Collections Framework

Agenda

- Collections and Collection Framework
- Benefits of Collection Framework
- Interfaces
  - Collection, Set, List, Map, Iterator, Comparable, Comparator
- Implementations
  - Abstract Implementations
    - AbstractCollection, AbstractSet, AbstractList, AbstractMap
  - General-Purpose Implementations
    - HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap
  - Wrapper Implementations
    - Collections.unmodifiable, Collections.synchronized
- Algorithms
  - Collections and Arrays classes
  - Sort, binarySearch, reverse, shuffle, fill, copy, min, max, etc.
  - Comparator and Comparable interfaces
- Thread-safe collections
  - synchronizedCollection, synchronizedSet, synchronizedList, etc
- Collections and Generics

What is Collection?

- A ***collection*** — 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).
- A collection (or container) in Java is a class that is capable of holding many other objects.

What is Collections Framework?

- A Collections Framework is a unified architecture (or system) for organizing, representing, handling and manipulating collections.
- A Collections Framework is based on four elements:
  1. Interfaces that characterize common collection types
     - Example: List, Set, Map, Iterator, etc
  2. Abstract Classes (AbstractList, AbstractMap, etc) which can be used as a starting point for custom collections and which are extended by the JDK implementation classes
     - Example: AbstractCollection, AbstractList, AbstractMap, AbstractSet, etc
  3. Concrete classes which provide implementations of the Interfaces.
     - Example: ArrayList, LinkedList, HashMap, etc
  4. Algorithms that provide behaviors commonly required when using collections.
     - Example: Sorting, Searching, Shuffling, etc

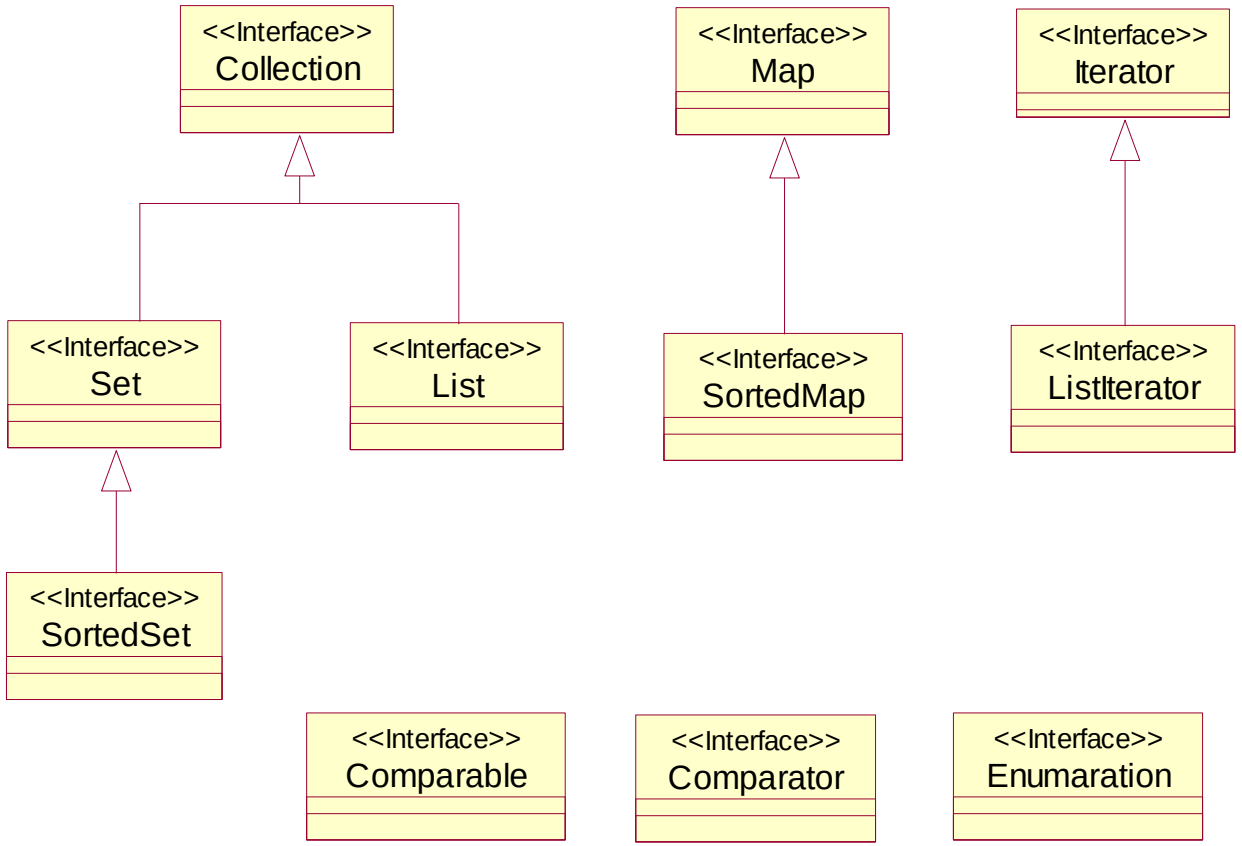
Benefits of Using Collections Framework

- ***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.

Java Collections Facts

- Java built-in collection classes and interfaces
  - Found in the java.util package
- Collection interface
  - Root of all collection interfaces
- Group of objects, which are also called elements
- May allow duplicates and requires no specific ordering
- There are other Collection Frameworks similar to Java Collections
  - C++ has Standard Template Library (STD)
  - Smalltalk has collection hierarchy
  - Both are complex and not easy to learn

Collection Interfaces



Interfaces: Collection

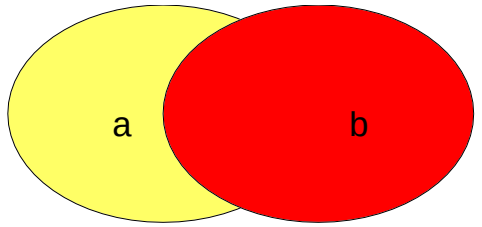
- Root of all collection interfaces
- It represents a group of objects known as its elements.
- Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered.
- Java doesn't provide any direct implementations of this interface but provides implementations of more specific sub interfaces, such as Set and List.

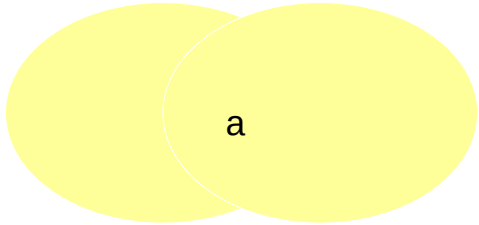
Interfaces: Collection

```
public interface Collection {
    // Query Operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator iterator();
    Object[] toArray();
    Object[] toArray(Object a[]);

    // Modification Operations
    boolean add(Object o);
    boolean remove(Object o);

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();

    // Comparison and hashing
    boolean equals(Object o);
    int hashCode();
}
```

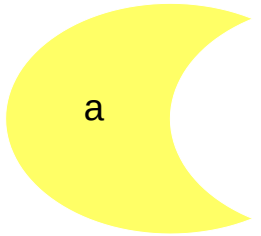
Collection Operations

- Basic Operations
  - int size();
  - boolean isEmpty();
  - boolean contains(Object o);
  - boolean add(Object o);
  - boolean remove(Object o);
- Bulk Operations
  - boolean containsAll(Collection c);
  - boolean addAll(Collection c);
  - boolean removeAll(Collection c);
  - boolean retainAll(Collection c);
  - void clear();
- Array Operations
  - Object[] toArray()
  - Object[] toArray(Object [])

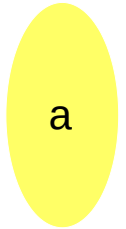




a.addAll(b)



a.removeAll(b)



a.retainAll(b)

Interfaces: Set

- It is an ***unordered*** collection that cannot contain ***duplicate*** elements.
- It models the mathematical set abstraction.
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- 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.
- Some built-in implementing classes:
  - ***HashSet*** : Fast, unordered
  - ***TreeSet***: Slower, but ordered. It also supports methods in the SortedSet interface
  - ***LinkedHashSet:*** Hash table and linked list implementation of the Set interface. An insertion-ordered Set implementation that runs nearly as fast as HashSet.

Interfaces: Set

```
public interface Set extends Collection {
    // Query Operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator iterator();
    Object[] toArray();
    Object[] toArray(Object a[]);

    // Modification Operations
    boolean add(Object o);
    boolean remove(Object o);

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean retainAll(Collection c);
    boolean removeAll(Collection c);
    void clear();

    // Comparison and hashing
    boolean equals(Object o);
    int hashCode();
}
```

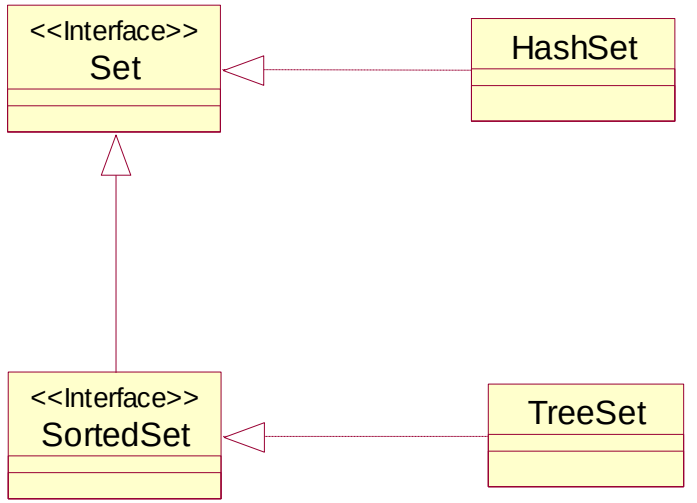
Interfaces: SortedSet

- It is an ***ordered*** collection that cannot contain ***duplicate*** elements.
- The element added to a SortedSet must either implement *Comparable* or you must provide a Comparator to the constructor to its implementation class: TreeSet.

```
public interface SortedSet extends Set {
    Comparator comparator();
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);
    Object first();
    Object last();
}
```

Set Implementations

- There are two implementations of the Set interface
  - HashSet – Fast, unordered,
  - TreeSet – Slower, but ordered. It also supports methods in the SortedSet interface
- More often you will use a HashSet for storing your duplicate-free collection.
- The TreeSet implementations useful when you need to extract elements from a collection in a sorted manner. It is generally faster to add elements to the HashSet then convert the collection to a TreeeSet for sorted traversal.
- Example: SetExample class
- UML Diagrams



Interfaces: List

- It is an ***ordered*** collection that might contain ***duplicate*** elements.
- Some built-in implementing classes: ***ArrayList***, ***LinkedList*** and ***Vector***
- In addition to the operations inherited from Collection, the List interface includes operations for the following:
  - 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.

Interfaces: List

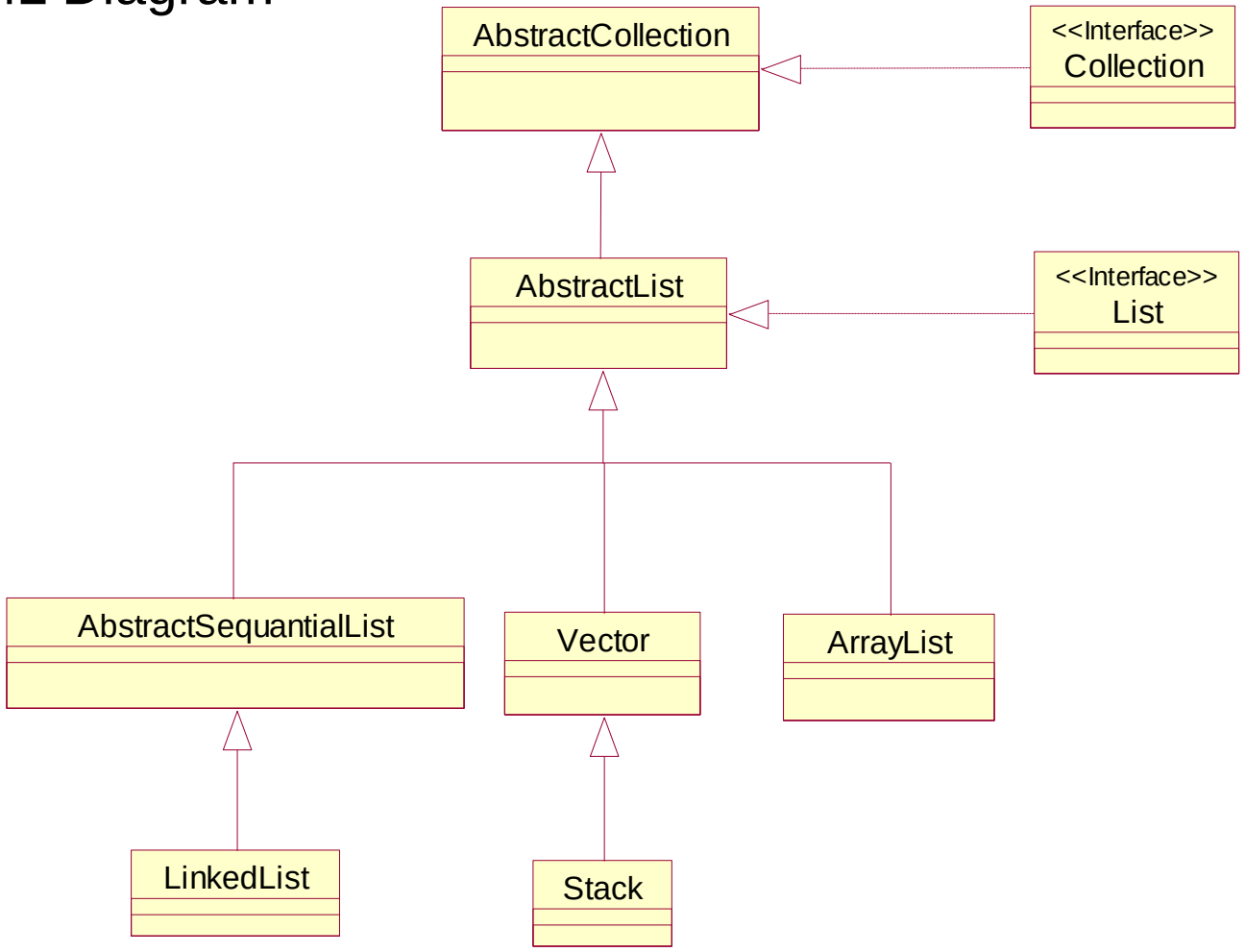
```
public interface List extends Collection {
    // Query Operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator iterator();
    Object[] toArray();
    Object[] toArray(Object a[]);

    // Modification Operations
    boolean add(Object o);
    boolean remove(Object o);

    // Bulk Modification Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean addAll(int index, Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();

    // Comparison and hashing
    boolean equals(Object o);
    int hashCode();

    // Positional Access Operations
    Object get(int index);

    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);

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

    // List Iterators
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // View
    List subList(int fromIndex, int toIndex);
}
```

List Implementations

- ***ArrayList*** and ***LinkedList*** are the two implementations of the ***List*** interface.
- If you need to support random access, without inserting or removing elements from any place to other than the end, then ArrayList offers you the optimal collection
- ArrayList is resizable-array implementation of the List interface. (Essentially an unsynchronized Vector.)
- ArrayList is very fast and uses the least amount of memory. But it is expensive to add and remove elements to/from the beginning and middle.
- The LinkedList class provides uniformly named methods to get, remove and insert an element at the beginning and end of the list.

List Implementations

- UML Diagram



Interfaces: Iterator

- The standard Iterator interface allows us to traverse the elements in a collection (with hasNext() and next()) and optionally removes an element from the underlying collection.

```
public interface Iterator {
        boolean  hasNext();
        /*Returns true if the iteration has more elements.*/
        Object next();
        /*Returns the next element in the iteration
        and moves this iterator forward onto the next element.*/
        void remove();
        /*Removes from the underlying collection the last element
        returned by the iterator (optional operation).*/
 }
```

```
…
Iterator iter = c.iterator();

while (iter.hasNext()) {
       Object currentElement = iter.next();
/*handle the current element*/
        // …
        Iter.remove();
/* removes the last element returned by
next()*/
        // …
}
```

- The Enumeration interface is historical version of Iterator which allows you to iterate through all the elements of a collection. However, not all libraries support the newer interface, so you may find yourself using Enumeration from time to time.

Interfaces: ListIterator

- The ListIterator interface extends the Iterator interface to support bi-directional access (next, previous) as well as adding or removing or changing elements in the underlying collection (add, remove, set)
- Since lists are ordered, we can determine an index of where we are within a list.

```
public interface ListIterator extends Iterator{
    boolean  hasPrevious();
    Object previous();
    int nextIndex();
    int previousIndex();
    public void set (Object o);
    public void add (Object o);
}
```

Interfaces: Map

- The Map interface is not an extension of the Collection interface.
- Instead, the interface starts off its own interface hierarchy for maintaining key-value associations.
- The interface describes a mapping from keys to values, ***without duplicate keys***, by definition.
- Some built-in implementing classes: ***AbstractMap***, ***HashMap*** and ***TreeMap***
- The Map interface provides three collection views, which allow a map's contents to be viewed as *a set of keys, collection of values, or set of key-value mappings.*

Interfaces: Map

- The interface methods can be broken down into three sets of operations: altering, querying and providing alternative views

  /*The alteration operation allows you to add and remove key-value pairs from the map. Both the key and value can be null. However you should not add a Map to itself as a key or value.*/

  Object **put**(Object key, Object value)
  Object **remove**(Object key)
  void **putAll(Map t)**
  void **clear**()

  //The query operations allow you to check on the contents of the map
  Object **get**(Object key)
  boolean **containsKey**(Object key)
  boolean **containsValue**(Object value)
  int **size**()
  boolean **isEmpty**()

  /*The set methods allow you to work with the group of keys or values as a collection*/
  Set **keySet**()
  Collection **values**()
  Set **entrySet**()

Interfaces: SortedMap

- The SortedMap interface keeps all elements in a sorted order.
- Since the elements are ordered, the interface supports methods to extract ranges of elements from the map.
- The Comparator object passed in defines the sort order

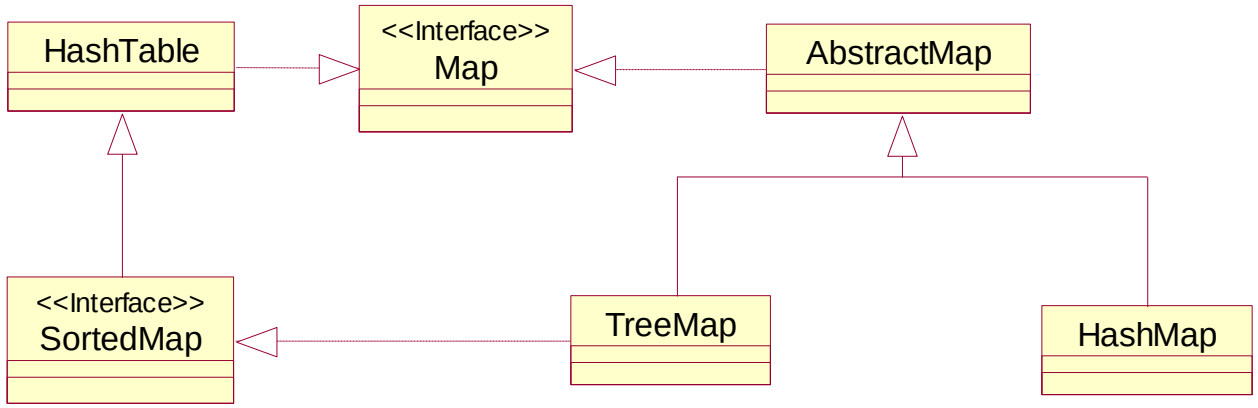
```
public interface SortedMap extends Map {
    Comparator comparator();
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap tailMap(Object fromKey);
    Object firstKey();
    Object lastKey();
}
```

Map Implementations

- There are two general purpose concrete implementations of the Map interface.
  - HashMap
  - TreeMap
- The decision as to which one to use is same as the decision made for deciding between HashSet and TreeSet
  - *HashMap* is much faster than TreeMap
  - *TreeMap* maintains the ability to do ordered iteration
- Hashtable is historical version of HashMap

Map Implementations

- UML Diagram



Collections Algorithms

- The ***Collections*** class, available as part of the Collections Framework, provide support for various algorithms with the collection classes, both new and old.

sort(List)	Sorts a list using a merge sort algorithm, which provides average-case performance comparable to a high-quality quicksort, guaranteed O(n*log n) performance (unlike quicksort), and stability (unlike quicksort). (A stable sort is one that does not reorder equal elements.)
binarySearch(List, Object)	Searches for an element in an ordered list using the binary search algorithm.
reverse(List)	Reverses the order of the elements in the a list.
shuffle(List)	Randomly permutes the elements in a list.
fill(List, Object)	Overwrites every element in a list with the specified value.
copy(List dest, List src)	Copies the source list into the destination list.
min(Collection)	Returns the minimum element in a collection.
max(Collection)	Returns the maximum element in a collection.
rotate(List list, int distance)	Rotates all of the elements in the list by the the specified distance.
replaceAll(List list, Object oldVal, Object newVal)	Replaces all occurrences of one specified value with another.
indexOfSubList(List source, List target)	Returns the index of the first sublist of source that is equal to target.
lastIndexOfSubList(List source, List target)	Returns the index of the last sublist of source that is equal to target.
swap(List list, int, int)	Swaps the elements at the specified positions in the specified list.

Comparator Interface

- Many methods in the ***Collections*** require Comparator object
  - Void sort (List list, Comparator c);
- Custom ***Comparator***s are used to provide the Collection algorithm the ability to compare two objects for comparison.
- The ***Comparator*** interface has two simple methods.

```
public interface Comparator {
        public int compare(Object o1, Object o2);
        boolean equals (Object o);
}
```

- ***compare*** method should return
  - Negative int number (<=-1) - first argument is less than second one
  - Zero (==0) - first argument is equal to the second one
  - Positive integer number (>=1) - first argument is greater than the second one
- Example:
  - Create a PersonNameComparator class implementing Comparator interface
  - Implement compare() method
  - List personList = new ArrayList();
  - Comparator comp = new PersonNameComparator();
  - Collections.sort(personList, comp);

Comparable Interface

- Many sort, order and search methods in the ***Collections*** class have two versions: One that takes a Comparator and that doesn't.
- If there is no ***Comparator***, Collections doesn't know how to sort them but objects themselves know how to sort
- To use Collections methods that have no ***Comparator***, we must pass in collections of objects where each object in that collection implements the ***Comparable*** interface.

```
public interface Comparable {
        public int compareTo(Object o);
}
```

- If we want to use Collections.sort (person), then Person class has to implement Comparable

```
public class Person implements Comparable {
      ……
      public int compareTo(Object o) {
         Person p = (Person) o;
         return name.compareTo(p.name);
      }
}
```

Arrays Algorithms

- Java 2 includes a handy class for manipulating standard Java arrays: Arrays class
- The ***Arrays*** class, like collection class contains only static methods.

Method Name	Task
sort	Sorts the supplied array
binarySearch	Does a binary search for a particular item
equals	Compares two arrays for equality
fill	Fills all elements in the array with the specified object
asList	Converts the array of objects into a List

- Examples
  - Set aSet = new HashSet(Arrays.asList(anArray));
  - String[] uniqueArray = (String[]) aSet.toArray(new String[0]);

Thread-safe collections

- The key difference between the historical collection classes and the new
- implementations within the Collections Framework is that the new classes are not thread-safe.
- You use synchronization only when you need it, making everything work much faster.
- If, however, you are using a collection in a multi-threaded environment, where multiple threads can modify the collection simultaneously, the modifications need to be synchronized.
- The Collections class provides for the ability to ***wrap*** existing collections into synchronized ones with another set of six methods:
  - Collection synchronizedCollection(Collection collection)
  - List synchronizedList(List list)
  - Map synchronizedMap(Map map)
  - Set synchronizedSet(Set set)
    - Set set = Collections.synchronizedSet(new HashSet());
  - SortedMap synchronizedSortedMap(SortedMap map)
  - SortedSet synchronizedSortedSet(SortedSet set)
- Making a collection unmodifiable also makes a collection thread-safe, as the collection can't be modified.
  - Collection unmodifiableSetCollection(Collection collection)

Collections and Generics

- Generics were introduced with Java 1.5
- Generics allow you to abstract over types. The most common examples are containers (collections)
- When you take an element out of a Collection, you must cast it to the type of element that is stored in the collection. Besides being inconvenient, this is unsafe. The compiler does not check that your cast is the same as the collection's type, so the cast can fail at run time.
- Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked. Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and can insert the correct casts on values being taken out of the collection.
- Example: Collection<String> c
- When you see the code <Type>, read it as “of Type”; the declaration above reads as “Collection of String c.” The code using generics is clearer and safer. We have eliminated an unsafe cast and a number of extra parentheses. More importantly, we have moved part of the specification of the method from a comment to its signature, so the compiler can verify at compile time that the type constraints are not violated at run time.

Collections and Generics

- Here is a simple example using traditional Collections:

```
// Removes 4-letter words from c. Elements must be strings
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      if (((String) i.next()).length() == 4)
        i.remove();
}
```

- Here is the same example modified to use generics:

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
      if (i.next().length() == 4)
        i.remove();
}
```

Summary

Interface	Abstract Implementation	Implementation Classes	Synchronized?	Characteristics	Historical	Synchronized?
Collection	AbstractCollection	N/A				
Set	AbstractSet	HashSet	No	Hash table, fast, no duplication, no order		
		TreeSet	No	Balanced binary tree, slower, no duplication, ordered		
		LinkedHashSet	No	Slower, no duplication, ordered		
List	AbstractList	ArrayList	No	Resizable array, fast, ordered, allows duplication	Vector	Yes
	AbstractSequantialList	LinkedList	No	Linked list, slower, ordered, allows duplication	Stack	
Map	AbstractMap	HashMap	No	Hash table, fast, maps keys to values, no duplication, no iteration order	Hashtable	Yes
		TreeMap	No	Balanced binary tree. slower, ordered iteration,	Properties	
		LinkedHashMap	No	Insertion order iteration,		

Additional Reading and Other Resources

- ***Collections Framework by Oracle***
  - ***https://docs.oracle.com/javase/tutorial/collections/***
- ***Annotated Outline of Collections Framework by Sun***
  - ***http://docs.oracle.com/javase/6/docs/technotes/guides/collections/reference.html***
- ***Java Collections Framework by IBM developerWorks***
  - ***Under eCentennial | Content / Additional Resources | Resources for Java Intermediate | Java Collections Framework***
- ***Just Java 2 by Peter van der Linden***
  - ***Chapter 15. Regular Expressions, Collections, Utilities***
- ***Generics by Oracle***
  - ***https://docs.oracle.com/javase/tutorial/java/generics/***