Introduction to Computation and Problem Solving

Class 32: The Java Collections Framework

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

Goals

- To introduce you to the data structure classes that come with the JDK;
- To talk about how you design a *library* of related classes
- To review which data structures are the best tools for a variety of algorithmic tasks

2

History

In the original version of the Java Development Kit, JDK 1.0, developers were provided very few data structures. These were:

- `Vector`
- `Stack`: which extended `Vector`
- `Hashtable`: very similar to our implementation of `HashMap`
- `Dictionary`: an abstract class that defined the interface and some functionality for classes that map keys to values. `Dictionary` served as the base class for `Hashtable`.
- `Enumeration`: was a simple version of our `Iterator` that allowed you to iterate over instances of `Hashtable` and `Vector`.

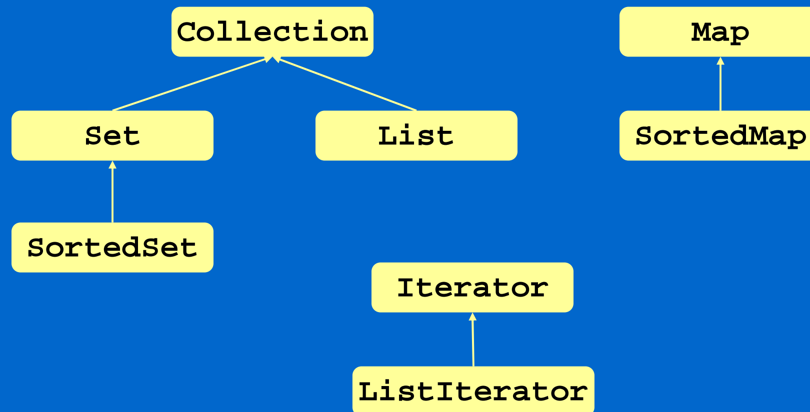
3

The Java Collections Framework

- The Java designers realized that this set of data structures was inadequate.
- They prototyped a new set of data structures as a separate toolkit late in JDK 1.1, and then made it a formal part of the JDK in version 1.2.
- This later, fuller, better designed set of classes is called the Java Collections Framework.
- There is a good tutorial on its use at http://java.sun.com/docs/books/tutorial/collections/index.html.

4

Collections Interfaces, 1



5

Collections Interfaces, 2

- `Collection`: is the most basic interface; it has the functionality of an unordered list, aka a *multi-set*, a set that doesn't not check for duplicates.
- `Set`: adds set semantics; that is, it won't allow duplicate members.
- `List`: adds list semantics; that is, a sense of order and position
- `SortedSet`: adds order to set semantics; our binary search tree that just consisted of keys without values could be implemented as a `SortedSet`

6

Collections Interfaces, 3

- `Map`: the basic interface for data structures that map keys to values; it uses an inner class called an `Entry`
- `SortedMap`: a map of ordered keys with values; our binary search tree is a good example
- `Iterator`: our Iterator except that it is *fail fast*; it throws a `ConcurrentModificationException` if you use an instance after the underlying `Collection` has been modified.
- `ListIterator`: a bidirectional iterator.

Collection Implementations

- The Java designers worked out the architecture of interfaces independently of any data structures. In a second stage they created a number of concrete implementations of the interfaces based on slightly more sophisticated versions of the data structures that we have been studying:
  - *Resizable Array*: similar to the technique used for our Stack implementation.
  - *Linked List*: uses a doubly, not singly, linked list.
  - *Hash Table*: very similar to our implementation except that it will grow the number of slots once the load factor passes a value that can be set in the constructor.
  - *Balanced Tree*: similar to our binary search tree implementation but based on the more sophisticated Red-Black tree that rebalances the tree after some operations.

Collection Implementations, 2

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Inter-faces	*List*		`Array-List`		`Linked-List`
	Set	`HashSet`			
	Sorted Set			`TreeSet`	
	Map	`HashMap`			
	Sorted Map			`TreeMap`	

9

Collection Implementations, 3

- Notice the gaps. There is no list based on a hash table. Why? What about a set implementation based on a linked list?
- The goal of the Java designers was to create a small set of implementations to get users started, not an exhaustive set.
- They expect that developers will extend the set of collection classes just as they expect that developers will add more stream classes.
- The most important part of the collection design was the architecture of interfaces. That's why they called it a *framework.*

10

Implementation Choices

- Be careful of implementation choices. Not all the implementations are going to be efficient at all their operations.
- For instance, an `ArrayList` will allow you to remove its first element by calling remove(0). We already know this will be particularly inefficient.
- You can also iterate over a `LinkedList`, l, using the following code, but once again it will be very inefficient. Why?

```
for ( int i = 0; i < l.size(); i++ )  {
   Object o = l.get( i );
   // do what you need with o
}
```

11

`Collection` Interface

```
public interface Collection<E> {
   // Query Operations
   int size();
   boolean isEmpty();
   boolean contains(Object o);
   boolean containsAll(Collection c);
   Iterator<E> iterator();
   Object[] toArray();
   <T> T toArray(T t[]);
   boolean equals(Object o);
   int hashCode();
```

12

`Collection` Interface, 2

```
  // Modification Operations
  boolean add(E e);
  boolean remove(E e);
  boolean addAll(Collection<? extends E> c);
  boolean removeAll(Collection c);
  boolean retainAll(Collection c);
  void clear();
}
```

13

Bulk Operations

- Note the *bulk* operations. All the interfaces derived from `Collection` support them:
  - `boolean containsAll( Collection c )`
  - `boolean addAll( Collection c )`
  - `boolean removeAll( Collection c )`
  - `boolean retainAll( Collection c )`
- `containsAll()` returns true if the target of the method contains all the elements in the argument collection. The other three methods return true if the operation changes the target collection.

14

Bulk Constructors

All `Collection` (`Map`) implementations support at least two constructors.

- a default constructor that creates an empty `Collection` (`Map`) of the appropriate type, and
- a constructor that takes a `Collection` (`Map`) argument that creates a `Collection` (`Map`) of the appropriate type that contains references to all the objects in the `Collection` (`Map`) supplied as the argument.

15

Array Operations

- All `Collection` implementations also possess a method with the signature:

```
Object [] toArray()
```

  which returns the contents of the collection in an appropriately sized array.

- A variant

```
<T> T[] toArray(T t[]);
```

  returns all the elements of the collection in array `t[]` itself or an appropriately sized array of the same type. All the elements in the Collection must be of type T.

16

Array Examples

- `Collection c = new HashSet();`
  `// put Strings in c`
  `String[] s = c.toArray(new String[0]);`
  returns all elements of `c` in an array of `String`s.
- Note this is evil:
  `c.add( new Integer(42) );`
  `String[] s = c.toArray(new String[0]);`
  You will get an `ArrayStoreException`.
- `Collection c = new HashSet();`
  `// if c contains only Strings`
  `String[] s = (String[]) c.toArray( );`

17

`List` Interface

```
public interface List<E> extends Collection<E> {
  // adds the following to Collection
  boolean addAll(int index,
                 Collection<? extends E> c);
  E get(int index);
  E set(int index, E element);
  void add(int index, E element);
  E remove(int index);
  int indexOf(Object o);
  int lastIndexOf(Object o);
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
  List<E> subList(int fromIndex, int toIndex);
}
```

18

Views

- Several of the interfaces also support *view* operations. In a view operation, a collection returns another collection that provides a specialized and usually partial view of its contents. A good example is

```
List subList( int from, int to )
```

- This method returns a second list that starts with the `from`'*th* element of the parent list and ends just before the `to`'*th* element. But the returned sublist is not a copy. It remains part of the original list.
- As an example, an elegant way to delete the 4th through 10th elements of a list is the following:

```
myList.subList( 3, 10 ).clear();
```

19

`Set` Interface

```
public interface Set extends Collection {
  // has the same methods as Collection
  // but with stricter semantics,
  // no duplicate elements
}
```

20

SortedSet Interface

```
public interface SortedSet<E> extends Set {
  // adds the following to Set
  Comparator comparator();
  SortedSet<E> subSet(E fromElement,
                      E toElement);
  SortedSet<E> headSet(E toElement);
  SortedSet<E> tailSet(E fromElement);
  E first();
  E last();
}
```

21

Iterator Interface

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

22

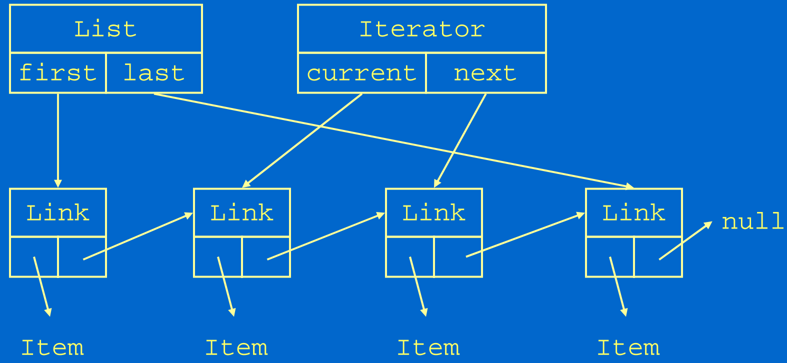
`ListIterator` Interface

```
public interface ListIterator<E>
  extends Iterator<E> {
  // adds the following to Iterator
  boolean hasPrevious();
  E previous();
  int nextIndex();
  int previousIndex();
  void set(E o);
  void add(E o);
}
```


Concurrent Modification

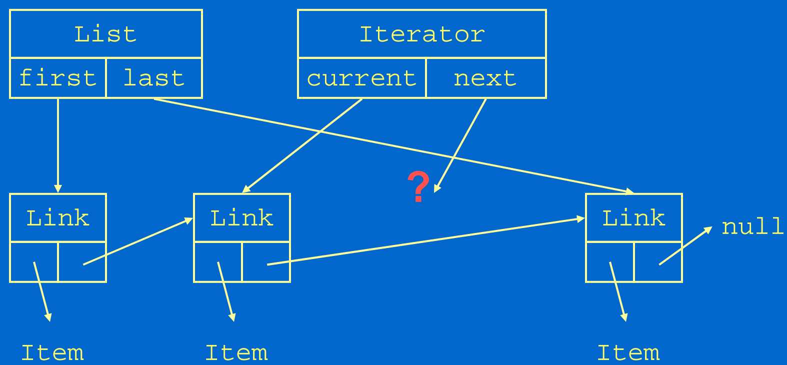
- Iterators have to keep references to the underlying data structure to maintain their sense of current position.
- Thought experiment:
  - create a LinkedList, add elements, get an iterator, advance it
  - now using a method in the list not the iterator, delete the iterator's next item
  - now call next() on the iterator
  - what goes wrong?
- Any time you create an iterator, modify the underlying collection, and then use the iterator, you will get a `ConcurrentModificationException`.
- How do you think they implement the check?

Concurrent Modification, Before



25

Concurrent Modification, After



26

Map Interface

```
public interface Map<K, V> {
  // Query Operations
  int size();
  boolean isEmpty();
  boolean containsKey(K key);
  boolean containsValue(V value);
  V get(K key);
```

27

Map Interface, 2

```
  // Modification Operations
  V put(K key, V value);
  V remove(K key);
  void putAll(Map<? extends K, ? extends V> t);
  void clear();
  // Views
  public Set<K> keySet();
  public Collection<V> values();
  public Set<Map.Entry<K, V>> entrySet();
  boolean equals(Object o);
  int hashCode();
```

28

Nested `Map.Entry` Interface

```
public interface Entry<K, V> {
    K getKey();
    V getValue();
    V setValue(V value);
    boolean equals(Object o);
    int hashCode();
  }
}
```

29

`Map` Views

- `Map`s also provide views. In fact they are crucial because a `Map` does not provide an `iterator()` method.
- They are two separate idioms for iterating over a Map, depending on whether you want to iterate over the keys or the values:

```
Map<K, V> m = ...;
// iterates over keys
for (Iterator<K> i=m.keySet().iterator();
i.hasNext();)
{...}
// iterates over values
for (Iterator<V> i=m.values().iterator();
i.hasNext();)
{...}
```

- In the second example, `values()` returns a Collection, not a Set, because the same value can occur multiple times in a map while a key must be unique.

30

`SortedMap` Interface

```
public interface SortedMap<K, V>
      extends Map<K, V> {
  // adds the following to Map
  Comparator 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();
}
```


Optional Operations

- Not all implementations implement all the operations in an interface. The documentation makes it clear whether a particular implementation implements a given method. If it doesn't, it throws an `UnsupportedOperationException` (unchecked).
- At first, this just seems perverse, but the goal is to provide more flexibility. Imagine a program where you want to maintain a master index in which you want a normal user to be able to look up entries but only a privileged user to add entries to the index. The index could be implemented as a `SortedMap`, but then how could you keep an arbitrary user from adding and deleting entries? The answer is to subclass `SortedMap` and override each method you wanted to forbid with a simple method that throws an `UnsupportedOperationException`.

Algorithms

- The `Collections` class contains implementations of a number of useful algorithms implemented as `static` methods that take a `List` or sometimes a more general `Collection` as the target of the algorithm.
- Algorithms include `sort()`, `reverse()`, `min()`, `max()`, `fill(Object o)`, etc.

```
static void sort( List l, Comparator c )
static void fill( List l, Object o )
```

- There is a similar class `Arrays`, which implements most of the same algorithms for arrays.

33

Read-Only Collections

`Collections` can also produce unmodifiable (read only) views of all types of collections that will throw an `UnsupportedOperationException` on any write operation:

```
public static Collection
  unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
...
```

34