Java Collections Framework

Objectives

- Know how to use the Java Collections Framework API
- Know how to choose the appropriate collection classes
- Know how to adapt collection classes to your application

History

- **Container classes in JDK 1.1.x**
  - `Vector` **class**
  - `Hashtable` **class**
  - `Enumeration` **Interface**
- **Java 2 Collections API starting with 1.2**
- **Generics added in Java 5**

The Collections Framework

- Collections are objects designed to store other objects in useful ways
- The Java Collections Framework is based on a few interfaces and several implementations
- Most of the classes we will use are in the `java.util` package

Java Collections API

- Using the collections API means coding to interfaces and choosing an implementation
- The interface indicates what you want to do
- The implementation indicates how

Collections

- A collection represents a group of objects, known as its elements.
  - Duplicate elements may or may not be allowed.
  - A collection may be ordered or unordered.
  - Classes and interfaces are provided for different types of collections and traversing the items in the list.

Collection Implementations

- **The concrete classes implementing the collection interfaces have the following form to their name**

<Implementation-style><Interface>

<table>
<tr><th></th><th colspan="6">Implementation</th></tr>
<tr><th>Interface</th><th>Hash Table</th><th>Resizable Array</th><th>Priority Heap</th><th>Balanced Tree</th><th>Linked List</th><th>HashTable + Linked List</th></tr>
<tr><td>List</td><td></td><td>ArrayList</td><td></td><td></td><td rowspan="2">LinkedList</td><td></td></tr>
<tr><td>Deque**</td><td></td><td rowspan="2">ArrayDeque**</td><td></td><td></td><td></td></tr>
<tr><td>Queue*</td><td></td><td>PriorityQueue*</td><td></td><td></td><td></td></tr>
<tr><td>Set</td><td>HashSet</td><td></td><td></td><td rowspan="3">TreeSet</td><td></td><td>LinkedHashSet</td></tr>
<tr><td>SortedSet</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>NavigableSet**</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

*** Introduced in Java 5 (1.5)**

**** Introduced in Java 6 (1.6)**

Iteration Interfaces

`Iterator<E> & Iterable<T>`

- **The** `Iterator` **interface provides a standard way to iterate over the objects in a collection.**
- **The** `java.lang.Iterable` **interface provides a single operation for obtaining an object's** `Iterator`.

  `Iterator<T> iterator()`

  - **All container classes implement the** `Iterable` **interface.**
  - **Arrays also implement** `Iterable`
  - **Only** `Iterable` **objects can be used in the enhanced** `for` **loop**

Iteration Interfaces

`Iterator<E>` Methods

```
boolean hasNext()

E next()

void remove()
```

- **Typically utilized to access a collection of objects as follows:**

```
// assuming a collection of strings

Iterator<String> it = someCollection.iterator();
while( it.hasNext() ) {
   String s = it.next();

   ...

}


// or...
for( String s; someCollection) {
   s.whatever();

   ...

}
```

Collection<E>

- **The** `Collection` **interface represents collections in a general way**
  - **Serves as a base interface from which more restrictive collections are extended.**

```
java.util.Collection
├ java.util.List
├ java.util.Queue
│ └ java.util.Deque
└ java.util.Set
  └ java.util.SortedSet
    └ java.util.NavigableSet
```

Collection<E> Operations

```
boolean add( E e )
boolean addAll( Collection<? Extends E> c )
void clear()
boolean contains( Object o )
boolean containsAll(Collection<? Extends E> c )
boolean isEmpty()
Iterator<E> iterator()
boolean remove( Object o )
boolean removeAll( Collection<?> c )
boolean retainAll( Collection<?> c )
int size()
Object[] toArray()
<T> T[] toArray( T[] a )
```

List<E>

- An ordered collection.
  - Provides precise control over where in the list each element is inserted.
  - Elements may be accessed by their integer index.
  - Provides for searching for elements in the list.
  - Typically allow duplicate elements.

List<E> Operations

```
void add( int index, E element )
boolean add( E e )
boolean addAll( Collection<? Extends E> c )
boolean addAll( int index,
                Collection<? Extends E> c )
E get( int index )
int indexOf( Object o )
int lastIndexOf( Object o )
Object remove( int index )
Object set( int index, Object element )
List<E> subList( int fromIndex, int toIndex )
```

Queue<E>

- A collection for holding work prior to processing.
  - Provides additional operations for insertion, extraction and inspection
  - Two variants for these operations:
    - One throws an exception if operation fails
    - The other returns a special value (`null` or `false`) if the operation fails
      - Intended for use with capacity restricted implementations
  - Capacity may be restricted
  - Typically FIFO, but may support `Comparator`/`Comparable` ordering
  - Introduced in Java 5

Queue<E> Operations

```
boolean offer(E element )
boolean add( E element ) throws IllegalStateException

E peek()
E element() throws NoSuchElementException

E poll()
E remove() throws NoSuchElementException
```

Deque<E>

- A double ended queue
  - Extends the `Queue` interface with additional operations for insertion, extraction and inspection at both ends
  - Like `Queue`, two variants for these operations:
    - One throws an exception if operation fails
    - The other returns a special value (`null` or `false`) if the operation fails
      - Intended for use with capacity restricted implementations
  - Capacity may or may not be restricted
  - Pronounced “deck”
  - Introduced in Java 6

Deque<E> Operations

```
boolean offerFirst(E element )
boolean addFirst( E element ) throws IllegalStateException
boolean offerLast(E element )
boolean addLast( E element ) throws IllegalStateException

E peekFirst()
E getFirst() throws NoSuchElementException
E peekLast()
E getLast() throws NoSuchElementException

E pollFirst()
E removeFirst() throws NoSuchElementException
E pollLast()
E removeLast() throws NoSuchElementException
```

Set<E>

- A collection that contains no duplicate elements.
  - Models the mathematical set abstraction.
  - Specifies no operations beyond those of the `Collection` interface.
  - Care must be used when using mutable objects as elements
  - May not allow `null` element

Comparable<T> and Comparator<T>

- **Two interfaces are provided to enable sorting**
  - **The java.lang.`Comparable<T>` interface determines a classes "natural ordering"**

    `int compareTo(T o)`

    - Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object
    - **Should be consistent with equals method**
      - For every e1 and e2 of the class the following holds

        `(e1.compareTo(e2) == 0) == e1.equals(e2)`
  - **The `Comparator<T>` interface defines an abstract operation for comparing two objects for order.**

    `int compare(T o1, T o2)`

    - Returns negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second
    - **Should be consistent with equals method**

      `(compare(x, y) == 0) == x.equals(y)`

SortedSet<E>

- Guarantees that its iterator will traverse the set in ascending element order.
  - Provides operations for accessing first and last elements, and obtaining restricted range subsets
  - Sorted according to the an ordering determined by a `Comparator` provided at creation time.
  - Absent a `Comparator` uses "natural ordering"

SortedSet<E> Operations

```
Comparator<? super E> comparator()
E first()
E last()
SortedSet<E> subSet( E fromElement, E toElement )
SortedSet<E> headSet( E toElement )
SortedSet<E> tailSet( E fromElement )
```

NavigableSet<E>

- **Extends** SortedSet **with support for closest match searching and reversed traversal**
  - **Provides operations for reporting the closest match to a search target, and viewing the set in reversed order**

NavigableSet<E> Operations

```
E ceiling( E e )
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
E floor( E e )
E higher( E e )
E lower( E e )
E pollFirst()
E pollLast()
```

Collection Implementations

- **The concrete classes implementing the collection interfaces have the following form to their name**

`<Implementation-style><Interface>`

<table>
<tr><th></th><th colspan="6">Implementation</th></tr>
<tr><th>Interface</th><th>Hash Table</th><th>Resizable Array</th><th>Priority Heap</th><th>Balanced Tree</th><th>Linked List</th><th>HashTable + Linked List</th></tr>
<tr><td>List</td><td></td><td>ArrayList</td><td></td><td></td><td rowspan="2">LinkedList</td><td></td></tr>
<tr><td>Deque**</td><td></td><td rowspan="2">ArrayDeque**</td><td></td><td></td><td></td></tr>
<tr><td>Queue*</td><td></td><td>PriorityQueue*</td><td></td><td></td><td></td></tr>
<tr><td>Set</td><td>HashSet</td><td></td><td></td><td rowspan="3">TreeSet</td><td></td><td>LinkedHashSet</td></tr>
<tr><td>SortedSet</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>NavigableSet**</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

*** Introduced in Java 5 (1.5)**

**** Introduced in Java 6 (1.6)**

`ArrayList<E>`

- **Implementation of a growable array of objects.**
  - **Like an array, contains components that can be accessed using an integer index.**
  - **Permits `null` elements**
  - **No size restrictions, grows as necessary**
  - **The most commonly used collection.**
  - **O(1) time for:**
    `isEmpty`, `get`, `set`, **and** `size`
  - **Amortized O(1) time for:**
    `add`

`ArrayDeque<E>`

- **Implementation of a growable array of objects.**
  - **Like an array, contains components that can be accessed using an integer index.**
  - **The `null` element is prohibited**
  - **No size restrictions, grows as necessary**
  - **Amortized O(1) time for:**

    Most operations
  - **O(n) time for:**

    `remove`, `removeFirstOccurance`, `removeLastOccurance`, **and** `contains`

`PriorityQueue<E>`

- **Implementation of a heap**
  - **Ordered by** `Comparator` **or "natural-ordering"**
  - **Like an array, contains components that can be accessed using an integer index.**
  - **The** `null` **element is prohibited**
  - **No size restrictions, grows as necessary**
  - **O(log(n)) time for:**
    `add`, `offer`, `poll`, **and** `remove()`
  - **O(n) time for:**
    `contains`, **and** `remove(Object)`
  - **O(1) time for:**
    `peek`, `element`, **and** `size`

`LinkedList<E>`

- **Implementation of a doubly-linked list**
  - **Operations allow linked lists to be used as a stack, queue, or double-ended queue**
  - **Permits `null` elements**
  - **No size restrictions, grows as necessary**
  - **O(1) time for:**

    Operations at the ends of the list (`addFirst`, `addLast`, `getFirst`, `getLast`, …)
  - **O(n) time for:**

    Operations utilizing an index

HashSet<E>

- A hashtable implementation of an unordered set.
  - No guarantee of iteration order or that the order will remain constant over time.
  - Permits `null` elements
  - No size restrictions, grows as necessary
  - O(1) time for:

    `add`, `remove`, `contains`, and `size`

LinkedHashSet<E>

- **A set implementation using a doubly-linked list and hashtable.**
  - **The linked list defines insures insertion-order iteration**
    - **Insertion order is not altered my multiple insertions**
  - **Permits `null` elements**
  - **No size restrictions, grows as necessary**
  - **O(1) time for:**
    `add`, `remove`, **and** `contains`
    - **Performance is slightly worse than that of `HashSet`**
      - Iteration performance is likely to be better than that of `HashSet`

TreeSet<E>

- **Red-Black tree based set implementation.**
  - **Ordered by** `Comparator` **or "natural-ordering"**
  - **Permits** `null` **elements**
  - **No size restrictions, grows as necessary**
  - **O(log(n)) time for:**
    `add`, `remove`, **and** `contains`

Map Implementations

- **The concrete classes implementing the map interfaces have the following form to their name**

  <Implementation-style><Interface>

	Implementation		
Interface	Hash Table	Balanced Tree	HashTable + Linked List
Map	HashMap	TreeMap	LinkedHashMap
SortedMap			
NavigableMap**			

*** Introduced in Java 5 (1.5)**

**** Introduced in Java 6 (1.6)**

Map

- Serves as the root of the map interface hierarchy.

```
java.util.Map
└ java.util.SortedMap
  └ java.util.NavigableMap
```

Map<K,V>

- An interface for mapping keys to values.
  - Prohibits duplicate keys
  - Each key can map to at most one value.
  - Provides three collection views
    - Set of keys
    - Collection of values
    - Set of key-value mappings
  - Extreme care must be taken if using mutable objects for key

Map<K,V> Operations

```
void clear()
boolean containsKey( Object key )
boolean containsValue( Object value )
Set<Map.Entry<K,V>> entrySet()
V get( Object key )
boolean isEmpty()
Set<K> keySet()
V put( K key, V value )
void putAll( Map<? extends K,? extends V>  m )
V remove( Object key )
int size()
Collection<V> values()
```

Maps and `hashCode` & `equals`

- Maps rely on the `hashCode` method of the key objects to determine placement in the map
- The `hashCode` and `equals` methods must be consistent
  - When overriding `hashCode`, its general contract must be met
    - Multiple invocations on the same object must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified
    - If two objects are equal according to the `equals` method, then `hashCode` must produce the same integer result for both objects.
    - If two objects are unequal according to the `equals` method, then hashCode need not return distinct integer results for each object.
      - However, producing distinct integer results for unequal objects may improve the performance of map implementations.

Maps and `hashCode` & `equals`

- **When overriding `equals`, its general contract must be met**
  - **It is *reflexive*:**
    - For any non-null reference value `x`, `x.equals(x)` should return `true`.
  - **It is *symmetric*:**
    - For any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
  - **It is *transitive*:**
    - For any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
  - **It is *consistent*:**
    - For any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
  - **For any non-null reference value `x`, `x.equals(null)` should return `false`.**

SortedMap<K,V>

- Provides a total ordering on its keys
  - Sorted according to the an ordering determined by a `Comparator` provided at creation time.
  - Absent a `Comparator` uses "natural ordering"

SortedMap<K,V> Operations

```
Comparator<? super K> comparator()
K firstKey()
K lastKey()
SortedMap<K,V> subMap( K fromKey, K toKey )
SortedMap<K,V> headMap( K toKey )
SortedMap<K,V> tailMap( K fromKey )
```

NavigableMap<K,V>

- **Extends** `SortedMap` **with support for closest match searching and reversed traversal**
  - **Provides operations for reporting the closest match to a search target, and viewing the set in reversed order**

NavigableMap<K,V> Operations

```
Map.Entry<K,V> ceilingEntry( K key )
Map.Entry<K,V> lowerEntry( K key )
Map.Entry<K,V> floorEntry( K key )
Map.Entry<K,V> higherEntry( K key )
Map.Entry<K,V> firstEntry()
Map.Entry<K,V> pollFirstEntry()
Map.Entry<K,V> lastEntry()
Map.Entry<K,V> pollLastEntry()

K ceilingKey( K key )
K lowerKey( K key )
K floorKey( K key )
K higherKey( K key )

NavigableSet<K> descendingKeySet()
NavigableMap<K,V> descendingMap()
```

Map Implementations

- **The concrete classes implementing the map interfaces have the following form to their name**

  <Implementation-style><Interface>

	Implementation		
Interface	Hash Table	Balanced Tree	HashTable + Linked List
Map	HashMap	TreeMap	LinkedHashMap
SortedMap			
NavigableMap**			

*** Introduced in Java 5 (1.5)**

**** Introduced in Java 6 (1.6)**

HashMap<K,V>

- **Hashtable implementation**
  - **No guarantee of iteration order or that the order will remain constant over time.**
  - **Permits `null` values and the `null` key**
  - **No size restrictions, grows as necessary**
  - **O(1) time for:**

    `get`, **and** `put`

TreeMap<K,V>

- **Red-Black tree based implementation of NavigableMap.**
  - **Ordered by `Comparator` or "natural-ordering" of keys**
  - **Permits `null` values and the `null` key**
  - **No size restrictions, grows as necessary**
  - **O(log(n)) time for:**

    `containsKey`, `get`, `put` **and** `remove`

LinkedHashMap<K,V>

- **A map implementation using a doubly-linked list and hashtable.**
  - **The linked list defines insures insertion-order iteration**
    - **Insertion order is not altered my multiple insertions**
  - **Permits `null` values and the `null` key**
  - **No size restrictions, grows as necessary**
  - **O(1) time for:**

    `get`, **and** `put`
    - **Performance is slightly worse than that of `HashSet`**
      - Iteration performance is likely to be better than that of `HashSet`

The `Collections` and `Arrays` Classes

- `Collections` **utility class contains methods to modify** `Collection` **class objects**
  - sort elements
  - Search for elements
  - Reverse elements
  - Unmodifiable collection
  - Provide synchronized access and read-only collections
  - Randomize (shuffle) elements
  - etc. (study the javadoc)
- `Arrays` **utility class has many of the same methods for arrays**

Collections

- **Provides a variety of collection utility methods**
  - **Sorting, methods sort the list in place**

```
<T extends Comparable<? super T>> void sort(List<T> list)
<T> void sort(List<T> list, Comparator<? super T> c)
```

  - **Searching, searches list for specified object using binary search**

```
<T> int binarySearch(
        List<? extends Comparable<? super T>> list, T key)
<T> int binarySearch( List<? extends T> list, T key,
                      Comparator<? super T> c)
```

Collections

- **Max and min, methods return the maximum or minimum element**

```
<T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
<T> T max(Collection<? extends T> coll,
          Comparator<? super T> comp)
<T extends Object & Comparable<? super T>>
    T min(Collection<? extends T> coll)
<T> T min(Collection<? extends T> coll,
          Comparator<? super T> comp)
```

Collections

- **Synchronization, methods return a synchronized wrapper of the operand**

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

Collections

– **Unmodifiability, methods return an unmodifiable view or operand**

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

Programming Tips and Design

- **Program in terms of interface types rather than implementation types:**
  - `List list = new ArrayList();`
- **preferable to**
  - `ArrayList list = new ArrayList();`

Programming Tips and Design(cont’d)

- Passing collection types as parameters to and return types from methods
  - Use the least specific type to promote generality
  - Examples:
    - `Collection` instead of `List`
    - `List` instead of `ArrayList`

Strategy Patterns

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

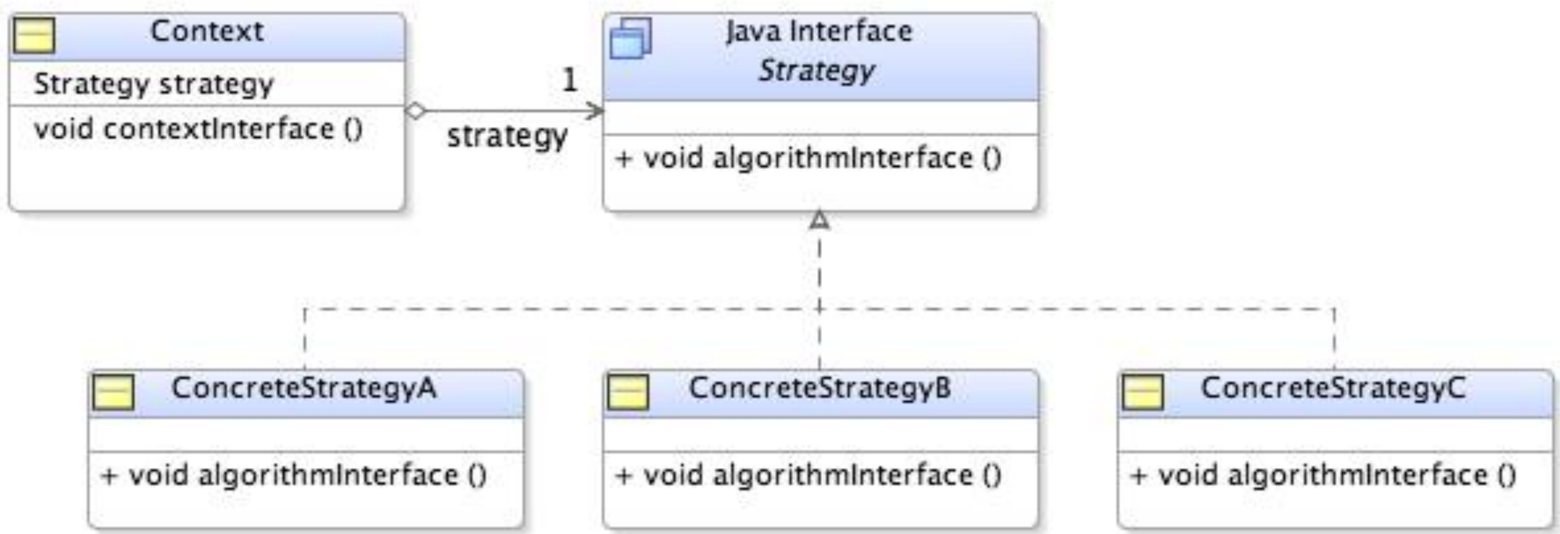
Strategy Pattern
Applicability

- Use the strategy pattern when:
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - Different variants of an algorithm are needed. For example, algorithms might be defined reflecting different space/time trade-offs.
  - An algorithm uses data that the client shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

Strategy Pattern
Applicability

- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many, conditionals move related conditional branches into their own Strategy class.

Strategy Pattern



Strategy Pattern
Responsibilities

- `Strategy`
  - Declares an interface common to all supported algorithms. `Context` uses this interface to call the algorithm defined by a `ConcreteStrategy`.
- `ConcreteStrategy` subclasses
  - Implements the algorithm using the `Strategy` interface.

Strategy Patterns
Responsibilities

- `Context`
  - Is configured with a `ConcreteStrategy` object
  - Maintains a reference to a `Strategy` object.
  - May define an interface that lets `Strategy` access its data.

Comparing Objects

- `java.lang.Comparable` **interface**
  - **Designed to sort objects into "natural ordering"**
  - **Implemented by String and wrapper classes**
  - **Requires a single method:**

    ```
    int compareTo(<T> other);
    ```
  - **Returns negative int, 0, or positive int based on whether this object is less than, equal to , or greater than the other object**
- **If all you ever need is "natural ordering" just implement** `Comparable` **to use** `Collections.sort()`

Comparing Objects (cont’d)

- **Sometimes “natural ordering” isn’t enough; there may be several useful ways to sort**
- **In these cases, implement the** `java.util.Comparator` **interface in a class separate from the class you want to compare**
- **Two methods:**
  - `int compare(<T> first, <T> second)`
    - **Just like** `compareTo()`**, returns negative int, 0, or positive int to indicate order**
  - `boolean equals()` **// optional**
    - **If implemented determines if two** `Comparators` **provide the same ordering**

Comparator Example

```
import java.util.Comparator;

public class IntegerComparator implements Comparator<Integer> {
    public IntegerComparator() {
    }

    public int compare(final Integer arg0, final Integer arg1) {
        int diff = 0;
        if (arg0.intValue() > arg1.intValue()) {
           diff = 1;
       } else if (arg0.intValue() < arg1.intValue()) {
          diff = -1;
       }
       return diff;
    }
}
```