Collections, sorting, and searching in Java

- What is the collections API?
- Why use the collections API?
- Details of the API
- Summary

What is the Collections API?

Implementations of Common Collections

- Sets
  - HashSet
  - ArraySet
  - Hashtable - legacy support
- Lists (a.k.a. sequences)
  - ArrayList
  - LinkedList
  - Vector - legacy support
- Maps
  - HashMap
  - ArrayMap
  - TreeMap

A Framework for Multiple, Interoperating Implementations

- Interfaces representing various kinds of collections
  - Collections (generic collection API)
  - Lists (a.k.a. sequences)
  - Sets
  - Maps
  - Sorted sets
  - Sorted maps

Interfaces

- *Collection* - A group of Objects that may or may not be ordered, duplicate free, mutable, etc.
- *Set* - The familiar set abstraction
  - No duplicate elements permitted
  - Order (if any) is generally established at set creation time
  - Extends Collection
- *List* - Ordered collection
  - Duplicates are generally permitted
  - provides precise control over the position of each element
  - Extends *Collection*

Interfaces (cont)

- *Map* - A mapping from keys to values
- *SortedSet* - A *Set* whose elements are automatically sorted
- *SortedMap* - A *Map* whose mappings are automatically sorted by key

Abstract Classes

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

An Infrastructure for Object Ordering

- Classes can specify a “natural” ordering
- Other orderings can be created
- Support for sorting and searching
  - TreeMap
  - LinkedList & ArrayList
  - Collections.sort(…)
  - Collections.binarySearch(…)
  - Array sorting & searching via ‘Arrays’ static methods

An Infrastructure for Iterating Through a Collection

- *Iterator* - interface which replaces *Enumeration*
- *ListIterator* - *Iterator* that adds capabilities specific to lists
- *Enumeration* - legacy interface still supported by legacy collections like Vector and Hashtable

Why use the API?

Don’t Re-Invent the Wheel

- Use ‘Array.xxx’ methods to search and sort arrays of primitives and Objects
- Use existing collection implementations
- Reuse implementations many times with different sorting/searching criteria
- Leverage abstract implementations if need to implement a collection

Why use the API?

Allow for New Implementations

```
// Today
Map myMap = new HashMap(…);

// In the future
Map myMap = new AcmeSoftMap(…);

// The rest of the code stays the same
myMap.put( key1, object1 );
myMap.put( key2, object2 );
Iterator values = myMap.keySet().values()
while ( values.hasNext() )
{
    Object current = values.next();
    // Do work here
}
```

It’s Extensible

- For any class which has a ‘natural’ order, implement the *Comparable* interface

```
public class UnitedStatesPhoneNumber
implements Comparable
{
        // ...
}
```

Extensibility (cont)

- Can use a *Comparator* to
  - change ordering in sorted collections
  - perform sorts with different ordering criterum

```
// Reverse the natural ordering of U.S. phone numbers
public class ReverseUSPhoneNumberCompare
implements Comparator
{
        // ...
}
```

Details of the API

Details of the API

`Collection`

- A group of Objects that may or may not
  - be ordered
  - have duplicates
  - be mutable
  - etc.
- Important Methods
  - add(Object o)
  - addAll(*Collection* c)
  - remove(Object o)
  - removeAll(*Collection* c)
  - clear()
  - contains(Object o)
  - containsAll(*Collection* c)
  - iterator()
  - size()
  - retainAll(*Collection* c)
  - toArray()
  - toArray(Object[] a);

Set

- The familiar set abstraction
  - No duplicates
  - Order established at set creation time
  - Extends *Collection*
- Doesn’t add new methods (beyond *Collection)*, constrains what objects may be contained
- Objects can’t change while in the set (or undefined results occur)

SortedSet

- A sorted *Set*. Elements are ordered using
  - their natural ordering

  OR
  - by a *Comparator*
- Important new methods:
  - first() - return smallest element in set
  - headSet(Object endItem) - all items less than endItem
  - last() - return largest element
  - subSet(Object from, Object to)
  - tailSet(Object firstItem) - all elements >= firstItem

List

- Ordered *Collection*
  - Also known as a sequence
  - Duplicates generally permitted
  - Control position of each element
  - Extends *Collection*
- Adds many methods beyond *Collection*

List (cont)

- Important new methods:
  - add(int index, Object o)
  - addAll(int index, *Collection* c)
  - get(int index)
  - indexOf(Object o)
  - lastIndexOf(Object o)
  - listIterator()
  - listIterator(int index)
  - remove(int index)
  - removeRange(int fromIndex, int toIndex)
  - set(int index, Object element)

Map

- Mapping from keys to values
- Each key maps to at most one value
- Does <u>not</u> extend *Collection*
- Keys can’t change while in the set (or undefined results occur)
- Important methods
  - put(Object key, Object o)
  - putAll(Map m)
  - containsKey(Object key)
  - containsValue(Object value)
  - get(Object key)
  - remove(Object key)
  - size()
  - keySet()
  - values()

SortedMap

- A sorted *Map*. Entries sorted by key using:
  - keys' natural ordering

  OR

  - by a *Comparator*
- Important added methods
  - firstKey()
  - lastKey()
  - headMap(Object toKey)
  - subMap(Object fromKey, Object toKey)
  - tailMap(Object fromKey)
  - comparator()

Properties of Collection Interfaces

- Some methods are optional
- They are marked in the Javadoc as “(optional operation)”
- For details, see the Collections Design FAQ (or talk to me)

Collection Implementations

- HashSet - a *Set* backed by a hash table
  - General purpose *Set* implementation
- TreeSet - imposes an ordering on its elements
- ArrayList - Like a vector with no legacy code (e.g. unsynchronized for speed)
- LinkedList - a doubly-linked *List*

Collection Implementations (cont)

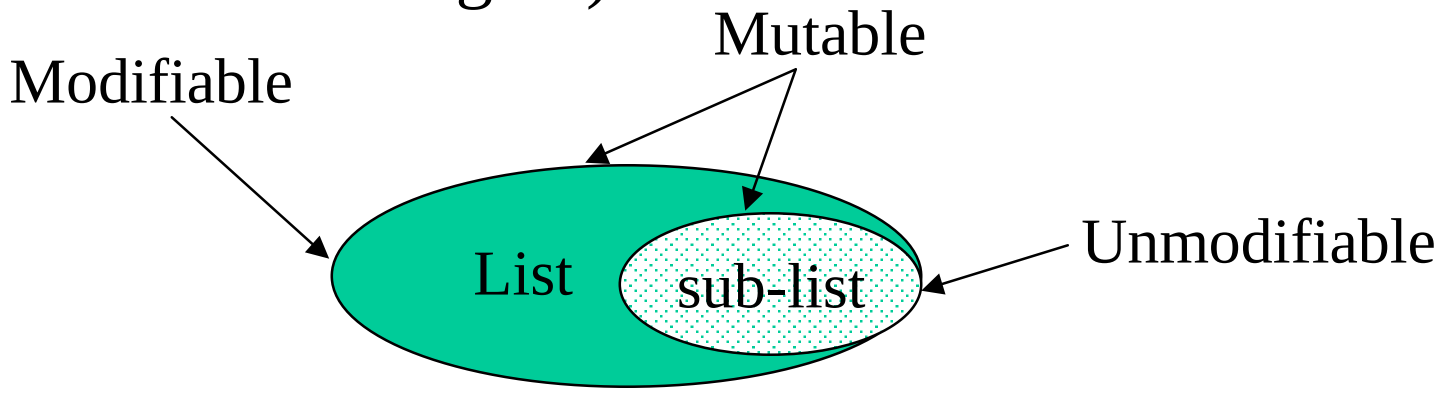
- HashMap - a hash table *Map*
- TreeMap - imposes an ordering on its entries
- Vector - legacy *List* implementation
- Hashtable - legacy *Map* implementation

Properties of Default Implementations

- Not synchronized
  - faster when don’t need synchronization
  - use Collections.synchronizedXXX to wrap synchronization around a collection
- Fixed- or Variable-sized
  e.g. Lists around native arrays are fixed-size;
  LinkedList is variable-sized
- May not allow Modification
  (UnsupportedOperationException)
- May not be mutable (changeable)

‘Modifiable’ vs. ‘Mutable’

- Some operations return a ‘view’ on a Collection (e.g. List.subList(...))
- Such a view may be unmodifiable (you can’t change it)
- The underlying collection may be mutable (someone can change it)



Iterators

- *Iterator* - An iterator over a *Collection*
  - takes the place of *Enumeration*
  - differ from enumerations
    - Can remove elements during iteration
    - Method names have been improved
- Methods of *Iterator*
  - hasNext() - returns true if the iteration has more elements
  - next() - returns the next element in the interation
  - remove() - removes last element returned by the *Iterator*

Iterators (cont)

- *ListIterator* - extends *Iterator*
  - move backwards as well as forwards
  - add and replace as well as remove
- Important new methods
  - add(Object o)
  - hasPrevious()
  - nextIndex()
  - previous()
  - previousIndex()
  - set(Object o)

Details of the API

Comparable

```
public class UnitedStatesPhoneNumber implements Comparable
{
    private int areaCode, prefix, exchange;
    // more code here…
    public int compareTo( Object anObject ) {
            // Generates a ClassCastException if the wrong type
            // of object is passed.
            UnitedStatesPhoneNumber toCompare =
                (UnitedStatesPhoneNumber)  anObject;
            int ordering = areaCode - toCompare.areaCode;
            if ( ordering == 0 ) {
                    ordering = prefix - toCompare.prefix;
                    if ( ordering == 0 ) {
                            ordering = exchange - toCompare.exchange;
                    }
            }
            return ordering;
    }
}
```

Comparator

```
// Reverse the natural ordering of U.S. phone numbers
public class ReverseUSPhoneNumberCompare implements Comparator
{
   public int compare ( Object object1, Object object2 ) {
            // Generates a ClassCastException if the wrong
            // type of object is passed.
            UnitedStatesPhoneNumber num1 =
                        (UnitedStatesPhoneNumber) object1;
            int ordering = - num1.compareTo( object2 );
            return ordering;
   }
}
```

Algorithms

- Collections.sort
  - sort(*List* list)
    - uses natural ordering
    - so all objects must be *Comparable* to each other
  - sort(*List* list, *Comparator* c)
    - uses ordering specified by the Comparator, ‘c’

Algorithms (cont)

- Collections.binarySearch
  - binarySearch(*List* list, Object key)

    Search for specified key. List must already be sorted
  - binarySearch(*List* list, Object key, *Comparator* c)

    Search for specified key. List must already be sorted via ‘c’.

Algorithms (cont)

- Collections.min
  - min(*Collection* coll) - use natural ordering of the items
  - min(*Collection* coll, *Comparator* c) - use ordering of ‘c’
- Collections.max - opposite of Collections.min
  - max(*Collection* coll)
  - max(*Collection* coll, *Comparator* c)

Algorithms (cont)

- Collections.subList
  - subList(List list,
    int fromIndex,
    int toIndex)

Get a sub-list of a of *List*

Unmodifiable views

- Use static methods of ‘Collections’ class
  - unmodifiableCollection(…)
  - unmodifiableSet(…)
  - unmodifiableList(…)

  ...
- Wraps a thin veneer around passed object
- Their iterators prevent modification too

Synchronization

- ‘Collections’ class again
  - synchronizedCollection(…)
  - synchronizedSet(…)
  - synchronizedList(…)

  ...
- Again, a thin veneer around object
- Works for any class implementing particular interface

Array Sorting & Searching

- Use class ‘Arrays’ static methods
  - binarySearch - perform a binary search on an already sorted array
  - equals - comparing every item to a single object
  - fill - fill every entry in the array with the specified value
  - sort - tuned quicksort
  - toList - convert array to a fixed-size *List* (good when you need a *List* instead of an array)
- Implemented for:
  - long, int, short, char, byte, double, float, Object
- Object versions
  - sort(Object[] a)
  - sort(Object[] a, *Comparator* c)

Runtime Exceptions

UnsupportedOperationException

- Thrown if object doesn't support operation

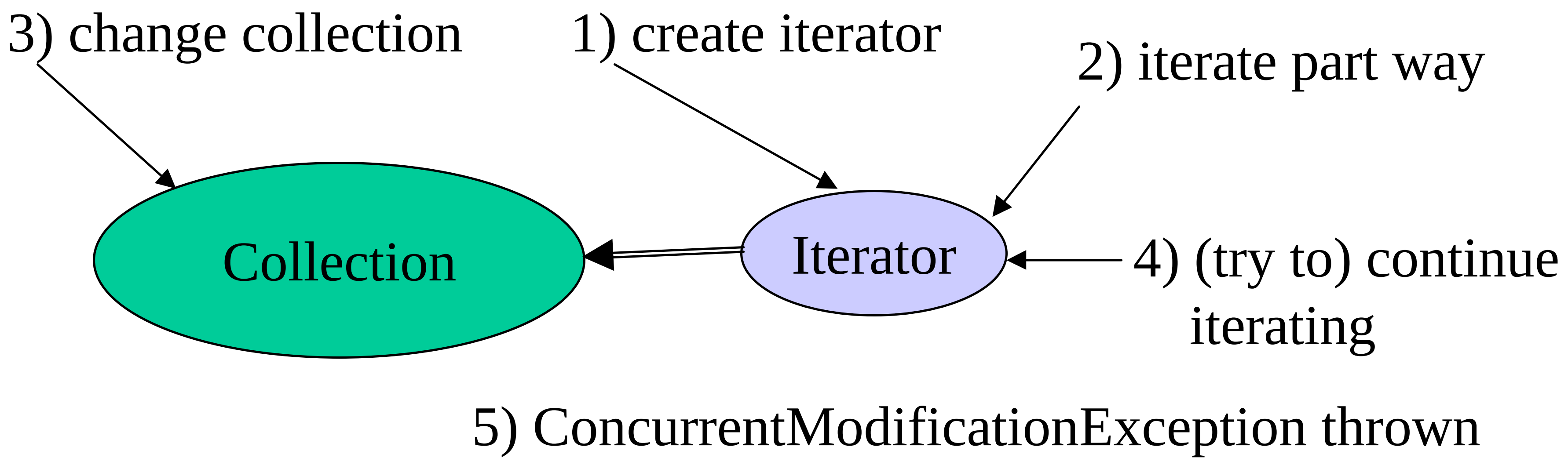
  For example

  - Attempting add/remove on an immutable *Collection*
  - Try to modify a collection via unmodifiable
  - Try to remove from a *Collection* which doesn't support deletion
  - etc.

Runtime Exceptions (cont)

ConcurrentModificationException

- thrown when collection has been modified during iteration
- Default implementations are not thread-safe (called "fail-fast")



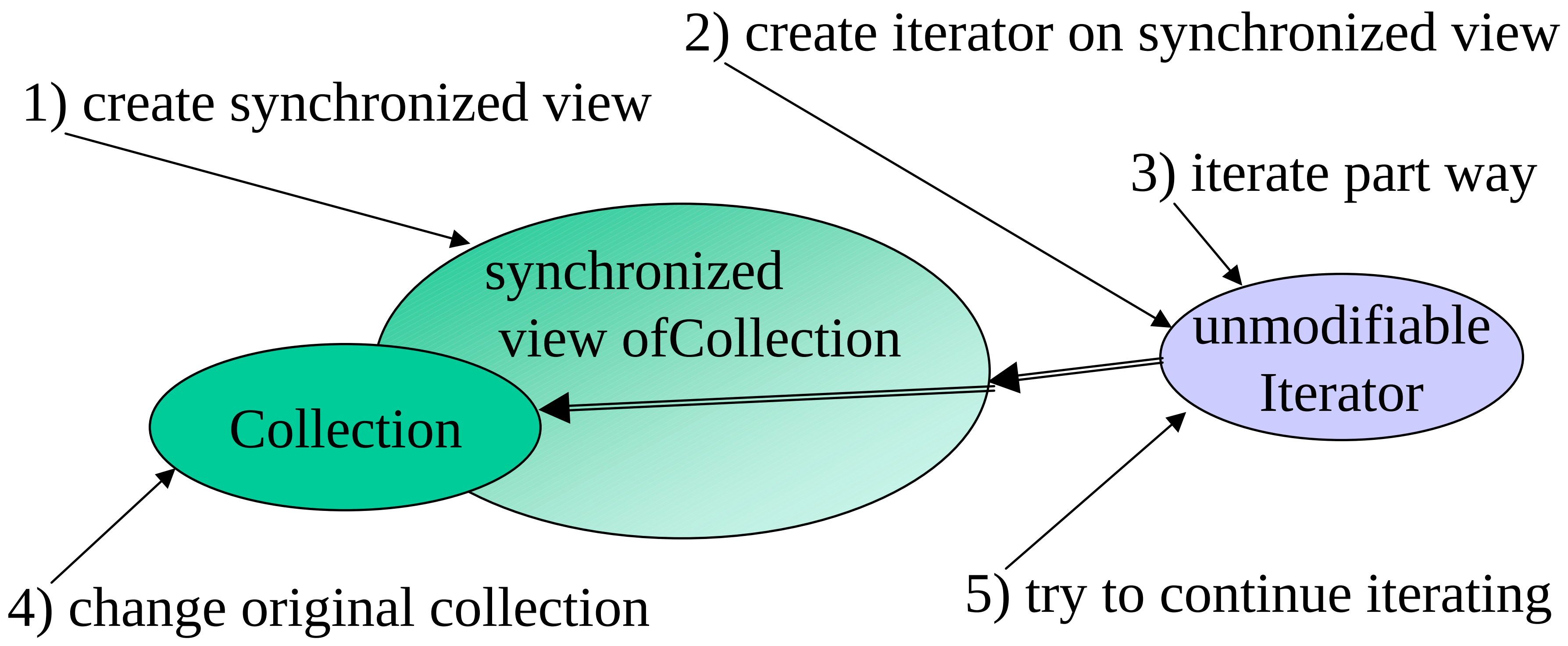
Runtime Exceptions (cont)

ConcurrentModificationException (cont)

- Faster when
  - synchronization not needed
  - performed at a higher level
- Use ‘Collections.synchronizedXXX’ to get synchronization
  - simple safety
  - doesn’t make iterators safe (still sensitive to changes)

Runtime Exceptions (cont)

‘Collections’ static methods don’t make iterators safe


2) create iterator on synchronized view
1) create synchronized view
3) iterate part way
synchronized view ofCollection
unmodifiable Iterator
Collection
4) change original collection
5) try to continue iterating

6) ConcurrentModificationException thrown

Example: Switching Implementations

- ArrayList **VS.** LinkedList
  - Create an immutable *List* with N copies of the same Integer object
  - Time operations upon an ArrayList and a LinkedList
  - Best choice depends on usage

	LinkedList	**ArrayList**
Fill	1100 ms	170 ms
iterate	170 ms	160 ms
'for' loop	57890 ms	60 ms
Add 10,000 to front	720 ms	6420 ms
Add 10,000 to end	1980 ms	170 ms
Remove 10,000 from front	2640 ms	37240 ms
Remove 10,000 from end	<1 ms	<1 ms

Issue: JDK Classes and *Comparable*

- JDK 1.1.x classes don’t implement the *Comparable* interface
- To work around this for JDK classes (like String, Date, etc)
  - Subclass the given class and implement *Comparable*
    - centralizes ordering
    - won’t work for existing code which creates instances of the original class
  - Implement a *Comparator*
    - works with existing class
    - must use same comparator when sorting & searching
- Not an issue for classes we implement
  - We can simply implemente *Comparable*

Summary

- API is general and extensible
- Has good default implementations
- Allows for easy customization of behavior
- Allows for new implementations
- Includes assistance in writing new implementations
- Available for JDK 1.1.x now
- Standard part of JDK 1.2

For more Information

- **JDK 1.1.x implementation**
  - http://java.sun.com/beans/infobus index.html#COLLECTIONS
  - classes are in com.sun.java.util.collections
  - documentation is sparse

- **JDK 1.2 implementation - part of JDK 1.2**
  - http://java.sun.com:80/products/jdk/1.2/index.html
  - classes are in java.util and java.lang

For more Information (cont)

- **Better documentation**
  - http://java.sun.com:80/products/jdk/1.2/docs/guide/collections/index.html
  - JDK 1.1.x implementation is the same except classes are in com.sun.java.util.collections