Lists in Java

Part of the Collections Framework

Kinds of Collections

- Collection--a group of objects, called *elements*
  - Set--An unordered collection with no duplicates
    - SortedSet--An ordered collection with no duplicates
  - List--an ordered collection, duplicates are allowed
- Map--a collection that maps *keys* to *values*
  - SortedMap--a collection ordered by the keys
- Note that there are *two* distinct hierarchies

Using Collections

- import java.util.*
  or import java.util.Collection;

- There is a sister class, java.util.Collection**s**; that provides a number of algorithms for use with collections: sort, binarySearch, copy, shuffle, reverse, max, min, etc.

Collections are interfaces

- Collection is actually an interface
- Each kind of Collection has one or more implementations
- You can create new kinds of Collections
- When you implement an interface, you promise to supply the required methods
- Some Collection methods are optional
  - How can an interface declare an *optional* method?

Creating a Collection

- All Collection implementations should have two constructors:
  - A no-argument constructor to create an empty collection
  - A constructor with another Collection as argument
- All the Sun-supplied implementations obey this rule, but--
- If you implement your own Collection type, this rule cannot be enforced, because an Interface cannot specify constructors

Collection: Basic operations

```
int size( );
boolean isEmpty( );
boolean contains(Object element);
boolean add(Object element);       // Optional
boolean remove(Object element); // Optional
Iterator iterator( );
```

Collection: Iterator

```
public interface Iterator {
    boolean hasNext( );
        // true if there is another element
    Object next( );
        // returns the next element (advances the iterator)
     void remove( );    // Optional
       // removes the element returned by next
}
```

Using an Iterator

- ```
  static void printAll (Collection coll) {
      Iterator iter = coll.iterator( );
      while (iter.hasNext( )) {
         System.out.println(iter.next( ) );
      }
  }
  ```
- Note that this code is polymorphic--it will work for *any* collection
  ```

Collection: Bulk operations

```
boolean containsAll(Collection c);
boolean addAll(Collection c);       // Optional
boolean removeAll(Collection c);    // Optional
boolean retainAll(Collection c);    // Optional
void clear( );                      // Optional
```

- addAll, removeAll, retainAll return true if the object receiving the message was modified

Mixing Collection types

- Note that most methods, such as

  ```
  boolean containsAll(Collection c);
  ```

  are defined for *any* type of `Collection`, and take *any* type of `Collection` as an argument
- This makes it very easy to work with different types of Collections

singleton

- Collections.singleton(e) returns an immutable set containing only the element e
- c.removeAll(Collections.singleton(e)); will remove all occurrences of e from the Collection c

Collection: Array operations

- Object[ ] toArray( );
  - creates a new array of Objects
- Object[ ] toArray(Object a[ ]);
  - Allows the caller to provide the array
- Examples:

```
Object[ ] a = c.toArray( );
String[ ] a;
a = (String[ ]) c.toArray(new String[0]);
```

The List interface

- A List is ordered and may have duplicates
- Operations are exactly those for Collections

```
int size( );
boolean isEmpty( );
boolean contains(Object e);
boolean add(Object e);
boolean remove(Object e);
Iterator iterator( );
```

```
boolean containsAll(Collection c);
boolean addAll(Collection c);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
void clear( );

Object[ ] toArray( );
Object[ ] toArray(Object a[ ]);
```

List implementations

- List is an interface; you can't say new List ( )
- There are two implementations:
  - LinkedList gives faster insertions and deletions
  - ArrayList gives faster random access
- It's poor style to expose the implementation, so:
- Good: List list = new LinkedList ( );
  Bad: LinkedList list = new LinkedList ( );

Inherited List methods

- list.remove(e) removes the *first* e
- add and addAll add to the *end* of the list
- To append one list to another:

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

- To append two lists into a new list:

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

- Again, it's good style to hide the implementation

List: Positional access

```
Object get(int index);  // Required --
                        // the rest are optional
Object set(int index, Object element);
void add(int index, Object element);
Object remove(int index);
abstract boolean addAll(int index, Collection c);
```

- These operations are more efficient with the ArrayList implementation

List: Searching

```
int indexOf(Object o);
int lastIndexOf(Object o);
```

- equals and hashCode work even if implementations are different

Interface List: Iteration

- Iterators specific to Lists:

  ```
  ListIterator listIterator( );
  ListIterator listIterator(int index);
  ```

  - starts at the position indicated (0 is first element)
- Inherited methods:

  ```
  boolean hasNext( );
  Object next( );
  void remove( );
  ```

- Additional methods:

  ```
  boolean hasPrevious()
  Object previous()
  ```

List: Iterating backwards

```
boolean hasPrevious( );
Object previous( );
int nextIndex( );
int previousIndex( );
```

- Think of the iterator as "between" elements
- Hence, next followed by previous gives you the same element each time

List: More operations

- void add(Object o);
  - Inserts an object at the cursor position
- Object set(Object o); // Optional
  - Replace the current element; return the old one
- Object remove(int index); // Optional
  - Remove and return the element at that position

List: Range-view

- `List subList(int from, int to);` allows you to manipulate part of a list
- A sublist may be used just like any other list

Thank you