Collections (Collection Framework)

Sang Shin
Java Technology Architect
Sun Microsystems, Inc.
sang.shin@sun.com
www.javapassion.com

Disclaimer & Acknowledgments

- Even though Sang Shin is a full-time employee of Sun Microsystems, the contents here are created as his own personal endeavor and thus does not necessarily reflect any official stance of Sun Microsystems on any particular technology
- Acknowledgments
  - The contents of this presentation was created from The Java Tutorials in java.sun.com

Topics

- What is and Why Collections?
- Core Collection Interfaces
- Implementations
- Algorithms
- Custom Implementations
- Interoperability

What is a Collection?

What is and Why Collection Framework?

What is a Collection?

- A “collection” object — sometimes called a container — is simply an object that groups multiple elements into a single unit
- Collections are used to store, retrieve, manipulate, and communicate aggregate data
  - Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

What is a Collection Framework?

- A collections framework is a unified architecture for representing and manipulating collections
- All collections frameworks contain the following:
  - Interfaces
  - Implementations
  - Algorithms

Benefits of Collection Framework

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
  - The collection interfaces are the vernacular by which APIs pass collections back and forth
- Reduce effort to learn and use new APIs
- Reduces effort to design new APIs
- Fosters software reuse
  - New data structures that conform to the standard collection interfaces are by nature reusable

Interfaces & Implementations in Collection Framework

Interfaces

- Collection interfaces are abstract data types that represent collections
  - Collection interfaces are in the form of Java interfaces
- Interfaces allow collections to be manipulated independently of the implementation details of their representation
  - Polymorphic behavior
- In Java programming language (and other object-oriented languages), interfaces generally form a hierarchy
  - You choose one that meets your need as a type

Implementations

- These are the concrete implementations of the collection interfaces

Types of Implementations

- General-purpose implementations
- Special-purpose implementations
- Concurrent implementations
- Wrapper implementations
- Convenience implementations
- Abstract implementations

Implementations

- Implementations are the data objects used to store collections, which implement the interfaces
- Each of the general-purpose implementations (you will see in the following slide) provides all optional operations contained in its interface
- Java Collections Framework also provides several special-purpose implementations for situations that require nonstandard performance, usage restrictions, or other unusual behavior

General Purpose Implementations

General-purpose Implementations

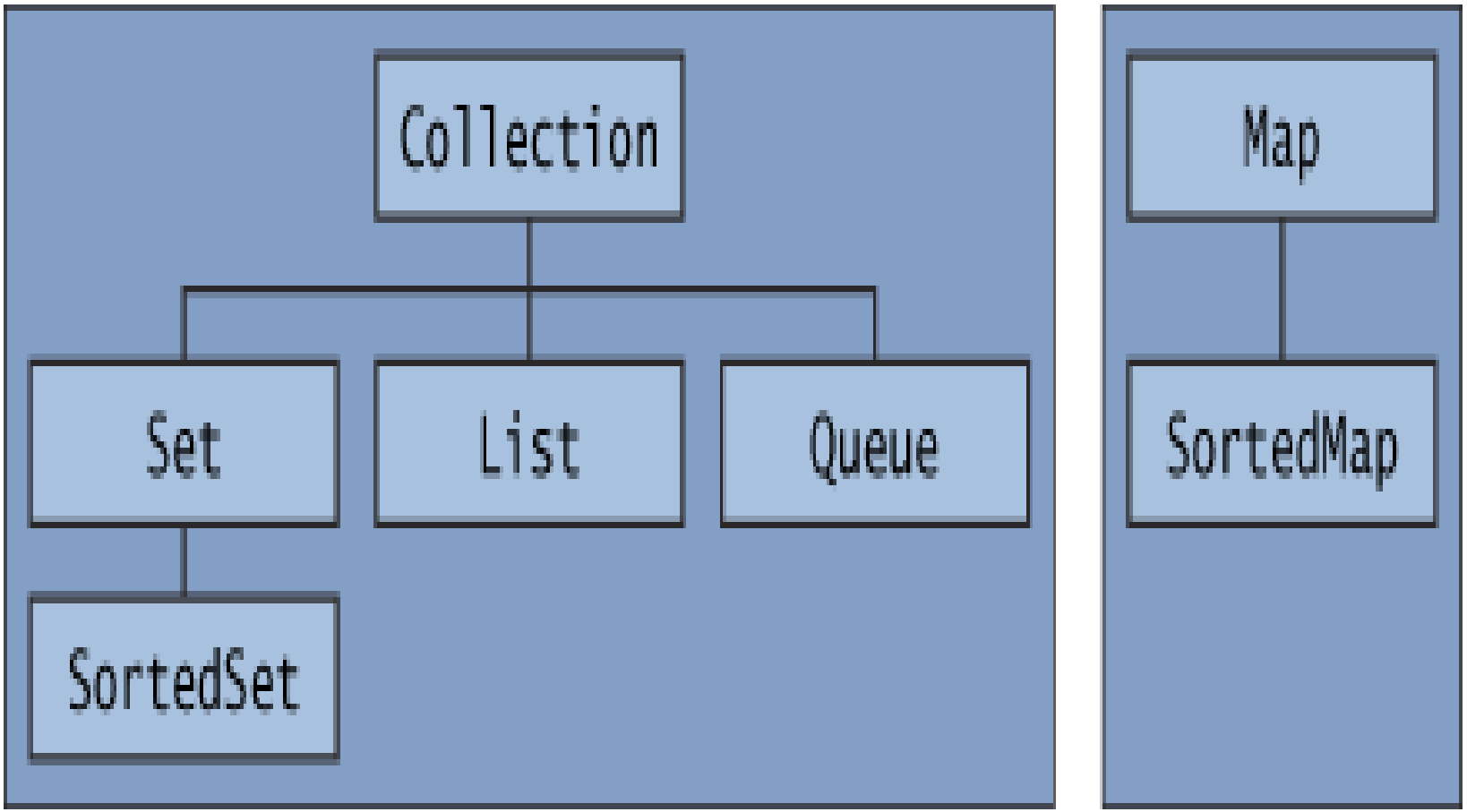
Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

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 collection interface
  - In essence, algorithms are reusable functionality.

Core Collection Interfaces

Core Collection Interfaces Hierarchy



Core Collection Interfaces

- Core collection interfaces are the foundation of the Java Collections Framework
- Core collection interfaces form a inheritance hierarchy among themselves
  - You can create a new Collection interface from them (highly likely you don't have to)

“Collection” Interface

“Collection” Interface

- The root of the collection hierarchy
- Is the least common denominator that all collection implement
  - Every collection object is a type of Collection interface
- Is used to pass collection objects around and to manipulate them when maximum generality is desired
  - Use Collection interface as a type
- JDK doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List

“Collection” Interface (Java SE 5)

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);       //optional
    boolean retainAll(Collection<?> c);       //optional
    void clear();                             //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Example: Usage "Collection" Interface as a Type

```
// Create a ArrayList collection object instance and assign it
// to Collection type.
Collection c1 = new ArrayList();

// Use methods of Collection interface.
//
// Polymorphic behavior is expected. For example,
// the add() implementation of ArrayList class will be
// invoked. For example, depending on the implementation,
// duplication is allowed or not allowed.

boolean b1 = c1.isEmpty();
boolean b2 = c1.add(new Integer(1));
```

“Collection” Interface: add() and remove() Operations

add() and remove() methods of Collection Interface

- The add() method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't
- It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call.
  - add() method of Set interface follows “no duplicate” rule

“Collection” Interface: Traversing

Two Schemes of Traversing Collections

- for-each
  - The for-each construct allows you to concisely traverse a collection or array using a for loop

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

- Iterator
  - An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

Iterator Interface

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); //optional
}
```

- hasNext() method returns true if the iteration has more elements
- next() method returns the next element in the iteration
- 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

Use Iterator over for-each when you need to

- Remove the current element
  - The for-each construct hides the iterator, so you cannot call remove
  - Therefore, the for-each construct is not usable for filtering.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

- Iterate over multiple collections in parallel

“Collection” Interface: Bulk Operations

Bulk Operations

- containsAll() — returns true if the target Collection contains all of the elements in the specified Collection.
- addAll() — adds all of the elements in the specified Collection to the target Collection.
- removeAll() — removes from the target Collection all of its elements that are also contained in the specified Collection.
- retainAll() — removes from the target Collection all its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- clear() — removes all elements from the Collection.

Example: removeAll()

- Remove all instances of a specified element, e, from a Collection, c
  - c.removeAll(Collections.singleton(e));
- Remove all of the null elements from a Collection
  - c.removeAll(Collections.singleton(null));
- Collections.singleton(), which is a static factory method that returns an immutable Set containing only the specified element

“Collection” Interface: Array Operations

Array Operations

- The toArray() method is provided as a bridge between collections and older APIs that expect arrays on input
- The array operations allow the contents of a Collection to be translated into an array.

Example: Array Operations

- The simple form with no arguments creates a new array of Object
  - Object[] a = c.toArray();
- Suppose that c is known to contain only strings. The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.
  - String[] a = c.toArray();

Set Interface & Implementations

“Set” Interface

- A collection that cannot contain duplicate elements
- Models the mathematical set abstraction and is used to represent sets
  - cards comprising a poker hand
  - courses making up a student's schedule
  - the processes running on a machine
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited

“Set” Interface (Java SE 5)

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);       //optional
    boolean retainAll(Collection<?> c);       //optional
    void clear();                             //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

“equals” operation of “Set” Interface

- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ
  - Two Set instances are equal if they contain the same elements

“SortedSet” Interface

- 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 roll

Implementations of “Set” Interface

- HashSet
- TreeSet
- LinkedHashSet

HashSet

- HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees
- Mostly commonly used implementation

Caveats of Using HashSet

- Iteration is linear in the sum of the number of entries and the number of buckets (the capacity)
  - Choosing an initial capacity that's too high can waste both space and time
  - Choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity

Example: Set Interface & HashSet

```
public class MyOwnUtilityClass {

    // Note that the first parameter type is set to
    // Set interface not a particular implementation
    // class such as HashSet.  This makes the caller of
    // this method to pass instances of different
    // implementations of Set interface while
    // this function picks up polymorphic behavior
    // depending on the actual implementation type
    // of the object instance passed.

    public static void checkDuplicate(Set s, String[] args){
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);

        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

Example: Set Interface & HashSet

```
public class Main {

    public static void main(String[] args) {

        Set s = new HashSet();      // Order is not guaranteed
        MyOwnUtilityClass.checkDuplicate(s, args);

        s = new TreeSet();          // Order according to values
        MyOwnUtilityClass.checkDuplicate(s, args);

        s = new LinkedHashSet(); // Order according to insertion
        MyOwnUtilityClass.checkDuplicate(s, args);
    }
}
```

Example: Set Interface & HashSet

The numbers are added in the following order
2
3
4
1
2
Set type = java.util.HashSet [3, 2, 4, 1]
Set type = java.util.TreeSet [1, 2, 3, 4]
Set type = java.util.LinkedHashSet [2, 3, 4, 1]

TreeSet

- When you need to use the operations in the SortedSet interface, or if value-ordered iteration is required

Example: Set Interface & TreeSet

```
public static void main(String[] args) {

    Set ts = new TreeSet();

    ts.add("one");
    ts.add("two");
    ts.add("three");
    ts.add("four");
    ts.add("three");

    System.out.println("Members from TreeSet = " + ts);
}
```

Result:

Members from TreeSet = [four, one, three, two]

LinkedHashSet

- Implemented as a hash table with a linked list running through it
- Provides insertion-ordered iteration (least recently inserted to most recently) and runs nearly as fast as HashSet.
- Spares its clients from the unspecified, generally chaotic ordering provided by HashSet without incurring the increased cost associated with TreeSet

Example: Set Interface & LinkedHashSet

```
public static void main(String[] args) {

    Set ts2 = new LinkedHashSet();

    ts2.add(2);
    ts2.add(1);
    ts2.add(3);
    ts2.add(3);

    System.out.println("Members from LinkedHashSet = " + ts2);

}
```

Result:

Members from LinkedHashSet = [2, 1, 3]

List Interface & Implementations

“List” Interface

- 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)
  - If you've used Vector, you're already familiar with the general basics of List

Additional Operations Supported by "List" Interface over "Collection"

- Positional access — manipulates elements based on their numerical position in the list
- Search — searches for a specified object in the list and returns its numerical position
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature
- Range-view — performs arbitrary range operations on the list.

“List” Interface

```
public interface List<E> extends Collection<E> {
   // Positional access
   E get(int index);
   E set(int index, E element);    //optional
   boolean add(E element);         //optional
   void add(int index, E element); //optional
   E remove(int index);            //optional
   boolean addAll(int index,
      Collection<? extends E> c); //optional

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

   // Range-view
   List<E> subList(int from, int to);
}
```

Implementations of “List” Interface

- ArrayList
  - Offers constant-time positional access
  - Fast
  - Think of ArrayList as Vector without the synchronization overhead
  - Most commonly used implementation
- LinkedList
  - Use it you frequently add elements to the beginning of the List or iterate over the List to delete elements from its interior

Map Interface & Implementations

“Map” Interface

- Handles key/value pairs
- A Map cannot contain duplicate keys; each key can map to at most one value
  - If you've used Hashtable, you're already familiar with the basics of Map.

“Map” Interface (Java SE 5)

```
public interface Map<K,V> {
   // Basic operations
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk operations
   void putAll(Map<? extends K, ? extends V> m);
   void clear();

   // Collection Views
   public Set<K> keySet();
   public Collection<V> values();
   public Set<Map.Entry<K,V>> entrySet();

   // Interface for entrySet elements
   public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
   }
```

“SortedMap” Interface

- 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

Implementations of “Map” Interface

- HashMap
  - Use it you want maximum speed and don't care about iteration order
  - Most commonly used implementation
- TreeMap
  - Use it when you need SortedMap operations or key-ordered Collection-view iteration
- LinkedHashMap
  - Use if you want near-HashMap performance and insertion-order iteration

Queue Interface & Implementations

“Queue” Interface

- A collection used to hold multiple elements prior to processing
- Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations
- Typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner

Implementations of Queue Interface

- General purpose Queue implementations
  - LinkedList implements the Queue interface, providing FIFO queue operations for add, poll, and so on
  - PriorityQueue class is a priority queue based on the heap data structure
    - This queue orders elements according to an order specified at construction time, which can be the elements' natural ordering or the ordering imposed by an explicit Comparator.
- Concurrent Queue implementations

Convenience Implementations

Collections Class

- Provides methods to return the empty Set, List, and Map — emptySet, emptyList, and emptyMap
  - The main use of these constants is as input to methods that take a Collection of values when you don't want to provide any values at all
  - tourist.declarePurchases(Collections.emptySet());
- Provides methods for sorting and shuffling
  - Collections.sort(l);
  - Collections.shuffle(l);

Abstract Classes

Abstract Classes

- Includes abstract implementations
  - AbstractCollection
  - AbstractSet
  - AbstractList
  - AbstractSequentialList
  - AbstractMap
- To aid in custom implementations
  - Reduces code you have to write

Algorithms

Algorithms

- Sorting
- Shuffling
- Routine data manipulation
- Searching
- Composition
- Find extreme values

Algorithms

- Provided as static methods of Collections class
- The first argument is the collection on which the operation is to be performed
- The majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on arbitrary Collection instances

Sorting

- The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship
- Two forms of the operations
  - takes a List and sorts it according to its elements' natural ordering
  - takes a Comparator in addition to a List and sorts the elements with the Comparator

Natural Ordering

- The type of the elements in a List already implements Comparable interface
- Examples
  - If the List consists of String elements, it will be sorted into alphabetical order
  - If it consists of Date elements, it will be sorted into chronological order
  - String and Date both implement the Comparable interface. Comparable implementations provide a natural ordering for a class, which allows objects of that class to be sorted automatically
- If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a ClassCastException.

Sorting by Natural Order of List

```
 // Set up test data
String n[] = {
    new String("John"),
    new String("Karl"),
    new String("Groucho"),
    new String("Oscar")
};

// Create a List from an array
List l = Arrays.asList(n);

// Perform the sorting operation
Collections.sort(l);
```

Sorting by Comparator Interface

- A comparison function, which imposes a total ordering on some collection of objects
- Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order
- Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering

Sorting by Natural Order of List

```
// Set up test data
ArrayList u2 = new ArrayList();
u2.add("Beautiful Day");
u2.add("Stuck In A Moment You Can't Get Out Of");
u2.add("Elevation");
u2.add("Walk On");
u2.add("Kite");
u2.add("In A Little While");
u2.add("Wild Honey");
u2.add("Peace On Earth");
u2.add("When I Look At The World");
u2.add("New York");
u2.add("Grace");

Comparator comp = Comparators.stringComparator();
Collections.sort(u2, comp);
System.out.println(u2);
```

Shuffling

- The shuffle algorithm does the opposite of what sort does, destroying any trace of order that may have been present in a List
  - That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness
- Useful in implementing games of chance
  - could be used to shuffle a List of Card objects representing a deck
  - generating test cases

Routine Data Manipulation

- The Collections class provides five algorithms for doing routine data manipulation on List objects
  - reverse — reverses the order of the elements in a List.
  - fill — overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
  - copy — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
  - swap — swaps the elements at the specified positions in a List.
  - addAll — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

Searching

- The Collections class has binarySearch() method for searching a specified element in a sorted List

```
// Set up testing data
 String name[] = {
     new String("Sang"),
     new String("Shin"),
     new String("Boston"),
     new String("Passion"),
     new String("Shin"),
 };

 List l = Arrays.asList(name);

 int position = Collections.binarySearch(l, "Boston");
 System.out.println("Position of the searched item = " + position);
```

Composition

- Collections.frequency(l) — counts the number of times the specified element occurs in the specified collection
- Collections.disjoint(l1, l2) — determines whether two Collections are disjoint; that is, whether they contain no elements in commonThe Collections class has binarySearch() method for searching a specified element in a sorted List

Collection Framework

Sang Shin
Java Technology Architect
Sun Microsystems, Inc.
sang.shin@sun.com
www.javapassion.com