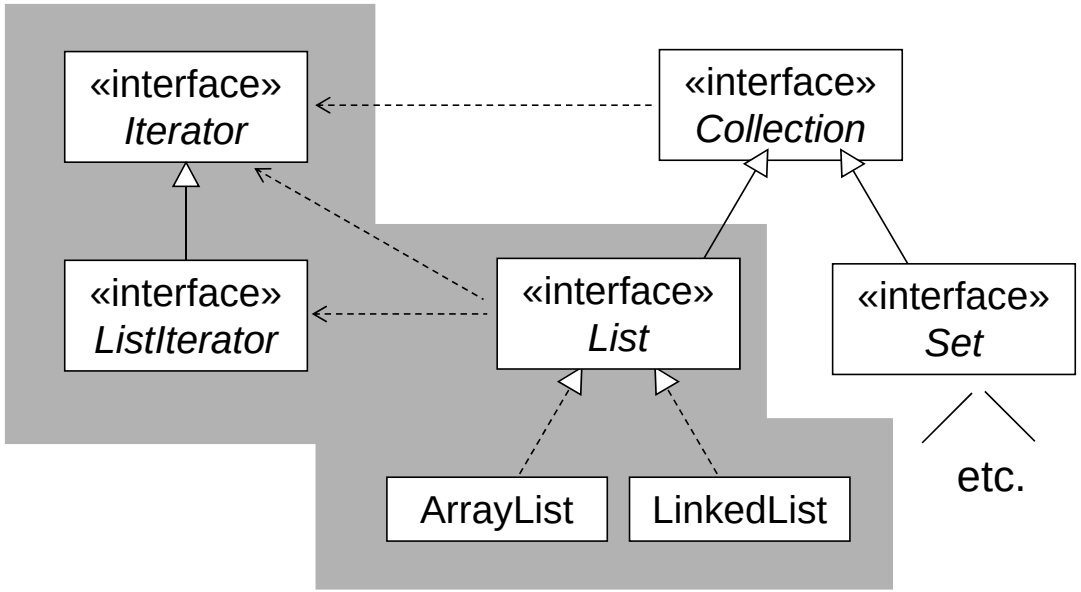
Lists, *ListIterator*

- A list represents a collection in which all elements are numbered by indices:

  $a_0, a_1, ..., a_{n-1}$

- java.util:
  - *List* interface
  - ArrayList
  - LinkedList
- *ListIterator* is an extended iterator, specific for lists (*ListIterator* is a subinterface of *Iterator*).

Lists (cont'd)

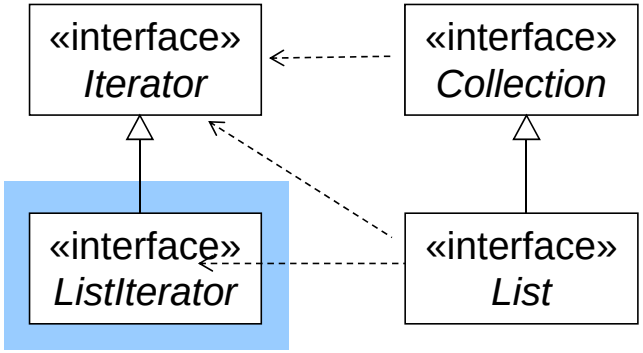

«interface»
Iterator
«interface»
Collection
«interface»
ListIterator
«interface»
List
«interface»
Set
etc.
ArrayList
LinkedList

List<E> Methods

```
// All Collection<E> methods, plus:
E get (int index);
E set (int index, E obj);
void add (int index, E obj);
E remove (int index);
int indexOf (Object obj);
int lastindexOf (Object obj);
ListIterator<E> listIterator ()
ListIterator<E> listIterator (int index)
```

Returns a **ListIterator** that starts iterations at index **i**

ListIterator<E> Methods



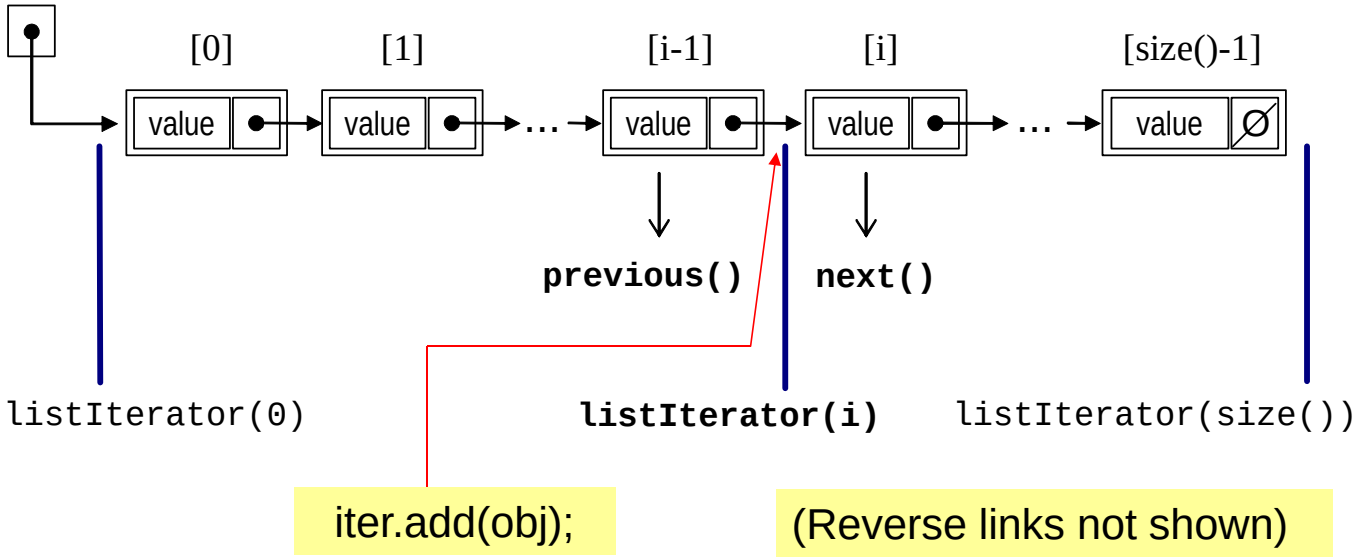

```
// The three Iterator<E> methods, plus:
int nextIndex ()
boolean hasPrevious ()
E previous ()
int previousIndex ()
void add (E obj)
void set (E obj)
```

Can traverse the list backward

Can add elements to the list (inserts after the last visited element)

Can change elements (changes the last visited element)

ListIterator "Cursor" Positioning



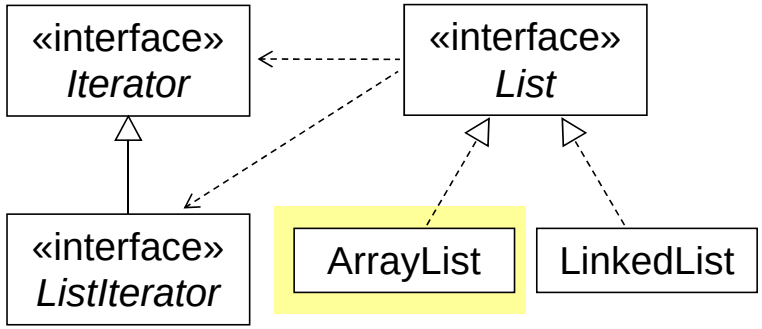
Example

```
package linkedList;
import java.util.*;
public class LinkedListTest
{
  public static void main(String[] args)
  {
        List<String> a = new LinkedList<>();
        a.add("Amy");
        a.add("Carl");
        a.add("Erica");

        List<String> b = new LinkedList<>();
        b.add("Bob");
        b.add("Doug");
        b.add("Frances");
        b.add("Gloria");

        // merge the words from b into a
        ListIterator<String> aIter = a.listIterator();
        Iterator<String> bIter = b.iterator();
```

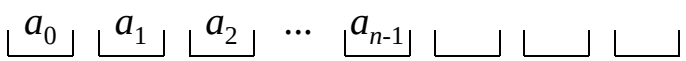
Example

```
      while (bIter.hasNext())
      {
        if (aIter.hasNext()) aIter.next();
        aIter.add(bIter.next());
      }
      System.out.println(a);
      // remove every second word from b
      bIter = b.iterator();
      while (bIter.hasNext())
      { bIter.next(); // skip one element
        if (bIter.hasNext())
        {
          bIter.next();      // skip next element
          bIter.remove(); // remove that element
        }
      }
      System.out.println(b);
      // bulk operation: remove all words in b from a
      a.removeAll(b);
      System.out.println(a);
   }
}
```

ArrayList



- Represents a list as a *dynamic array* (array that is resized when full)
- Provides *random access* to the elements



- Implements all the methods of *List*<*E*>

ArrayList-Example

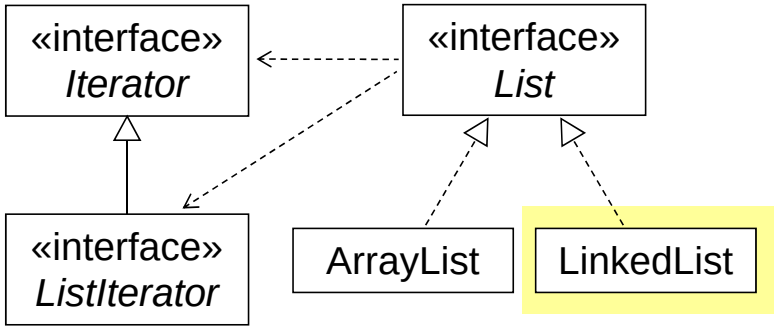
```
import java.util.*;
public class ArrayListExample {
   public static void main(String args[]) {
      /*Creation of ArrayList: I'm going to add String *elements */
      ArrayList<String> obj = new ArrayList<String>();
      /*This is how elements should be added to the array list*/
      obj.add("Ajeet"); obj.add("Harry"); obj.add("Chaitanya");
      obj.add("Steve"); obj.add("Anuj");
       /* Displaying array list elements */
      System.out.println("Currently the array list has following elements:"+obj);
       /*Add element at the given index*/
      obj.add(0, "Rahul"); obj.add(1, "Justin");
       /*Remove elements from array list like this*/
      obj.remove("Chaitanya"); obj.remove("Harry");
      System.out.println("Current array list is:"+obj);
      /*Remove element from the given index*/
      obj.remove(1);
      System.out.println("Current array list is:"+obj);
   }
}
```

ArrayList-Example

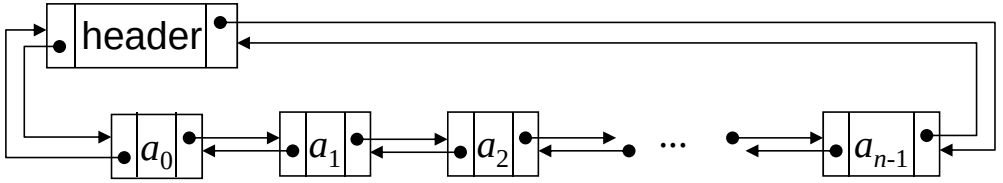
Output:

```
Currently the array list has following elements:[Ajeet, Harry, Chaitanya, Steve, Anuj]
Current array list is:[Rahul, Justin, Ajeet, Steve, Anuj]
Current array list is:[Rahul, Ajeet, Steve, Anuj]
```

LinkedList



- Represents a list as a doubly-linked list with a header node



- Implements all the methods of *List*<*E*>

LinkedList

- Additional methods specific to LinkedList:

```
void addFirst (E obj)
void addLast (E obj)
E getFirst ()
E getLast ()
E removeFirst ()
E removeLast ()
```

Example

```
List l1 = new LinkedList();
for(int i=0; i<=5; i++) {
    l1.add("a"+i);
}
System.out.println(l1);
l1.add(3,"a100");
System.out.println(l1);
l1.set(6,"a200");
System.out.println(l1);
System.out.print((String)l1.get(2)+ " ");
System.out.println(l1.indexOf("a3"));
l1.remove(1);
System.out.println(l1);
```

Output:

```
[a0, a1, a2, a3, a4, a5]
[a0, a1, a2, a100, a3, a4, a5]
[a0, a1, a2, a100, a3, a4, a200]
a2 4
[a0, a2, a100, a3, a4, a200]
```

ArrayList vs. LinkedList

- Implements a list as an array
- Provides *random access* to the elements
- Inserting and removing elements requires shifting of subsequent elements
- Needs to be resized when runs out of space

- Implements a list as a doubly-linked list with a header node
- No random access to the elements — needs to traverse the list to get to the *i*-th element
- Inserting and removing elements is done by rearranging the links — no shifting
- Nodes are allocated and released as necessary

ArrayList vs. LinkedList (cont'd)

	ArrayList	LinkedList
get(i) and **set(i, obj)**	***O*(1)**	*O*(*n*)
add(i, obj) and **remove(i)**	*O*(*n*)	*O*(*1*)
add(0, obj)	*O*(*n*)	***O*(1)**
add(obj)	***O*(1)**	***O*(1)**
contains(obj)	*O*(*n*)	*O*(*n*)

ArrayList vs. LinkedList (cont'd)

```
for (int i = 0;  i < list.size();  i++)
{
   Object x = list.get (i);
   ...
}
```

Works well for an **ArrayList** — $O(n)$
inefficient for a **LinkedList** — $O(n^2)$

```
Iterator iter = list.iterator ( );
while (iter.hasNext ( ))
{
   Object x = iter.next ( );
   ...
}
```

```
for (Object x : list)
{
   ...
}
```

Work well for <u>both</u>
an **ArrayList** and a **LinkedList** — $O(n)$

List Algorithms

Most algorithms in the Collections class apply specifically to List.

- sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- shuffle — randomly permutes the elements in a List.
- reverse — reverses the order of the elements in a List.
- rotate — rotates all the elements in a List by a specified distance.
- swap — swaps the elements at specified positions in a List.
- replaceAll — replaces all occurrences of one specified value with another.

List Algorithms

- fill — overwrites every element in a List with the specified value.
- copy — copies the source List into the destination List.
- binarySearch — searches for an element in an ordered List using the binary search algorithm.
- indexOfSubList — returns the index of the first sublist of one List that is equal to another.
- lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.

Example

```
List l1 = new LinkedList();
List l2 = new LinkedList();
for(int i=0; i<=9; i++) { l1.add("a"+i); }
System.out.println(l1);
Collections.shuffle(l1);
System.out.println(l1);
Collections.reverse(l1);
System.out.println(l1);
Collections.sort(l1);
System.out.println(l1);
System.out.println
   (Collections.binarySearch(l1,"a5"));
```

Output :

```
[a0, a1, a2, a3, a4, a5, a6, a7, a8, a9]
[a1, a3, a8, a9, a4, a6, a5, a2, a0, a7]
[a7, a0, a2, a5, a6, a4, a9, a8, a3, a1]
[a0, a1, a2, a3, a4, a5, a6, a7, a8, a9]
5
```

Comparable Interface

- The Comparable interface consists of the following method.

  public interface Comparable<T> { public int compareTo(T o); }

- The compareTo method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object.
- If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

Example

```
List l1 = new LinkedList();
l1.add(new Name("Karl", "M"));
l1.add(new Name("Steven", "Lee"));
l1.add(new Name("John", "O"));
l1.add(new Name("Tom", "M"));
System.out.println(l1);
Collections.sort(l1);
System.out.println(l1);
```

Output :

```
[Karl M, Steven Lee, John O, Tom M]
[Steven Lee, Karl M, Tom M, John O]
```

Implements Comparable

```
class Name implements Comparable {
                    … … …
    public int compareTo(Object o) {
        Name n = (Name)o;
        int lastCmp =
            lastName.compareTo(n.lastName);
        return
             (lastCmp!=0 ? lastCmp :
              firstName.compareTo(n.firstName));
    }
}
```

Queues

- A queue provides temporary storage in the FIFO (First-In-First-Out) manner.
- Useful for dealing with events that have to be processed in order of their arrival.
- java.util:
  - *Queue* interface
  - LinkedList (implements Queue)

Queue<E> Methods

Type of Operation	**Throws exception**	**Returns special value**
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queues (cont'd)

```
import java.util.*;
public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--) queue.add(i);
        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

Deque

- Deque is a double-ended-queue.
- A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points.

Type of operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

Sets

- A set is a collection without duplicate values
- Designed for finding a value quickly
- java.util:
  - *Set* interface
  - TreeSet
  - HashSet
  - LinkedHashSet
- Methods of *Set<E>* are the same as methods of *Collection<E>*

TreeSet<*E*>

- Works with Comparable objects (or takes a comparator as a parameter)
- Implements a set as a *Binary Search Tree*
- contains, add, and remove methods run in $O(\log n)$ time
- Iterator returns elements in ascending order

TreeSet<*E*> — Example

```
import java.util.TreeSet;
public class TreeSetExample {
   public static void main(String args[]) {
      // TreeSet of String Type
     TreeSet<String> tset = new TreeSet<String>();
      // Adding elements to TreeSet<String>
     tset.add("ABC");         tset.add("String");
     tset.add("Test");        tset.add("Pen");
     tset.add("Ink");         tset.add("Jack");
     //Displaying TreeSet
    System.out.println(tset);
  }
}
```

HashSet<*E*>

- Works with objects for which reasonable hashCode and equals methods are defined
- Implements a set as a *hash table*
- contains, add, and remove methods run in $O(1)$ time
- Iterator returns elements in no particular order

Example

```
import java.util.HashSet;
public class HashSetExample {
   public static void main(String args[]) {
     // HashSet declaration
     HashSet<String> hset = new HashSet<String>();
     // Adding elements to the HashSet
     hset.add("Apple");      hset.add("Mango");
     hset.add("Grapes");    hset.add("Orange");
     hset.add("Fig");
     //Addition of duplicate elements hset.add("Apple");
     hset.add("Mango");
     //Addition of null values hset.add(null);
    hset.add(null);
    //Displaying HashSet elements
    System.out.println(hset);
}
```

[null, Mango, Grapes, Apple, Orange, Fig]

LinkedHashSet

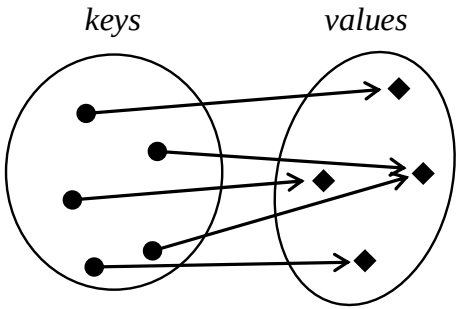
- LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).
- LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Example

```
import java.util.LinkedHashSet;
public class LinkedHashSetExample {
    public static void main(String args[]) {
     // LinkedHashSet of String Type
    LinkedHashSet<String> lhset = new LinkedHashSet<String>();
     // Adding elements to the LinkedHashSet
    lhset.add("Z");
    lhset.add("PQ");
    lhset.add("N");
    lhset.add("O");
    lhset.add("KK");
    lhset.add("FGH");
    System.out.println(lhset);
  }
}
```

Maps

- A *map* is not a collection; it represents a correspondence between a set of keys and a set of values.
- Only one value can correspond to a given key; several keys can be mapped onto the same value.



Maps (cont'd)

- java.util:
  - *Map* interface
  - TreeMap
  - HashMap
  - LinkedHashMAp

Map<K, V> Methods

```
boolean isEmpty ()
int size ()
V get (K key)
V put (K key, V value)
V remove (K key)
boolean containsKey (Object key)
Set<K> keySet ()
Set<Map.Entry<K,V>>entrySet()
```

Returns the set of all keys

Returns a set view of the mappings conatined in this map

TreeMap<*K*,*V*>

- Works with Comparable keys (or takes a comparator as a parameter)
- Implements the key set as a *Binary Search Tree*
- containsKey, get, and put methods run in $O(\log n)$ time

Example

```
import java.util.TreeMap;  import java.util.Set;
import java.util.Iterator;   import java.util.Map;
public class Details {
   public static void main(String args[]) {
      /* This is how to declare TreeMap */
      TreeMap<Integer, String> tmap = new TreeMap<Integer, String>();
      /*Adding elements to TreeMap*/
      tmap.put(1, "Data1");   tmap.put(23, "Data2");    tmap.put(70, "Data3");
      tmap.put(4, "Data4");   tmap.put(2, "Data5");
      /* Display content using Iterator*/
      Set set = tmap.entrySet();
      Iterator iterator = set.iterator();
      while(iterator.hasNext()) {
         Map.Entry mentry = (Map.Entry)iterator.next();
         System.out.print("key is: "+ mentry.getKey() + " & Value is: ");
         System.out.println(mentry.getValue());
      }
   }
}
```

Example

Output:

key is: 1 & Value is: Data1

key is: 2 & Value is: Data5

key is: 4 & Value is: Data4

key is: 23 & Value is: Data2

key is: 70 & Value is: Data3

HashMap<*K*,*V*>

- Works with keys for which reasonable hashCode and equals methods are defined
- Implements the key set as a *hash table* containsKey, get, and put methods run in $O(1)$ time

Example

```
package beginnersbook.com;
import java.util.HashMap; import java.util.Map;
import java.util.Iterator; import java.util.Set;
public class Details {
    public static void main(String args[]) {
      /* This is how to declare HashMap */
      HashMap<Integer, String> hmap = new HashMap<Integer, String>();
      /*Adding elements to HashMap*/
      hmap.put(12, "Chaitanya");    hmap.put(2, "Rahul");  hmap.put(7, "Singh");
      hmap.put(49, "Ajeet");          hmap.put(3, "Anuj");
      /* Display content using Iterator*/
     Set set = hmap.entrySet(); Iterator iterator = set.iterator();
     while(iterator.hasNext()) {
         Map.Entry mentry = (Map.Entry)iterator.next();
         System.out.print("key is: "+ mentry.getKey() + " & Value is: ");
         System.out.println(mentry.getValue());
      }
```

Example

```
    /* Get values based on key*/
    String var= hmap.get(2);
    System.out.println("Value at index 2 is: "+var);
    /* Remove values based on key*/
    hmap.remove(3);
    System.out.println("Map key and values after removal:");
    Set set2 = hmap.entrySet();
    Iterator iterator2 = set2.iterator();
    while(iterator2.hasNext()) {
       Map.Entry mentry2 = (Map.Entry)iterator2.next();
       System.out.print("Key is: "+mentry2.getKey() + " & Value is: ");
       System.out.println(mentry2.getValue());
   }
  }
}
```

Example

Output:

```
key is: 49 & Value is: Ajeet
key is: 2 & Value is: Rahul
key is: 3 & Value is: Anuj
key is: 7 & Value is: Singh
key is: 12 & Value is: Chaitanya
Value at index 2 is: Rahul
Map key and values after removal:
Key is: 49 & Value is: Ajeet
Key is: 2 & Value is: Rahul
Key is: 7 & Value is: Singh
Key is: 12 & Value is: Chaitanya
```

LinkedHashMap

- LinkedHashMap is a Hash table and linked list implementation of the Map interface, with predictable iteration order.
- This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
- This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

Example

```
import java.util.LinkedHashMap;   import java.util.Set;
import java.util.Iterator;                import java.util.Map;
public class LinkedHashMapDemo {
    public static void main(String args[]) {
      // HashMap Declaration
      LinkedHashMap<Integer, String> lhmap = new LinkedHashMap<Integer, String>();
      //Adding elements to LinkedHashMap
      lhmap.put(22, "Abey"); lhmap.put(33, "Dawn"); lhmap.put(1, "Sherry");
      lhmap.put(2, "Karon"); lhmap.put(100, "Jim");
      // Generating a Set of entries
     Set set = lhmap.entrySet();
      // Displaying elements of LinkedHashMap
      Iterator iterator = set.iterator();
      while(iterator.hasNext()) {
          Map.Entry me = (Map.Entry)iterator.next();
          System.out.print("Key is: "+ me.getKey() + "& Value is: "+me.getValue()+"\n");
      }
    }
}
```

Example

```
import java.util.*;
public class Freq {
   public static void main(String[] args) {
      Map<String, Integer> m = new HashMap<String, Integer>();
      // Initialize frequency table from command line
      for (String a : args) {
          Integer freq = m.get(a);
          m.put(a, (freq == null) ? 1 : freq + 1);
      }
       System.out.println(m.size() + " distinct words:");
       System.out.println(m);
  }
}
```

Example

- Run this program with the command:

  java Freq if it is to be it is up to me to delegate

- The program yields the following output.

  8 distinct words: {to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

Review

- Why Java collections are called “generic”?
- Name several methods of Collection.
- What is an iterator?
- How can we obtain an iterator for a given collection?
- Guess what happens when we call iter.next() when there is no next element.

Review (cont'd)

- What are the properties of a list?
- Name the key methods of the List interface.
- How is ArrayList implemented?
- How is LinkedList implemented?

Review (cont'd)

- Name a few methods specific to LinkedList.
- Name a few methods specific to ListIterator.
- Can you start iterations at any given position in a list?
- How is a set different from a list?
- Name a few methods of the Set interface.

Review (cont'd)

- What is the order of values returned by a TreeSet iterator?
- What is a map?
- In a map, can the same key be associated with several different values?