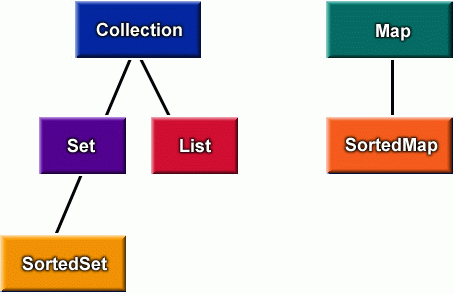
**Collection Framework**



The primary advantages of a collections framework are that it:

* **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
* **Increases performance** by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.
* **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
* **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
* **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
* **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

The collections framework consists of:

1. Collection Interfaces - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
2. General-purpose Implementations - Primary implementations of the collection interfaces.
3. Legacy Implementations - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
4. Wrapper Implementations - Add functionality, such as synchronization, to other implementations.
5. Convenience Implementations - High-performance "mini-implementations" of the collection interfaces.
6. Abstract Implementations - Partial implementations of the collection interfaces to facilitate custom implementations.
7. Algorithms - Static methods that perform useful functions on collections, such as sorting a list.
8. Infrastructure - Interfaces that provide essential support for the collection interfaces.
9. Array Utilities - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.

**Collection Interfaces:**

* Collections that do not support any modification operations (such as add, remove and clear) are referred to as *unmodifiable*. Collections that are not unmodifiable are referred to *modifiable.*
* Collections that additionally guarantee that no change in the Collection object will ever be visible are referred to as *immutable*. Collections that are not immutable are referred to as *mutable*.
* Lists that guarantee that their size remains constant even though the elements may change are referred to as *fixed-size*. Lists that are not fixed-size are referred to as *variable-size*.

**Collection Implementation**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **Implementations** | | | |
| **Hash Table** | **Resizable Array** | **Balanced Tree** | **Linked List** |
| **Interfaces** | **Set** | [HashSet](F:APIsJ2SDK1.4.0docsapijavautilHashSet.html) |  | [TreeSet](F:APIsJ2SDK1.4.0docsapijavautilTreeSet.html) |  |
| **List** |  | [ArrayList](F:APIsJ2SDK1.4.0docsapijavautilArrayList.html) |  | [LinkedList](F:APIsJ2SDK1.4.0docsapijavautilLinkedList.html) |
| **Map** | [HashMap](F:APIsJ2SDK1.4.0docsapijavautilHashMap.html) |  | [TreeMap](F:APIsJ2SDK1.4.0docsapijavautilTreeMap.html) |  |

The general-purpose implementations support all of the *optional operations* in the collection interfaces, and have no restrictions on the elements they may contain. They are unsynchronized, but the Collections class contains static factories called [synchronization wrappers](F:APIsJ2SDK1.4.0docsapijavautilCollections.html#synchronizedCollection(java.util.Collection)) that may be used to add synchronization to any unsynchronized collection. All of the new implementations have *fail-fast iterators*, which detect illegal concurrent modification, and fail quickly and cleanly (rather than behaving erratically).

The AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList and AbstractMap classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them. The API documentation for these classes describes precisely how each method is implemented so the implementer knows which methods should be overridden, given the performance of the "basic operations" of a specific implementation.

Annotated outline of Collection Framework

* **Collection Interfaces** - The primary means by which collections are manipulated.
  + [**Collection**](F:APIsJ2SDK1.4.0docsapijavautilCollection.html) - A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.
  + [**Set**](F:APIsJ2SDK1.4.0docsapijavautilSet.html) - The familiar set abstraction. No duplicate elements permitted. May or may not be ordered. Extends the Collection interface.
  + [**List**](F:APIsJ2SDK1.4.0docsapijavautilList.html) - Ordered collection, also known as a *sequence*. Duplicates are generally permitted. Allows positional access. Extends the Collection interface.
  + [**Map**](F:APIsJ2SDK1.4.0docsapijavautilMap.html) - A mapping from keys to values. Each key can map to at most one value.
  + [**SortedSet**](F:APIsJ2SDK1.4.0docsapijavautilSortedSet.html) - A set whose elements are automatically sorted, either in their *natural ordering* (see the [Comparable](F:APIsJ2SDK1.4.0docsapijavalangComparable.html) interface), or by a [Comparator](F:APIsJ2SDK1.4.0docsapijavautilComparator.html) object provided when a SortedSet instance is created. Extends the Set interface.
  + [**SortedMap**](F:APIsJ2SDK1.4.0docsapijavautilSortedMap.html) - A map whose mappings are automatically sorted by key, either in the keys' *natural ordering* or by a comparator provided when a SortedMap instance is created. Extends the Map interface.

**[ArrayList](F:\APIs\J2SDK1.4.0\docs\api\java\util\ArrayList.html)** - Resizable-array implementation of the List interface. Implements all optional list operations, and permit all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

**Constructors:**

**ArrayList()**

**ArryaList(Collection c)**

**ArrayList(int initialsize)**

//program that demonstrates ArrayList and Iterator

import java.util.\*;

class arraylistdemo

{

public static void main(String args[])

{

ArrayList al=new ArrayList();

al.add("a");

al.add("b");

al.add("c");

al.add("d");

al.add("e");

al.add("f");

al.add("g");

al.add(new Integer(90));

al.add(new Double(88.5));

Collection c = al;

ArrayList a2 = new ArrayList(c);

System.out.println("The elements of the array list are:"+al);

al.remove("f");

al.remove(2);

Iterator itr=al.iterator();

System.out.println("The elements of the array list after removal:");

while(itr.hasNext())

{

System.out.println(itr.next());

}

}

}

**Output**:

The elements of the array list are:[a, b, c, d, e, f, g]

The elements of the array list after removal:

a

b

d

e

g

[**LinkedList**](F:APIsJ2SDK1.4.0docsapijavautilLinkedList.html) - Doubly-linked list implementation of the List interface. May provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list. Useful for queues and double-ended queues (deques).

**Constructors:**

**LinkedList()**

**LinkedList(Collection c)**

//program that demonstrates LinkedList & ListIterator

import java.util.\*;

class linkedlistdemo

{

public static void main(String args[])

{

LinkedList ll=new LinkedList();

ll.add("a");

ll.add("b");

ll.add("c");

ll.add("d");

ll.add("e");

ll.addFirst("f");

ll.addLast("g");

System.out.println("The elements of the LinkedList in forward direction:");

ListIterator li=ll.listIterator();

while(li.hasNext())

{

System.out.println(li.next());

}

System.out.println("The elements of the LinkedList in reverse direction:");

while(li.hasPrevious())

{

System.out.println(li.previous());

}

}

}

**Output:**

The elements of the LinkedList in forward direction:

f

a

b

c

d

e

g

The elements of the LinkedList in reverse direction:

g

e

d

c

b

a

f

**HashtSet:** This class implements 'Set' interface. It uses Hashtable to store the elements into it. It is not an efficient Set class i.e. there is no guarantee of order of the elements.

**Constructors:**

**HashSet()**

**HashSet(Collection c)**

**HashSet(int initialsize)**

//program that demonstrates HashSet

import java.util.\*;

class hashsetdemo

{

public static void main(String args[])

{

HashSet hs=new HashSet();

hs.add("a");

hs.add("b");

hs.add("c");

hs.add("d");

hs.add("e");

hs.add("f");

System.out.println("the elements of the HashSet are:");

Iterator itr=hs.iterator();

while(itr.hasNext())

{

System.out.println(itr.next());

}

}

}

**Output:**

the elements of the HashSet are:

d

a

c

f

b

e

[**TreeSet**](F:APIsJ2SDK1.4.0docsapijavautilTreeSet.html) Red-black tree implementation of the SortedSet interface. This class implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the *natural order* of the elements (see Comparable), or by the comparator provided at set creation time, depending on which constructor is used.

**Constructors:**

**TreeSet()**

**TreeSet(Collection c)**

**TreeSet(int initialsize)**

//program that demonstrates TreeSet

import java.util.\*;

class treesetdemo

{

public static void main(String args[])

{

TreeSet ts=new TreeSet();

ts.add("x");

ts.add("b");

ts.add("d");

ts.add("c");

ts.add("p");

ts.add("l");

ts.add("m");

System.out.println("the elements of the TreeSet are:");

Iterator itr=ts.iterator();

while(itr.hasNext())

{

System.out.println(itr.next());

}

}

}

**Output:**

the elements of the TreeSet are:

b

c

d

l

m

p

x

**LinkedHashSet:** This class introduced from j2sdk1.4.0 version onwards. This class implements 'List' interface and extends 'HashSet' class. This class arranges the elements in the order in which you inserti.e. original order.

**Constructors:**

**LinkedHashSet()**

**LinkedHashSet(Collection c)**

**LinkedHashSet(int initialsize)**

//program that demonstrates LinkedHashSet

import java.util.\*;

class linkedhashsetdemo

{

public static void main(String args[])

{

LinkedHashSet lhs=new LinkedHashSet();

lhs.add("a");

lhs.add("x");

lhs.add("p");

lhs.add("c");

lhs.add("o");

lhs.add("q");

lhs.add("A");

System.out.println("The elements of the LinkeHashSet are:");

Iterator itr=lhs.iterator();

while(itr.hasNext())

{

System.out.println(itr.next());

}

}

}

**Output:**

The elements of the LinkeHashSet are:

a

x

p

c

o

q

A

[**HashMap**](F:APIsJ2SDK1.4.0docsapijavautilHashMap.html) - Hash table implementation of the Map interface. (Essentially an unsynchronized Hashtable that supports null keys and values.) The best all-around implementation of the Map interface.

**Constructors:**

**HashMap()**

**HashMap(Map m)**

**HashMap(int capacity)**

**HashMap(int capacity,float fillratio)**

//program that demonstrates TreeMap

import java.util.\*;

class treemapdemo

{

public static void main(String args[])

{

TreeMap tm=new TreeMap();

tm.put("Allen",new Double(5000.0));

tm.put("Alex",new Double(10000.0));

tm.put("John",new Double(7000.0));

tm.put("vandam",new Double(3000.0));

tm.put("Robert",new Double(2000.0));

Set s=tm.entrySet();

Iterator itr=s.iterator();

while(itr.hasNext())

{

Map.Entry me=(Map.Entry)itr.next();

System.out.println("Employee Name:"+me.getKey()+" Salary:"+me.getValue());

}

}

}

**Output:**

F:\ADVJAVA\collection>java hashmapdemo

Employee Name:Alex Salary:10000.0

Employee Name:Robert Salary:2000.0

Employee Name:vandam Salary:3000.0

Employee Name:John Salary:7000.0

Employee Name:Allen Salary:5000.0

[**TreeSet**](F:APIsJ2SDK1.4.0docsapijavautilTreeSet.html) Red-black tree implementation of the SortedSet interface. This class implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the *natural order* of the elements (see Comparable), or by the comparator provided at set creation time, depending on which constructor is used.

**Constructors:**

**TreeMap()**

**TreeMap(Map m)**

//program that demonstrates TreeMap

import java.util.\*;

class treemapdemo

{

public static void main(String args[])

{

TreeMap tm=new TreeMap();

tm.put("Allen",new Double(5000.0));

tm.put("Alex",new Double(10000.0));

tm.put("John",new Double(7000.0));

tm.put("vandam",new Double(3000.0));

tm.put("Robert",new Double(2000.0));

Set s=tm.entrySet();

Iterator itr=s.iterator();

while(itr.hasNext())

{

Map.Entry me=(Map.Entry)itr.next();

System.out.println("Employee Name: "+me.getKey()+" Salary:"+me.getValue());

}

}

}

**Output:**

F:\ADVJAVA\collection>java treemapdemo

Employee Name:Alex Salary:10000.0

Employee Name:Allen Salary:5000.0

Employee Name:John Salary:7000.0

Employee Name:Robert Salary:2000.0

Employee Name:vandam Salary:3000.0

**LinkedHashMap:** This class introduced from j2sdk1.4.2 version onwads. This class implements 'Map' interface and extends HashMap class. LinkedHashMap provides additional functionality to HashMap class. This class arranges the elements in the original order i.e. the order in which you insert.

**Constructors:**

**LinkedHashMap()**

**LinkedHashMap(Map m)**

**LinkedHashMap(int capacity,float fillratio)**

//program that demonstrates LinkedHashMap

import java.util.\*;

class linkedhashmapdemo

{

public static void main(String args[])

{

LinkedHashMap lhm=new LinkedHashMap();

lhm.put("Allen",new Double(5000.0));

lhm.put("Alex",new Double(10000.0));

lhm.put("John",new Double(7000.0));

lhm.put("vandam",new Double(3000.0));

lhm.put("Robert",new Double(2000.0));

Set s=lhm.entrySet();

Iterator itr=s.iterator();

while(itr.hasNext())

{

Map.Entry me=(Map.Entry)itr.next();

System.out.println("Employee Name:"+me.getKey()+" Salary:"+me.getValue());

}

}

}

**Output:**

F:\ADVJAVA\collection>java linkedhashmapdemo

Employee Name:Allen Salary:5000.0

Employee Name:Alex Salary:10000.0

Employee Name:John Salary:7000.0

Employee Name:vandam Salary:3000.0

Employee Name:Robert Salary:2000.0