Java Collections

(adapted from Sun's collections documentation)

APIs and Versions

- Number one hint for programming with Java Collections: use the API
  - http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html
- Be sure to use the 1.5.0 APIs to get the version with generics

Java Collections Framework

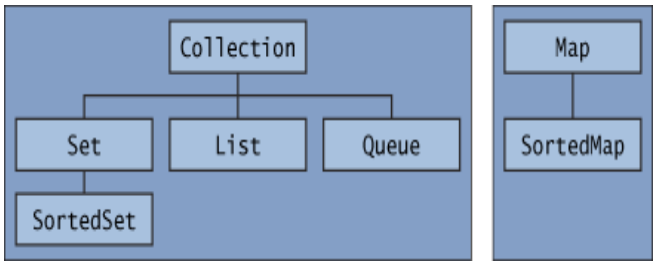
- The Java language API provides many of the data structures from this class for you.
- It defines a “collection” as “an object that represents a group of objects”.
- It defines a collections framework as “a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation.”

Collections Framework (cont)

- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.
- **Legacy Implementations** - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.
- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure** - Interfaces that provide essential support for the collection interfaces.
- **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

- The core collection interfaces encapsulate different types of collections. They represent the abstract data types that are part of the collections framework. They are interfaces so they do not provide an implementation!



public interface Collection<E> extends Iterable<E>

- Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. 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. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

public interface Collection<E> extends Iterable<E>

```
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);
}
```

A note on iterators

- An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator()` method. The following is the `Iterator` interface.

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

public interface **Set<E>** extends Collection<E>

- 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.

public interface **Set<E>** extends Collection<E>

```
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);
}
```

Note: nothing added to Collection interface – except no duplicates allowed

public interface **List<E>** extends Collection<E>

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

public interface **List<E>** extends Collection<E>

```
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);
}
```

A note on ListIterators

- The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.
- The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous
- The set method overwrites the last element returned by next or previous with the specified element.
- The add method inserts a new element into the list immediately before the current cursor position.

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

public interface **Queue<E>** extends Collection<E>

- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

public interface **Queue<E>** extends Collection<E>

```
public interface Queue<E> extends
  Collection<E> {
    E element();           //throws
    E peek();              //null
    boolean offer(E e);    //add - bool
    E remove();            //throws
    E poll();              //null
}
```

public interface **Map<K,V>**

- Map — an object that maps keys to values. 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.

public interface **Map<K,V>**

```
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);
    }
}
```

public interface **SortedSet<E>** extends Set<E>

- 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.

public interface **SortedSet<E>** extends Set<E>

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Note on Comparator interface

- Comparator is another interface (in addition to Comparable) provided by the Java API which can be used to order objects.
- You can use this interface to define an order that is different from the Comparable (natural) order.

public interface **SortedMap<K,V>** extends Map<K,V>

- 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.

public interface **SortedMap<K,V>** extends Map<K,V>

```
public interface SortedMap<K, V> extends Map<K, V>{

    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();

    Comparator<? super K> comparator();
}
```

General-purpose Implementations					
Interfaces	**Implementations**				
	Hash table	**Resizable array**	**Tree (sorted)**	**Linked list**	**Hash table + Linked list**
Set	**HashSet**		TreeSet **(sorted)**		LinkedHashSet
List		**ArrayList**		LinkedList	
Queue					
Map	**HashMap**		TreeMap **(sorted)**		LinkedHashMap

Note the naming convention

LinkedList also implements queue and there is a PriorityQueue implementation (implemented with heap)

implementations

- Each of the implementations offers the strengths and weaknesses of the underlying data structure.
- What does that mean for:
  - Hashtable
  - Resizable array
  - Tree
  - LinkedList
  - Hashtable plus LinkedList
- **Think about these tradeoffs when selecting the implementation!**

Choosing the datatype

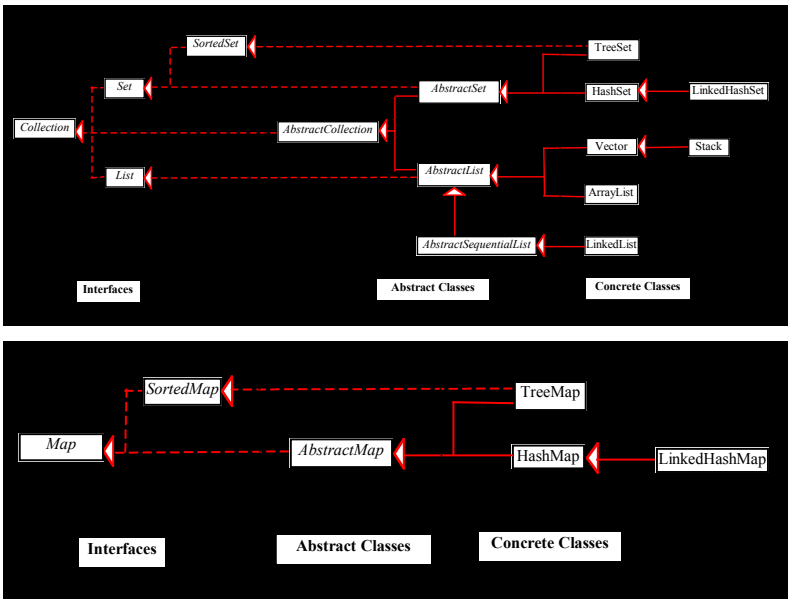
- When you declare a Set, List or Map, you should use Set, List or Map interface as the datatype instead of the implementing class. That will allow you to change the implementation by changing a single line of code!

```
import java.util.*;

public class Test {
   public static void main(String[] args) {
      Set<String> ss = new LinkedHashSet<String>();

      for (int i = 0; i < args.length; i++)
            ss.add(args[i]);

      Iterator i = ss.iterator();
      while (i.hasNext())
            System.out.println(i.next());
   }
}
```

```
import java.util.*;

public class Test {

    public static void main(String[] args)
    {
            //map to hold student grades
            Map<String, Integer> theMap = new HashMap<String, Integer>();

            theMap.put("Plant, Robert", 90);
            theMap.put("Coyne, Wayne", 92);
            theMap.put("Franti, Michael", 98);
            theMap.put("Lennon, John", 88);

            System.out.println(theMap);
            System.out.println("---------------------------------------");
            System.out.println(theMap.get("Korth, Evan"));
            System.out.println(theMap.get("Franti, Michael"));

    }

}
```

Other implementations in the API

- Wrapper implementations delegate all their real work to a specified collection but add (or remove) extra functionality on top of what the collection offers.
  - Synchronization Wrappers
  - Unmodifiable Wrappers
- Convenience implementations are mini-implementations that can be more convenient and more efficient than general-purpose implementations when you don't need their full power
  - List View of an Array
  - Immutable Multiple-Copy List
  - Immutable Singleton Set
  - Empty Set, List, and Map Constants



Making your own implementations

- Most of the time you can use the implementations provided for you in the Java API.
- In case the existing implementations do not satisfy your needs, you can write your own by extending the abstract classes provided in the collections framework.

algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
  - Sorting
  - Shuffling
  - Routine Data Manipulation
    - Reverse
    - Fill copy
    - etc.
  - Searching
    - Binary Search
  - Composition
    - Frequency
    - Disjoint
  - Finding extreme values
    - Min
    - Max