Collections

Interfaces, Implementations, Algorithms, Interoperability

see

java.sun.com/docs/books/tutorial/reallybigindex.html

java.sun.com/docs/books/tutorial/collections/index.html

What Is a Collection?

A collection (or container) is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.

Examples:

- a poker hand (a collection of cards)
- a mail folder (a collection of letters)
- a telephone directory (a collection of name-to-phone-number mappings)

What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain three things:

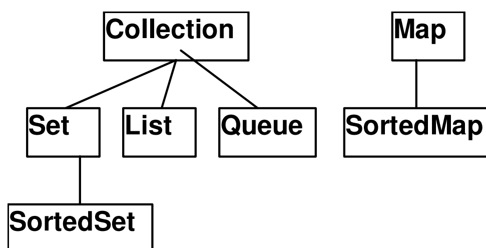
- Interfaces: abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.
- Implementations: concrete implementations of the collection interfaces.
- Algorithms: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

What are the Benefits of a Collections Framework?

- It reduces programming effort
- It increases program speed and quality
- It allows interoperability among unrelated APIs
- It reduces the effort to learn and use new APIs
- It reduces effort to design new APIs
- It fosters software reuse

Interfaces

The core collection interfaces are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation. The core collections interfaces are shown below:



Collection Classes

Collection

A Collection represents a group of objects, known as its elements. Some Collection implementations allow duplicate elements and others do not. Some are ordered and others unordered.

Collection is used to pass collections around and manipulate them when maximum generality is desired.

Set

A Set is a collection that cannot contain duplicate elements

It is used to represent sets like the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.

List

A List is 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

The user can access elements by their integer index (position).

Map

A Map is an object that maps keys to values

Maps cannot contain duplicate keys

Each key can map to at most one value

SortedSet and SortedMap

These are merely sorted versions of Set and Map

Object Ordering

There are two ways to order objects:

- The Comparable interface provides automatic natural order on classes that implement it
- The Comparator interface gives the programmer complete control over object ordering

These are not core collection interfaces, but underlying infrastructure.

The Collection Interface

A Collection represents a group of objects, known as its elements

The primary use of the Collection interface is to pass around collections of objects where maximum generality is desired

Suppose you have a Collection, c, which may be a List, a Set, or some other kind of Collection. The following one-liner creates a new ArrayList (an implementation of the List interface), initially containing all of the elements in c:

```
List<String> l = new ArrayList<String>(c);
```

The Collection interface

```
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 iterator();

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); Opt
    boolean removeAll(Collection<?> c); // Opt
    boolean retainAll(Collection<?> c); // Opt
    void clear();                     // Optional

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

Enumeration

An instance of the class `Enumeration` generates a series of elements, one at a time. Successive calls to the `nextElement()` method return successive elements of the series until `hasMoreElements()` returns false.

The class `StringTokenizer` implements the Enumeration interface

Iterators

The object returned by the iterator method implements the `Iterator` interface

An `Iterator`, is very similar to an `Enumeration`, but allows the caller to remove elements from the underlying collection during the iteration

Note: There is no safe way to remove elements from a collection while traversing it with an `Enumeration`

The Iterator interface is shown below:

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

The remove() method removes from the underlying Collection the last element that was returned by next(). The remove method may be called only once per call to next, and throws an exception if this condition is violated

Here is how to use an Iterator to filter a Collection by removing every element that does not satisfy some condition:

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

The enhanced for statement

The enhanced for statement has the form:

```
double d[] = new double[10];

for ( double d: a ) {
  // something using d
}
```

The Expression must either have type Iterable or else it must be of an array type

The scope of a local variable declared in the FormalParameter part of an enhanced for statement is the contained Statement

For an array

```
double d;
for( int #i = 0;
     #i < a.length;
     #i++ ) {
  d = a[#i];
  // something using d
}
```

If the type of the expression is a subtype of Iterable

```
for (I #i = Expression.iterator();
     #i.hasNext(); ) {

  E Identifier = #i.next();
   Statements
}
```

Bulk Operations

The bulk operations perform some operation on an entire Collection

`containsAll(c)`: Returns true if the target Collection contains all of the elements in the specified Collection

`addAll(c)`: Adds all of the elements in the specified Collection to the target Collection

`removeAll(c)`: Removes from the target Collection all of its elements that are also contained in the specified Collection

`retainAll(c)`: Removes from the target Collection all of its elements that are not also contained in the specified Collection. That is to say, it retains only those elements in the target Collection that are also contained in the specified Collection

`clear()`: Removes all elements from the Collection

The addAll, removeAll, and retainAll methods all return true if the target Collection was modified in the process of executing the operation.

As a simple example of the power of the bulk operations, consider following idiom to remove all instances of a specified element, e from a Collection, c.:

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose that you want to remove all of the null elements from a Collection:

```
c.removeAll(Collections.singleton((X)null));
```

Array Operations

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. They allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose c is a Collection. The following code dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c:

```
Object[] a = c.toArray();
```

Suppose 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(new String[0]);
```

The Set Interface

A Set is a Collection without duplicate elements

The Set interface extends Collection and contains no methods other than those inherited from Collection

Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set objects with different implementation types to be compared meaningfully

Two Set objects are equal if they contain the same elements.

The Set interface is shown below:

```
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 iterator();

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);Opt
    boolean removeAll(Collection<?> c); // Opt
    boolean retainAll(Collection<?> c); // Opt
    void clear();                    // Optional

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

Removing duplicates from a Collection:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

Set Example

Here's a little program that takes the words in its argument list and prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated:

```
import java.util.*;

public class FindDups {
  public static void main(String args[]) {
    Set<String> s = new HashSet<String>();
    for (String a: args)
      if (!s.add(a))
        System.out.println( "Duplicate detected: "
                           + a );

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

Now let's run the program:

```
% java FindDups i came i saw i left

Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

Note that the collection is always referred to by its interface type (`Set`), rather than by its implementation type (`HashSet`).

Set Example (SortedSet)

The implementation type of the `Set` in the example above is `HashSet`, which makes no guarantees as to the order of the elements in the `Set`. If you want the program to print the word list in alphabetical order, all you have to do is to change the set's implementation type from `HashSet` to `TreeSet`. Making this trivial one-line change causes the command line in the previous example to generate the following output:

```
% java FindDups i came i saw i left

Duplicate word detected: i
Duplicate word detected: i
4 distinct words detected: [came, i, left, saw]
```

Set Bulk Operations

Bulk operations perform standard set-algebraic operations. Suppose s1 and s2 are Sets. Here's what the bulk operations do:

- `s1.containsAll(s2)`: **Returns true if s2 is a subset of s1**
- `s1.addAll(s2)`: **Transforms s1 into the union of s1 and s2**
- `s1.retainAll(s2)`: **Transforms s1 into the intersection of s1 and s2**
- `s1.removeAll(s2)`: **Transforms s1 into the (asymmetric) set difference of s1 and s2**

To calculate the union, intersection, or set difference of two sets non-destructively the caller must copy one set before calling the appropriate bulk operation. For example:

```
Set<Type> union = new HashSet<Type> (s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type> (s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type> (s1);
difference.removeAll(s2);
```

The example used `HashSet` but any general-purpose Set implementation could be substituted.

Duplicates example again

Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want duplicates printed out repeatedly. This can be done by generating two sets, one containing every word in the argument list, and the other containing only the duplicates. Here's how the resulting program looks:

```
import java.util.*;

public class FindDups2 {
  public static void main(String args[]) {
    Set<String> uniques = new HashSet<String> ();
    Set<String> dups = new HashSet<String> ();

    for ( String a: args )
      if ( !uniques.add( a ) )
        dups.add( a );

    // destructive set-difference
    uniques.removeAll( dups );

    System.out.println( "Unique words:    "
                        + uniques);
    System.out.println("Duplicate words: "
                        + dups);
  }
}
```

Now let's run the revised program with the same same argument list we used before:

```
% java FindDups2 i came i saw i left

Unique words:    [came, left, saw]
Duplicate words: [i]
```

The List Interface

A List is an ordered Collection(sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for:

- Positional Access: access by numerical position in the list.
- Search: search for a specified object in the list and return its numerical position.
- List Iteration: extend Iterator semantics to take advantage of the list's sequential nature.
- Range-view: perform arbitrary range operations on the list.

The List interface is shown below:

```
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
    abstract 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);
}
```

There are two general-purpose List implementations

- ArrayList - generally the best-performing implementation
- LinkedList - sometimes offering better performance

List Collection Operations

The remove operation always removes the first occurrence of the specified element from the list

The add and addAll operations always append the new element(s) to the end of the list

The following idiom concatenates one list to another:

```
list1.addAll(list2);
```

Here's a non-destructive form of this idiom, which produces a third List consisting of the second list appended to the first:

```
List<X> list3 = new ArrayList<X>(list1);
list3.addAll(list2);
```

List strengthens the requirements on the equals and hashCode methods so that two List objects can be compared for logical equality without regard to their implementation classes

Two List objects are equal if they contain the same elements in the same order.

Positional Access and Search Operations

The set and remove operations return the old value that is being overwritten or removed

The `addAll()` operation inserts all of the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection's iterator

The following short program uses the `shuffle()` method to print the words in its argument list in random order:

```
import java.util.*;

public class Shuffle {
    public static void main(String args[]) {
        List<String> l = new ArrayList<String> ();
        for (String a: args)
            l.add(a);
        Collections.shuffle(l, new Random());
        System.out.println(l);
    }
}
```

The `Arrays` class has a static factory method called `asList()` that allows an array to be viewed as a `List`. This method does not copy the array. With other small changes you get the following smaller and faster program, whose behavior is identical to the previous program:

```
import java.util.*;
public class Shuffle {
    public static void main(String args[]) {
        List<String> l = Arrays.asList(args);
        Collections.shuffle(l);
        System.out.println(l);
    }
}
```

List Iterators

The Iterator returned by `List`'s `iterator()` operation returns the elements of the list in proper sequence. Additionally, `List` provides a richer iterator, called a `ListIterator`, that allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator. The ListIterator interface is summarized below (including the three methods it inherits from Iterator):

```
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 o);     // Optional
    void add(E o);     // Optional
}
```

Here's the standard idiom for iterating backwards through a list:

```
for (ListIterator i=l.listIterator(l.size());
     i.hasPrevious(); ) {
  Foo f = (Foo) i.previous();
    ...
}
```

The `listIterator()` form with no arguments returns a ListIterator positioned at the beginning of the list, and the form with an int argument returns a ListIterator positioned at the specified index

Range-view Operation

The range-view operation, `subList(int fromIndex, int toIndex)`, returns a List view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive. This half-open range mirrors the typical for-loop:

```
for (int i=fromIndex; i<toIndex; i++) {
    ...
}
```

The returned List is backed by the List on which subList was called, so changes in the former List are reflected in the latter.

Any operation that expects a List can be used as a range operation by passing a subList view instead of a whole List. For example, the following idiom removes a range of elements from a list:

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms may be constructed to search for an element in a range:

```
int i = list.subList(fromIndex,
toIndex).indexOf(o);
int j = list.subList(fromIndex,
                     toIndex).lastIndexOf(o);
```

Note that the above idioms return the index of the found element in the subList, not the index in the backing List.

Range-view Example

Here's a polymorphic algorithm whose implementation uses subList to deal a hand from a deck

It returns a new List (the "hand") containing the specified number of elements taken from the end of the specified List (the "deck"). The elements returned in the hand are removed from the deck.

```
public static <E> List<E> dealHand(
                             List<E> deck, int n ) {
    int deckSize = deck.size();
    List<E> handView = deck.subList( deckSize - n,
                                       deckSize);
    List<E> hand = new ArrayList<E> ( handView );
    handView.clear();
    return hand;
}
```

For many common List implementations, like ArrayList, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

Dealing Poker Hands Example

Here's a program using the dealHand method in combination with Collections.shuffle to generate hands from a normal 52-card deck. The program takes two command line arguments: the number of hands to deal and the number of cards in each hand

```
import java.util.*;

class Deal {
  public static void main(String args[]) {
    int numHands = Integer.parseInt(args[0]);
    int cardsPerHand = Integer.parseInt(args[1]);

    // Make a normal 52-card deck
    String[] suit = new String[] {
            "spades", "hearts", "diamonds",
"clubs"};
    String[] rank = new String[]
            {"ace","2","3","4","5","6","7","8",
             "9","10","jack","queen","king"};
    List<String> deck = new ArrayList<String> ();
    for (int i=0; i<suit.length; i++)
      for (int j=0; j<rank.length; j++)
        deck.add(rank[j] + " of " + suit[i]);
    Collections.shuffle(deck);
    for (int i=0; i<numHands; i++)
      System.out.println(dealHand(deck,
                                  cardsPerHand));
    }
}
```

Program output

```
% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of
spades, king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of
hearts, queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of
diamonds, 9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of
hearts, ace of hearts]
```

Warning on use of subList

The semantics of the List returned by subList become undefined if elements are added to or removed from the backing List in any way other than via the returned List

Thus, it's highly recommended that you use the List returned by subList only as a transient object, to perform one or a sequence of range operations on the backing List

The longer you use the subList object, the greater the probability that you'll compromise it by modifying the backing List directly (or through another subList object).

Queue

```
public interface Queue<E> extends Collection<E> {
    E element(); // peek at first
    boolean offer(E o); // add (no exception)
    E peek(); // peek at first
    E poll(); // remove and return first
    E remove(); // remove and return first
}

/* from Collection
    boolean add(E o);
*/
```

Algorithms

Most of the polymorphic algorithms in the `Collections` class apply specifically to `List`. Having all of these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms:

`sort(List<T>)`: 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(List<T>)`: Randomly permutes the elements in a List

`reverse(List<T>)`: Reverses the order of the elements in a List.

`fill(List<T>, T)`: Overwrites every element in a List with the specified value.

`copy(List<T> dest, List<T> src)`: Copies the source List into the destination List.

`binarySearch(List<T>, T)`: Searches for an element in an ordered List using the binary search algorithm.

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. The Map interface is shown below:

```
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> t);
    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);
    }
}
```

There are two `Map` implementations. `HashMap`, which stores its entries in a hash table, is the best-performing implementation. `TreeMap`, which stores its entries in a red-black tree, guarantees the order of iteration.

Printing a frequency table

Here's a simple program to generate a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
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( args[i] );
      m.put( a, ( freq == null
                ? 1
                : freq + 1) );
    }

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

The second argument of the put statement is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before, or one more than its current value if the word has already been seen. Let's run the program:

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

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

Printing a frequency table - sorted

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the `Map` from `HashMap` to `TreeMap`. Making this four character change causes the program to generate the following output from the same command line:

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

Map Bulk Operations

The `putAll()` operation is the Map analogue of the Collection interface's addAll() operation

In addition to its obvious use of dumping one Map into another, it has a second, more subtle, use

Suppose a Map is used to represent a collection of attribute-value pairs; the putAll operation, in combination with the standard Map constructor, provides a way to implement attribute map creation with default values. Here's a static factory method demonstrating this technique:

```
static <K,V> Map<K,V> newAttributeMap(Map<K,V>
defaults, Map<K,V> overrides) {
    Map<K,V> result =  new HashMap<K,V>
(defaults);
    result.putAll(overrides);
    return result;
}
```

Map Collection Views

The Collection-view methods allow a Map to be viewed as a Collection in three ways:

`keySet()`: the Set of keys contained in the Map.

`values()`: The Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.

`entrySet()`: The Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called `Map.Entry` that is the type of the elements in this Set.

The Collection-views provide the only means to iterate over a Map. Here's an example illustrating the standard idiom for iterating over the keys in a Map:

```
for ( KeyType key: m.keySet() )
  System.out.println( key );
```

The idiom for iterating over values is analogous. Here's the idiom for iterating over key-value pairs:

```
for(Map.Entry<KeyType, ValType> e: m.entrySet()) {
    System.out.println(e.getKey() + ": "
                       + e.getValue());
}
```

With all three Collection-views, calling an Iterator's remove operation removes the associated entry from the backing Map. With the entrySet view, it is also possible to change the value associated with a key, by calling a Map.Entry's setValue method during iteration.

The Collection-views do not support element addition under any circumstances.

Fancy Uses of Collection-Views: Map Algebra

Find out if one Map is a submap of another, that is, whether the first Map contains all of the key-value mappings in the second. The following idiom does the trick:

```
if (m1.entrySet().containsAll(m2.entrySet())) {
    ...
}
```

Find out if two Map objects contain mappings for all the same keys:

```
if (m1.keySet().equals(m2.keySet())) {
    ...
}
```

Find all the keys common to two Map objects:

```
Set<KeyType> commonKeys
        = new HashSet<KeyType> (a.keySet());
commonKeys.retainAll(b.keySet());
```

A similar idiom gets you the common values, and the common key-value pairs.

Remove all the key-value pairs that one Map has in common with another:

```
m1.entrySet().removeAll(m2.entrySet());
```

Remove from one Map all the keys that have mappings in another:

```
m1.keySet().removeAll(m2.keySet());
```

Object Ordering

A List l may be sorted as follows:

```
Collections.sort(l);
```

If the list consists of `String` elements, it will be sorted into lexicographic (alphabetical) order. If it consists of `Date` elements, it will be sorted into chronological order. `String` and `Date` both implement the `Comparable` interface. The `Comparable` interfaces provides a natural ordering for a class, which allows objects of that class to be sorted automatically. The following table summarizes the JDK classes that implement `Comparable`:

Class	Natural Ordering
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical
Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	lexicographic on system pathname.
String	lexicographic
Date	chronological
CollationKey	locale-specific lexicographic

If you try to sort a list whose elements do not implement `Comparable` or whose elements cannot be compared to one another, `Collections.sort(list)` will throw a `ClassCastException`. Elements that can be compared to one another are called mutually comparable. While it is possible to have elements of different types be mutually comparable, none of the JDK types listed above permit inter-class comparison.

Writing Your Own Comparable Types

The Comparable interface consists of a single 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, zero, or a positive integer as 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.

Implementing the `Comparable` interface

```
import java.util.*;

public class Name implements Comparable {
  private String  firstName, lastName;

  public Name(String firstName, String lastName) {
    if (firstName==null || lastName==null)
            throw new NullPointerException();
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String firstName()    {return firstName;}
  public String lastName()     {return lastName;}

  public boolean equals(Object o) {
    if (!(o instanceof Name)) return false;
    Name n = (Name)o;
    return n.firstName.equals(firstName) &&
              n.lastName.equals(lastName);
  }

  public int hashCode() {
    return 31*firstName.hashCode()
          + lastName.hashCode();
  }

  public String toString() {return firstName + " "
                                + lastName;}
  public int compareTo(Name n) {
    int lastCmp = lastName.compareTo(n.lastName);
    return (lastCmp!=0 ? lastCmp :
               firstName.compareTo(n.firstName));
  }
}
```

Implementing the `Comparable` interface for Names

Name objects are immutable - mutable objects that will be used as elements in Sets, or keys in Maps will break the collection.

The constructor checks its arguments for null. This ensures that all Name objects are well-formed, so that none of the other methods will ever throw a `NullPointerException`.

The `hashCode` method is redefined. This is essential for any class that redefines the equals method. It is required by the general contract for `Object.equals`. (Equal objects must have equal hash codes.)

The `equals` method returns false if the specified object is null, or of an inappropriate type. The `compareTo` method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.

The `toString` method has been redefined to print the Name in human-readable form. This is always a good idea, especially for objects that are going to get put into collections.

Implementing the `Comparable` interface in general

`compareTo` is implemented as follows

First you compare the most significant part of the object (for Names, the last name). Often, you can just use the natural ordering of the part's type. If the comparison results in anything other than zero (which represents equality), you're done: you just return the result. If the most significant parts are equal, you go on to compare the next-most-significant parts. (For names there are only two parts - first name and last name). If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal (or you were comparing the least-significant parts), at which point you'd return the result of the comparison.

There are restrictions on the behavior of the `compareTo` method

- it should be transitive
  - a<b and b<c -> a<c
  - a=b and b=c -> a=c
- consistent with equals
- comparing in the reverse order must change the sign of the comparison
- equal objects should have equal hashCodes
- inequivalent objects should never be equal

Comparators

To order or sort objects in a different order from their natural order you can use a Comparator

A Comparator is an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

The compare method compares its two arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

Example

Suppose you have a class called EmployeeRecord:

```
public class EmployeeRecord
                 implements Comparable {
    public Name name();
    public int employeeNumber();
    public Date hireDate();
             ...
}
```

Let's assume that the natural ordering of EmployeeRecord objects is Name-ordering on employee name. How do we get a list of employees in order of seniority? Here's a program that will produce the required list:

```
import java.util.*;

class EmpSort {
  static final Comparator<Employee> SENIOR_ORDER
        = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
      return
         e2.hireDate().compareTo(e1.hireDate());
    }
  };

  // Employee Database
  static final Collection<Employee> employees =
... ;

  public static void main(String args[]) {
    List<Employee> emp
        = new ArrayList<Employee>(employees);
    Collections.sort(emp, SENIOR_ORDER);
    System.out.println(emp);
  }
}
```

Note that the Comparator passes the hire-date of its second argument to its first to reverse the order.

Partial order and Total order

The Comparator in the above program cannot be used to order a sorted collection (such as TreeSet) because it generates a strictly partial ordering. If you insert multiple employees who were hired on the same date into a TreeSet with this Comparator, only the first one will be added to the set.

To fix this problem, you have to modify the Comparator so that it produces a total ordering. The way to do this is to do a two-part comparison where the first part is the one that we're actually interested in and the second part is attribute that uniquely identifies the object. Here's the Comparator that results:

```
static final Comparator SENIOR_ORDER
            = new Comparator<Employee>() {
  public int compare(Employee e1, Employee e2) {
    int dateCmp
          =
e2.hireDate().compareTo(e1.hireDate());
    if (dateCmp != 0)
      return dateCmp;
    return (e1.employeeNumber() <
e2.employeeNumber()
           ? -1
           : (e1.employeeNumber()
              == e2.employeeNumber() ? 0 : 1));
  }
};
```

Don't replace the final return statement in the Comparator with the simpler:

```
return r1.employeeNumber() - r2.employeeNumber();
```

It will fail if the subtraction overflows

The SortedSet Interface

A SortedSet is a Set that maintains its elements in ascending order, sorted according to the elements' natural order, or according to a Comparator provided at SortedSet creation time. In addition to the normal Set operations, the Set interface provides operations for:

Range-view: Performs arbitrary range operations on the sorted set.

Endpoints: Returns the first or last element in the sorted set.

Comparator access: Returns the Comparator used to sort the set (if any).

The SortedSet interface is shown below:

```
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();
}
```

Set Operations

The operations that SortedSet inherits from Set behave identically on sorted sets and normal sets with two exceptions:

The Iterator returned by the iterator operation traverses the sorted set in order.

The array returned by toArray contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the toString method of the JDK's SortedSet implementations returns a string containing all the elements of the sorted set, in order.

SortedSet Standard Constructors

The SortedSet has a constructor that takes a Collection as an argument and creates a SortedSet object that orders its elements according to their natural order. Additionally, by convention, SortedSet implementations provide two other standard constructors:

One that takes a Comparator and returns a new (empty) SortedSet sorted according to the specified Comparator.

One that takes a SortedSet and returns a new SortedSet containing the same elements as the given SortedSet, and sorted according to the same Comparator (or using the elements' natural ordering, if the specified SortedSet did too). Note that the compile-time type of the argument determines whether this constructor is invoked in preference to the ordinary Set constructor, and not the runtime type!

The first of these standard constructors is the normal way to create an empty SortedSet with an explicit Comparator. The second is similar in spirit to the standard Collection constructor: it creates a copy of a SortedSet with the same ordering, but with a programmer-specified implementation type.

Range-view Operations

The Range-view operations are analogous to those provided by the List interface.

Range-views of a sorted set remain valid even if the backing sorted set is modified directly. This is because the endpoints of a range view of a sorted set are absolute points in the element-space, rather than specific elements in the backing collection. A range-view of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element-space. Changes to the range-view write back to the backing sorted set and vice-versa.

Sorted sets provide three range-view operations.

subSet takes two endpoints that are objects that are comparable to the elements in the sorted set. The range is half-open, including its low endpoint but excluding the high one. Thus, the following one-liner tells you how many words between "doorbell" and "pickle", including "doorbell" but excluding "pickle", are contained in a SortedSet of strings called dictionary:

```
dictionary.subSet("doorbell", "pickle").size();
```

Similarly, the following one-liner removes all of the elements beginning with the letter "f":

```
dictionary.subSet("f", "g").clear();
```

A table telling you how many words begin with each letter:

```
for (char ch='a'; ch<='z'; ch++) {
  String from = new String(new char[] {ch});
  String to = new String(new char[]
{(char)(ch+1)});
  System.out.println(from + ": " +
                     dictionary.subSet(from,
to).size());
}
```

Range-view Operations

The SortedSet interface contains two more range-view operations, `headSet` and `tailSet`, both of which take a single Object argument.

- `headSet` returns a view of the initial portion of the backing SortedSet, up to but not including the specified object.
- `tailSet` returns a view of the final portion of the the backing SortedSet, beginning with the specified object, and continuing to the end of the backing SortedSet,

Thus, the following code allows you to view the dictionary as two disjoint "volumes" (a-m and n-z):

```
SortedSet volume1 = dictionary.headSet("n");
SortedSet volume2 = dictionary.tailSet("n");
```

Endpoint Operations

The SortedSet interface contains operations to return the first and last elements in the sorted set, called (not surprisingly) `first` and `last`.

The SortedMap Interface

A SortedMap is a Map that maintains its entries in ascending order, sorted according to the keys' natural order, or according to a Comparator provided at SortedMap creation time. (Natural order and Comparators are discussed in the section on Object Ordering.) In addition to the normal Map operations, the Map interface provides operations for:

- Range-view: Performs arbitrary range operations on the sorted map.
- Endpoints: Returns the first or last key in the sorted map.
- Comparator access: Returns the Comparator used to sort the map (if any).

```
public interface SortedMap<K,V> extends Map<K,V> {
    Comparator<? super K> comparator();

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

    K firstKey();
    K lastKey();
}
```

- This interface is the Map analogue of SortedSet.

SortedMap Operations

The operations that SortedMap inherits from Map behave identically on sorted maps and normal maps with two exceptions:

- The Iterator returned by the iterator operation on any of the sorted map's Collection-views traverse the collections in key-order.
- The arrays returned by the Collection-views' toArray operations contain the keys, values, or entries in key-order.

Although it isn't guaranteed by the interface, the toString method of the Collection-views in all the JDK's SortedMap implementations returns a string containing all the elements of the view, in key-order.

SortedMap Standard Constructors

SortedMap provides a standard constructor that takes a Map, and creates a SortedMap object that orders its entries according to their keys' natural order. Additionally, by convention, SortedMap implementations provide two other standard constructors:

- One that takes a Comparator and returns a new (empty) SortedMap sorted according to the specified Comparator.
- One that takes a SortedMap and returns a new SortedMap containing the same mappings as the given SortedMap, and sorted according to the same Comparator (or using the elements' natural ordering, if the specified SortedMap did too).

The first of these standard constructors is the normal way to create an empty SortedMap with an explicit Comparator. The second creates a copy of a SortedMap with the same ordering, but with a programmer specified implementation type.

The following snippet illustrates how this works:

```
final Comparator FUNNY_COMPARATOR = ... ;
Map m = new TreeMap(FUNNY_COMPARATOR);

// ... code to populate m

Map m2 = new TreeMap(m); // invokes TreeMap(Map)
Map m3 = new TreeMap((SortedMap)m)
                   // invokes TreeMap(SortedMap)
```

Arrays

It is too bad that arrays are not collections

- You loose all of the power provided by the collection framework

The class `Arrays` contains

- various methods for manipulating arrays (such as sorting and searching)
- It also contains methods that allows arrays to be viewed as lists.

Implementations

Implementations are the actual data objects used to store collections, which implement the core collection interfaces described in the previous lesson. The sections that follow describe three kinds of implementations:

General-purpose Implementations

- General-purpose implementations are the public classes that provide the primary implementations of the core collection interfaces.

Wrapper Implementations

- Wrapper implementations are used in combination with other implementations (often the general-purpose implementations) to provide added functionality.

Convenience Implementations

- Convenience implementations are mini-implementations, typically made available via static factory methods that provide convenient, efficient alternatives to the general-purpose implementations for special collections (like singleton sets).

Additionally, you can build your own implementations, based on the JDK's abstract implementations.

General Purpose Implementations

The general-purpose implementations are summarized in the table below. The table highlights their regular naming pattern: names are all of the form <Implementation> <Interface>, where <Interface> is the core collection interface implemented by the class, and <Implementation> signifies the data structure underlying the implementation.

Interface	Set	List
Implementation	HashSet	
		ArrayList
	TreeSet	
		LinkedList

The primary implementations are HashSet, ArrayList and HashMap. SortedSet and SortedMap interfaces do not have rows in the table above. Each of these interfaces has one implementation and these implementations (TreeSet and TreeMap) are listed in the Set and Map rows.

Set

The three general purpose Set implementations are HashSet, TreeSet, and LinkedHashSet. HashSet is much faster (constant time vs. log time for most operations), but offers no ordering guarantees. If you need to use the operations in the SortedSet, or in-order iteration is important to you, use TreeSet. Otherwise, use HashSet.

LinkedHashSet is a HashSet with a linked list running through it so you can iterate through the set in insertion order.

List

The two general purpose List implementations are ArrayList and LinkedList. Most of the time, you'll probably use ArrayList. It offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method System.arraycopy when it has to move multiple elements at once.

If your List is fixed in size (that is, you'll never use remove, add or any of the bulk operations other than containsAll) you should consider using Arrays.asList().

Map

The three general purpose Map implementations are HashMap, TreeMap, and LinkedHashMap. If you need SortedMap operations or in-order Collection-view iteration, go for TreeMap; otherwise, go for HashMap. LinkedHashMap is faster than HashMap and provides insertion-order iteration.

What About User Objects?

The Collections framework will work with any Java class

You need to be sure you have defined

- `equals()`
- `hashCode()`
- `compareTo()`

Don't use mutable objects for keys in a Map

`hashCode()`

`hashCode()` returns randomized integers for distinct objects.

If two objects are equal according to the `equals()` method, then the `hashCode()` method on each of the two objects must produce the same integer result.

When `hashCode()` is invoked on the same object more than once, it must return the same integer, provided no information used in equals comparisons has been modified.

It is not required that if two objects are unequal according to `equals()` that `hashCode()` must return distinct integer values.

Filling a HashTable and using a reasonable hashfunction

```
public int hashCode() {
  return 31*super.firstName.hashCode()
         + super.lastName.hashCode();
 }
```

```
import java.util.*;

public class Hash_1 extends Name_1 {
  static final int MAX = 20000;
  static HashMap aHashMap = new HashMap();

  public Hash_1(String firstName,String lastName){
        super(firstName, lastName);
  }

  public static void init()   {
    long milliSeconds =
                  System.currentTimeMillis();
    for(int index = 0; index <= MAX; index ++) {
      if ( index % 1000  == 0 )
        System.out.println(index + "/" + MAX );
      aHashMap.put(new Hash_1("A"+index,
                 "A"+index),
                 new Hash_1("A"+index,
                 "A"+index));
    }
    System.out.println("Time for filling: "
      +(System.currentTimeMillis()-milliSeconds));
  }

  public static void findIt(Hash_1 aHash_1) {
    long milliSeconds =
System.currentTimeMillis();
    if ( aHashMap.containsKey( aHash_1 ) )
      System.out.print(
             "\taHashMap: containsKey takes: ");
    System.out.println(System.currentTimeMillis()
                     - milliSeconds);
  }

  public static void findMax() {
    Hash_1 aHash_1 = new Hash_1( "A"+MAX,
"A"+MAX);
    System.out.println("Find Max = " + aHash_1);
    findIt(aHash_1);
  }

  public static void findMiddle() {
    Hash_1 aHash_1=new Hash_1("A"+MAX/2,
"A"+MAX/2);
    System.out.println("Find Middle = " +
aHash_1);
    findIt(aHash_1);
  }

  public static void findMin()        {
    Hash_1 aHash_1 = new Hash_1( "A" + 0, "A" +
0);
    System.out.println("Find Min = " + aHash_1);
    findIt(aHash_1);
  }

  public static void main(String args[] )      {
    long milliSeconds =
System.currentTimeMillis();

    init();
    findMax();
    findMiddle();
    findMin();
    System.exit(0);
  }
}
```

```
Time for filling: 1638
Find Max = A20000 A20000
        aHashMap: containsKey takes: 0
Find Middle = A10000 A10000
        aHashMap: containsKey takes: 0
Find Min = A0 A0
        aHashMap: containsKey takes: 0
```

Filling a HashTable and not using a reasonable hashfunction

```
  public int hashCode() {
    return 1;
  }

import java.util.*;

// Same code
// ...

    init();
    findMax();
    findMiddle();
    findMin();
    System.exit(0);
  }
}

% java Hash_2
Time for filling: 282381
Find Max = A20000 A20000
        aHashMap: containsKey takes: 1
Find Middle = A10000 A10000
        aHashMap: containsKey takes: 13
Find Min = A0 A0
        aHashMap: containsKey takes: 10
```

Wrapper Implementations

Wrapper implementations are implementations that delegate all of their real work to a specified collection, but add some extra functionality on top of what this collection offers.

Synchronization Wrappers

Unmodifiable Wrappers

Synchronization Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. There is one static factory method for each of the six core collection interfaces:

```
public static <T> Collection<T>
        synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T>
s);
public static <T> List<T>
        synchronizedList(List<T> list);
public static <K,V> Map<K,V>
        synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T>
        synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V>
        synchronizedSortedMap(SortedMap<K,V> m);
```

Each of these methods returns a synchronized (thread-safe) Collection backed by the specified collection. In order to guarantee serial access, it is critical that all access to the backing collection is accomplished through the returned collection. The easy way to guarantee this is to not to keep a reference to the backing collection. Creating the synchronized collection like this does the trick:

```
List<X> list = Collections.synchronizedList(
                    new ArrayList<X>());
```

Unmodifiable Wrappers

The unmodifiable wrappers are conceptually similar to the synchronization wrappers, but simpler. Rather than adding functionality to the wrapped collection, they take it away. In particular, they take away the ability to modify the collection, by intercepting all of the operations that would modify the collection, and throwing an UnsupportedOperationException. The unmodifiable wrappers have two main uses:

To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.

To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper. In this way, the second-class citizens can look but not touch, while you maintain full access.

Like the synchronization wrappers, there is one static factory method for each of the six core collection interfaces:

```
public static <T> Collection<T>
unmodifiableCollection(Collection<T> c);
public static <T> Set<T> unmodifiableSet(Set<T>
s);
public static <T> List<T>
        unmodifiableList(List<T> list);
public static <K,V> Map<K,V>
        unmodifiableMap(Map<K,V> m);
public static <T> SortedSet<T>
        unmodifiableSortedSet( SortedSet<T> s);
public static <K,V> SortedMap<K,V>
        unmodifiableSortedMap(SortedMap<K,V> m);
```

List-view of an Array

The Arrays.asList method returns a List-view of its array argument. Changes to the List write through to the array and vice-versa. The size of the collection is that of the array, and cannot be changed. If the add or remove method is called on the List, an UnsupportedOperationException will result.

The normal use of this implementation is as a bridge between array-based and collection-based APIs. It allows you to pass an array to a method expecting a Collection or a List. However, this implementation has another use. If you need a fixed-size List, it's more efficient than any general-purpose List implementation. Here's the idiom:

```
List<X> l = Arrays.asList(new X[size]);
```

Note that a reference to the backing array is not retained.

Immutable Multiple-Copy List

Occasionally you'll need an immutable List consisting of multiple copies of the same element. The Collections.nCopies method returns such a List. This implementation has two main uses. The first is to initialize a newly created List. For example, suppose you want an ArrayList initially consisting of 1000 null elements. The following incantation does the trick:

```
List<X> l = new ArrayList<X>(
              Collections.nCopies(1000, (X)null));
```

Of course, the initial value of each element needn't be null. The second main use is to grow an existing List. For example, suppose you want to add 69 copies of the string “fruit bat” to the end of a List. It's not clear why you'd want to do such a thing, but let's just suppose you did. Here's how you'd do it:

```
lovablePets.addAll(
          Collections.nCopies(69, "fruit bat"));
```

By using the form of addAll that takes an index as well as a Collection, you can add the new elements to the middle of a List instead of at the end.

Immutable Singleton Set

Sometimes you'll need an immutable singleton Set, which consists of a single, specified element. The Collections.singleton method returns such a Set. One use of this implementation is this idiom to remove all occurrences of a specified element from a Collection:

```
c.removeAll(Collections.singleton(e));
```

There's a related idiom to remove from a Map all elements that map to a specified value. For example, suppose you have a Map, profession, that maps people to their line of work. Suppose you want to eliminate all of the lawyers. This one-liner will do the deed:

```
profession.values().removeAll(
                   Collections.singleton(LAWYER));
```

One more use of this implementation is to provide a single input value to a method that is written to accept a Collection of values.

Empty Set and Empty List Constants

The Collections class provides two constants, representing the empty Set and the empty List, Collections.EMPTY_SET and Collections.EMPTY_LIST. 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.

Algorithms

The Collections class provides several useful static methods. All take the form of static methods whose first argument is the collection on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on List objects, but a couple of them (min and max) operate on arbitrary Collection objects.

- Shuffling
- Data manipulation
  - `reverse()`
  - `fill()`
  - `copy()`
- Searching
- Finding extreme values
  - `max()`
  - `min()`

Sorting

The sort algorithm reorders a List so that its elements are ascending order according to some ordering relation. Two forms of the operation are provided. The simple form just takes a List and sorts it according to its elements' natural ordering.

The sort operation is:

- Fast: This algorithm is guaranteed to run in n log(n) time, and runs substantially faster on nearly sorted lists.
- Stable: It doesn't reorder equal elements. If a user of a mail program sorts his in-box by mailing date, and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is only guaranteed if the second sort was stable.

Here's a trivial little program that prints out its arguments in lexicographic (alphabetical) order.

```
import java.util.*;

public class Sort {
    public static void main(String args[]) {
        List<String> l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

Let's run the program:

```
% java Sort i walk the line

[i, line, the, walk]
```

A second form of sort takes a Comparator in addition to a List and sorts the elements with the Comparator.

Shuffling

The shuffle algorithm does the opposite of what sort does: it destroys any trace of order that may have been present in a List.

There are two forms of this operation

- The first just takes a List and uses a default source of randomness.
- The second requires the caller to provide a Random object to use as a source of randomness.

Routine Data Manipulation

The Collections class provides three algorithms for doing routine data manipulation on List objects. All of these algorithms are pretty straightforward:

- `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 re-initializing 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.

Searching

The binarySearch algorithm searches for a specified element in a sorted List using the binary search algorithm.

- The first takes a List and an element to search for (the "search key"). This form assumes that the List is sorted into ascending order according to the natural ordering of its elements.
- The second form of the call takes a Comparator in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator.

The return value is the same for both forms:

- If the List contains the search key, its index is returned.
- If not, the return value is

```
( - ( insertion point ) - 1)
```

where the insertion point is defined as the point at which the value would be inserted into the List: the index of the first element greater than the value, or list.size() if all elements in the List are less than the specified value. This admittedly ugly formula was chosen to guarantee that the return value will be >= 0 if and only if the search key is found. It's basically a hack to combine a boolean ("found") and an integer ("index") into a single int return value.

Finding Extreme Values

The min and max algorithms return, respectively, the minimum and maximum element contained in a specified Collection. Both of these operations come in two forms. The simple form takes only a Collection, and returns the minimum (or maximum) element according to the elements' natural ordering. The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

API Design

In-Parameters

If your API contains a method that requires a collection on input, declare the relevant parameter type to be one of the collection interface types. Never use an implementation type.

Always use the least specific type that makes sense. Generally, the best types to use on input are the most general: Collection and Map.

Return Values

It's fine to return an object of any type that implements or extends one of the collection interfaces.