

Object - Oriented Programming & Design

Part VII: Java Collection Classes

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

Collection Classes

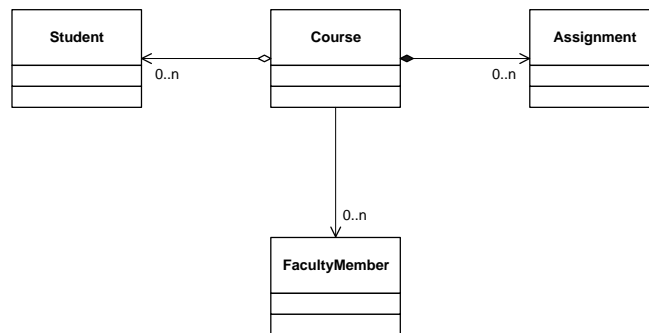
- A **collection** is a grouping of objects, usually of the same class, or with a common base class.
- Used to represent the *many* side of *one-to-many* associations, where the association must be navigable from the *one* side to the *many* side:
 - aggregations
 - compositions
- Excellent support in most OO languages:
 - The `java.util` package
 - The `java.util.concurrent` package
 - STL for C++ (Standard Template Library)
 - Collection classes in Smalltalk

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 2

Collection Example



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 3

Java Collections

- Pre JDK 1.2
 - Vector (ported to 1.2 Collection framework)
 - Hashtable (ported to 1.2 Collection framework)
- JDK 1.2 through JDK 1.4
 - JDK 1.2 Collections are *typeless*; they hold **objects**, and you usually must *downcast* the results of operations on them.
 - More uniform class structure
 - Wider variety of collections
 - Introduces Iterator design pattern
- JDK 1.5 +
 - Generics! It's about time :-)

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 4

Java Collections (JDK 1.2 to 1.4)



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 5

Collection Considerations

- There are many different types of collections. Choosing the best one for the job often requires careful consideration.
- Not all OO languages support all types of collections.
 - E.g., the C++ Standard Template Library (STL) does not support `HashTable`, but it does support `TreeMap`.
 - Prior to version 1.2, Java did not directly support sorted collections.
- Consider the types of operations you will need to make on the collection, because each type of collection performs some operations better (and/or more efficiently) than others:
 - Adding an element
 - Removing an element
 - Searching for an element
 - Sorting
 - Iterating

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 6

The Java Collections Framework

The Java Collections Framework (JDK version 1.2+) provides a robust set of collections, plus utilities to help with such things as thread safety, sorting and reversing a list.

There are interfaces for each kind of collection, abstract classes that mostly implement the interfaces, and concrete classes for all the variety; plus...

Two utility classes (with static “convenience” methods, for functions such as sorting):

- Collections
- Arrays

Two interfaces for ordered collections (TreeSet & TreeMap):

- Comparable
- Comparator

One other important interface:

- Iterator

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 7

The Collection Interface (JDK 1.2)

```
public interface Collection
{
    public boolean  add( Object e );
    public void    clear();
    public boolean  contains( Object e );
    public boolean  isEmpty();
    public Iterator iterator();
    public boolean  remove( Object e );
    public int     size();
    public Object[] toArray();
    ...
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 8

Java Generics (JDK 1.5 +)

- Generics provide increased type safety and expressiveness.

```
// Bad example - avoid this usage:
private final Collection foos = ...; // collection of Foos?
foos.add( new Bar() ); // Wrong, but compiles and runs
for( Iterator i = foos.iterator(); i.hasNext(); ) {
    Foo foo = (Foo) i.next(); // throws ClassCastException

// A better way:
private final Collection< Foo > foos = new ArrayList< Foo >();
foos.add( new Bar() ); // Will not compile unless Bar is-a Foo
for( Foo foo : foos ) { // No downcast required.
// Easy syntax, type safety, all good :-)
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 9

The Collection Interface (JDK 1.5)

```
public interface Collection< E >
    extends Iterable< E >
{
    public boolean    add( E o );
    public void      clear();
    public boolean    contains( Object o );
    public boolean    isEmpty();
    public Iterator< E > iterator();
    public boolean    remove( Object o );
    public int        size();
    public Object[]   toArray();
    ... ..
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 10

Arrays vs. Lists

- Java and C++ both have arrays at the language syntax level.
 - A type of *indexed collection*.
 - Size fixed at creation time.
 - The programmer must handle out-of-bounds situations.
 - C++ has tricky array pitfalls - better not to use them.

```
String[] ooLanguages = new String[] { "C++", "Java",  
    "Smalltalk", "Jade", "C#", "Lisp" };
```

- List is a *collection class*.
 - *Dynamically sized* array.
 - No out-of-bounds issues.
 - Supports the *Iterator* design pattern.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 11

java.util.ArrayList (JDK 1.2)

```
public class ArrayList implements List  
{  
    public          ArrayList();  
    public void     add( Object o );  
    public Object   remove( int index );  
    public Object   get( int index );  
    public int      size();  
    public Object[] toArray();  
    ...  
}  
...  
ArrayList al = new ArrayList();  
al.add( new MyClass() );  
MyClass mc = (MyClass) al.get( 0 ); // Downcast required !!
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 12

java.util.ArrayList (JDK 1.5)

```
public class ArrayList< E > extends AbstractList< E >
    implements List< E >, RandomAccess, Cloneable, Serializable {
    public          ArrayList();
    public          ArrayList( Collection< ? extends E > c );
    public boolean  add( E o );
    public E        remove( int index );
    public E        get( int index );
    public int      size();
    ... ..
}
// < ? Extends E > means: an unknown type that is a subtype of E,
// possibly E itself. This is an example of a bounded wildcard.

ArrayList< MyClass > al = new ArrayList< MyClass >();
al.add( new MyClass() );
MyClass mc = al.get( 0 ); // NO Downcast required !!
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 13

The Iterator Design Pattern

Intent: Separate the iteration over a collection class from the class being iterated upon, so that *multiple iterations can progress concurrently*, and so that *different iteration strategies might be used interchangeably*.

Iterators are objects, tightly bound to a single collection, that allow you to move through the collection.

- Clients of the collection perform many operations on the collection through iterators, rather than on the collection directly.
 - Accessing elements sequentially.
 - Searching for an element may return an *iterator*, rather than the element itself, especially if more than one element satisfies the search criteria. This is the case for example, with database queries using Java's JDBC (refer to the class `java.sql.ResultSet`).
- Iterators *can* enable more than one client to safely operate on a collection at once.

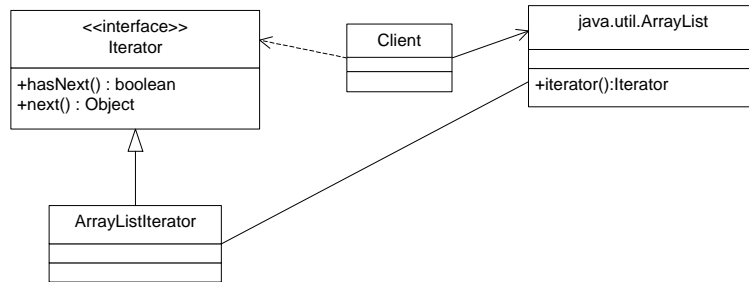
Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 14

Iterator (cont.)

- The Client class remains blissfully ignorant of the actual class that implements the Iterator interface.



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 15

java.util.Iterator

- An interface for an object that moves sequentially through a collection
- Each concrete collection class has its own implementation of Iterator
- The only way to get an Iterator instance is by calling the `iterator()` method on a Collection, or `keySet().iterator()` on a Map.
- If you add to or delete from a collection while iterating through it, your iterator might no longer be valid (be careful!). Use `it.remove()`;

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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 16

java.util.Iterator (cont.)

```
public void doFoo( List< Foo > listOfFoos )
{
    Iterator< Foo > fit = listOfFoos.iterator();
    while( fit.hasNext() )
    {
        Foo foo = fit.next(); // No downcast
        if( bogusFoo( foo ) )
        {
            fit.remove(); // Safe removal
        }
    }
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 17

Associative Arrays

- Very important collection class, also called *dictionaries* or *maps*
- Contains *key / value* pairs
- Like a normal array, but indexed by *key*, rather than by sequential integer.
- Supported by several Java classes
 - java.util.Hashtable (JDK 1.0+)
 - java.util.HashMap / TreeMap (JDK 1.2+)
 - java.util.concurrent.ConcurrentHashMap (JDK 1.5+)

```
HashMap< String, PhoneNum > phoneBook =
    new HashMap< String, PhoneNum >();
phoneBook.put( "Dupp, Jack", new PhoneNum( "720-555-9354" ));
phoneBook.put( "Lucks, Dee", new PhoneNum( "303-555-1764" ));
PhoneNum number = phoneBook.get( "Lucks, Dee" );
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 18

Associative Arrays (cont.)

```
import java.util.HashMap;
class MapTest {
    public static void main( String[] args ) {
        HashMap map = new HashMap(); // JDK 1.2 usage
        map.put( "a", "b" );
        map.put( "a", "c" );
        Object o = map.get( "a" );
        String s = (String) o; // Downcast required. Pre-1.5 usage
        System.out.println( "key = a; value = " + s );
    }
}

// outputs:
key = a; value = c
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 19

Hashtable – data structure

- Hash table data structures have been around for decades, mainly used as a fast associative array, and in many places other than OO containers.
- The hash table has a fixed number of “buckets.”
- A *hash function* is defined on the keys.
 - The hash function assigns the value of each key/value pair to a bucket.
 - Based on the key, the hash function computes a integer value between 0 and the number of buckets minus 1.
 - » For an *integer* type key, the hash function could return the key modulo the number of buckets.
 - » For a *string* type key, the hash function could add the ASCII values of all the characters, then return the sum modulo the number of buckets.
 - Hash values are not usually unique across the range of possible keys. When two keys produce the same hash values, they are said to *collide*.
 - Values with colliding keys are put into a linked list for the bucket.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 20

Hashtable and HashMap data structures

- Pros:
 - Easy to use dictionary container class, with a simple API: put() & get()
 - Part of the Java 1.2 Collections Framework.
 - Performs well if you have a good hash function.
 - Can store one-to-one, one-to-many, and many-to-many associations.
- Cons:
 - Has relatively poor performance if the hash function produces many keys collisions:
 - » Removing an element.
 - » Searching for an element.
 - Not sortable, so Iterators created from the map's keySet(), entrySet(), or values() return entries in random order (compare to TreeMap).

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 21

java.util.HashMap (JDK 1.2)

```
public class HashMap extends AbstractMap
{
    public void          clear();
    public boolean      containsKey( Object key );
    public boolean      containsVlaue( Object value );
    public Set          entrySet();
    public Object       get( Object key );
    public boolean      isEmpty();
    public Set          keySet();
    public void         put( Object key, Object value );
    public Object       remove( Object key );
    public int          size();
    public Collection   values();
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 22

java.util.HashMap (cont.)

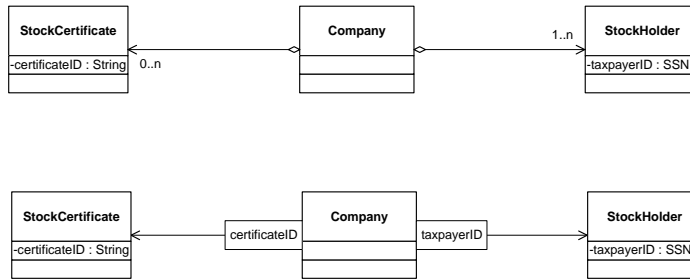
- HashMap doesn't implement the Collection interface, so to get an Iterator, you must call `keySet()`, `entrySet()`, or `values()` to get a Collection, and then iterate over that Collection.
- Not *thread-safe*, for increased performance. But, the `Collections` utility class provides a `synchronizedMap(Map m)` method, wrapping the given Map, providing *optional* thread safety. Or use `ConcurrentHashMap`.
- Hash functions
 - **Object** provides a default hash function based on the instance's location in memory, and so puts and gets from the map based on object identity. This is what you want in most instances.
 - **String** has a hash function based on the characters in the string, and so puts and gets from the map based on value equality. This is also what you usually want.

Map Considerations

- In Java, keys and values must be Objects, not primitive types.
 - For Objects that are used as keys in hash maps, *a test for equality is needed*. In Java, it is common to **override** `Object`'s `equals()` method.
 - For `TreeMaps` (and `TreeSets`), a sort order function is required, accomplished with **Comparator** or **Comparable**.
- Get used to using Maps !!!
 - The availability of associative arrays can transform the way you write programs.

Qualified Associations

- Qualified associations usually imply associative arrays



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 25

Tree Map

- Maintains elements in a type of *balanced tree*.
- Pros and cons similar to hash table, plus...
 - Tree maps keep the elements in sorted order, which can be a very useful property. Sorting requires that all elements implement the Comparable interface, or that the TreeMap be constructed with an appropriate Comparator.
- Comparison to hash tables:
 - In the best case, a hash table performs puts and gets somewhat better than a tree map. In the worst case, a hash table will perform much worse than a tree map.
 - Because of this, the C++ Standard Template Library (STL) chose to base its associative arrays on trees rather than hash tables.
 - Smalltalk implements its Dictionary class with hash tables.
 - Java provides both types (HashMap, and TreeMap).

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 26

Comparable & Comparator (JDK 1.2)

```
// return -1, 0, or 1 ...

interface Comparable
{
    public int compareTo( Object other );
}

interface Comparator
{
    public int compare( Object o1, Object o2 );
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 27

Comparable Example (JDK 1.2)

```
import java.util.*;

public class CollectionTest implements Comparable {
    private int value = 0;

    public int compareTo( Object other ) {
        int otherValue = ((CollectionTest) other).getValue();
        if( this.value == otherValue ) return 0;
        return ( this.value > otherValue ) ? 1 : -1;
    }

    private int getValue() {
        return value;
    }

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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 28

Comparable Example (cont.)

```
private void setValue( int value ) {
    this.value = value;
}
public String toString() {
    return "( " + value + " )";
}
// Note no equals() for value comparison
public static void printMap( Map map ) {
    Set keys = map.keySet();
    Iterator it = keys.iterator();
    while( it.hasNext() ) {
        Object key = it.next();
        Object value = map.get( key );
        System.out.print( "Key = " + key );
        System.out.println( " Value = " + value );
    }
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 29

Comparable Example (cont.)

```
public static void main( String[] args ) {
    CollectionTest[] cts = new CollectionTest[ 5 ];
    HashMap hash = new HashMap();
    TreeMap tree = new TreeMap();
    Random random = new Random( System.currentTimeMillis() );
    for( int i = 0; i < 4; i++ ) {
        CollectionTest ct = new CollectionTest();
        cts[ i ] = ct;
        int randy = Math.abs( random.nextInt() ) % 100;
        ct.setValue( randy );
        hash.put( ct, "" + i );
        tree.put( ct, "" + i );
    }
    List list = Arrays.asList( cts );
    System.out.println( "HASH dump:" );
    printMap( hash );
    System.out.println( "TREE dump:" );
    printMap( tree );
}
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 30

Comparable Example (cont.)

36 is **duplicate** in the HashMap but *not* in the TreeMap

- TreeMap uses `compareTo()` to test for equality.
- HashMap relies on `equals()`, which `CollectionTest` lacks.

HASH dump:

Key = (75) Value = 1

Key = (39) Value = 0

Key = (36) Value = 3

Key = (36) Value = 2

TREE dump:

Key = (36) Value = 3

Key = (39) Value = 0

Key = (75) Value = 1

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 31

Another Collections Example

```
import java.util.*;
public class CollectionsExample {
    public static void main( String args[] ) {
        // Print a list of ints in reverse sorted order.
        List< Integer > listOfInts = new LinkedList< Integer >();
        listOfInts.add( 7 ); // Note the use of "auto-boxing"
        listOfInts.add( 3 );
        listOfInts.add( 11 );

        Comparator< Integer > reverse = Collections.reverseOrder();
        Collections.sort( listOfInts, reverse );

        for( int i : listOfInts ) { // more "auto-boxing"
            System.out.println( i );
        }
    }
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 32

Sets

- A set is a collection of unique objects (there are no duplicates).
- Supported in Java (1.2) with `HashSet` and `TreeSet`.
- Useful for determining membership.

// Set of customers with outstanding charges is kept in memory:

```
HashSet custWithCharges = new HashSet();
```

// Somewhere else, in code that checks for such things...

```
if( custWithCharges.contains( cust.getId() )
{
    return false;
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 33

Other Java Collections

- **LinkedList** – A doubly-linked list.
- **Stack**
- **WeakHashMap** – Entries are automatically removed when their key is no longer in use. A weak reference is not counted as a reference in garbage collection.
- **BitSet** – Vector of bits that grows as needed.
- **Properties** – Set of properties (usually used for *.ini* type files).

Refer to `java.util.concurrent` for such classes as:

- `BlockingQueue` - Blocks or times out when adding to a full queue or removing from an empty queue.
- `ConcurrentMap<K, V>` - Adds *atomic* methods:

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(VII) Java Collection Classes - 34