Object – Oriented Design with UML and Java

Part VII: Java Collection Classes

Copyright © David Leberknight

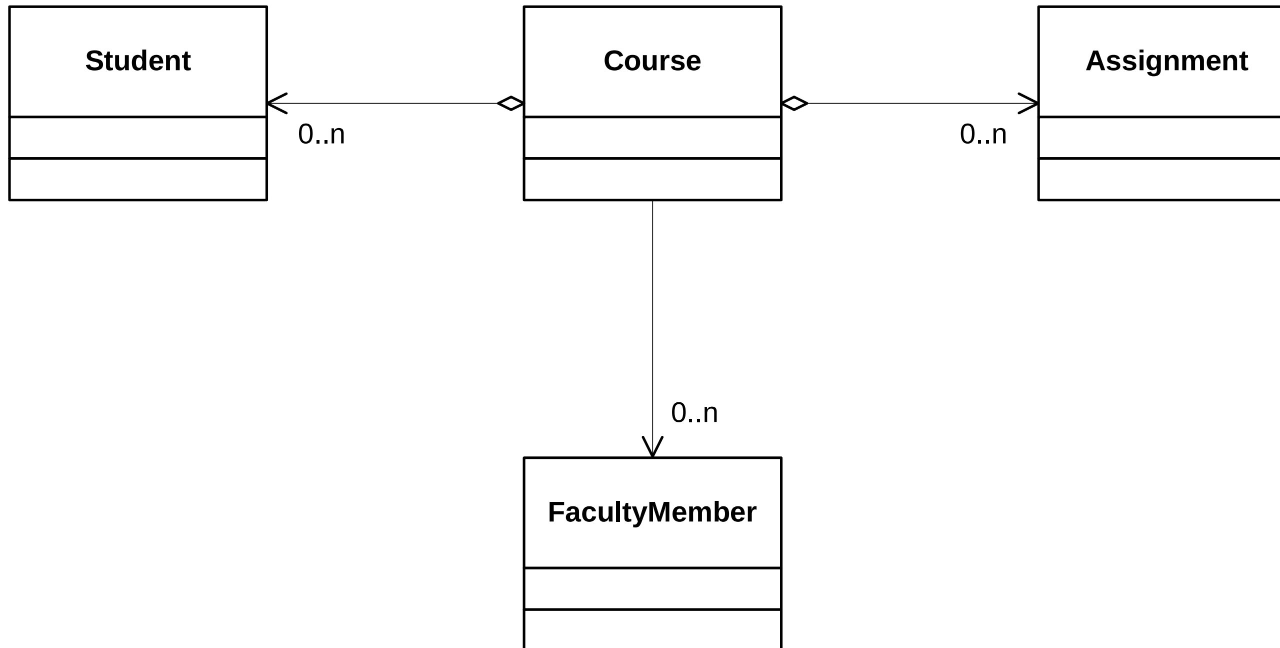
Version 2023

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 Example

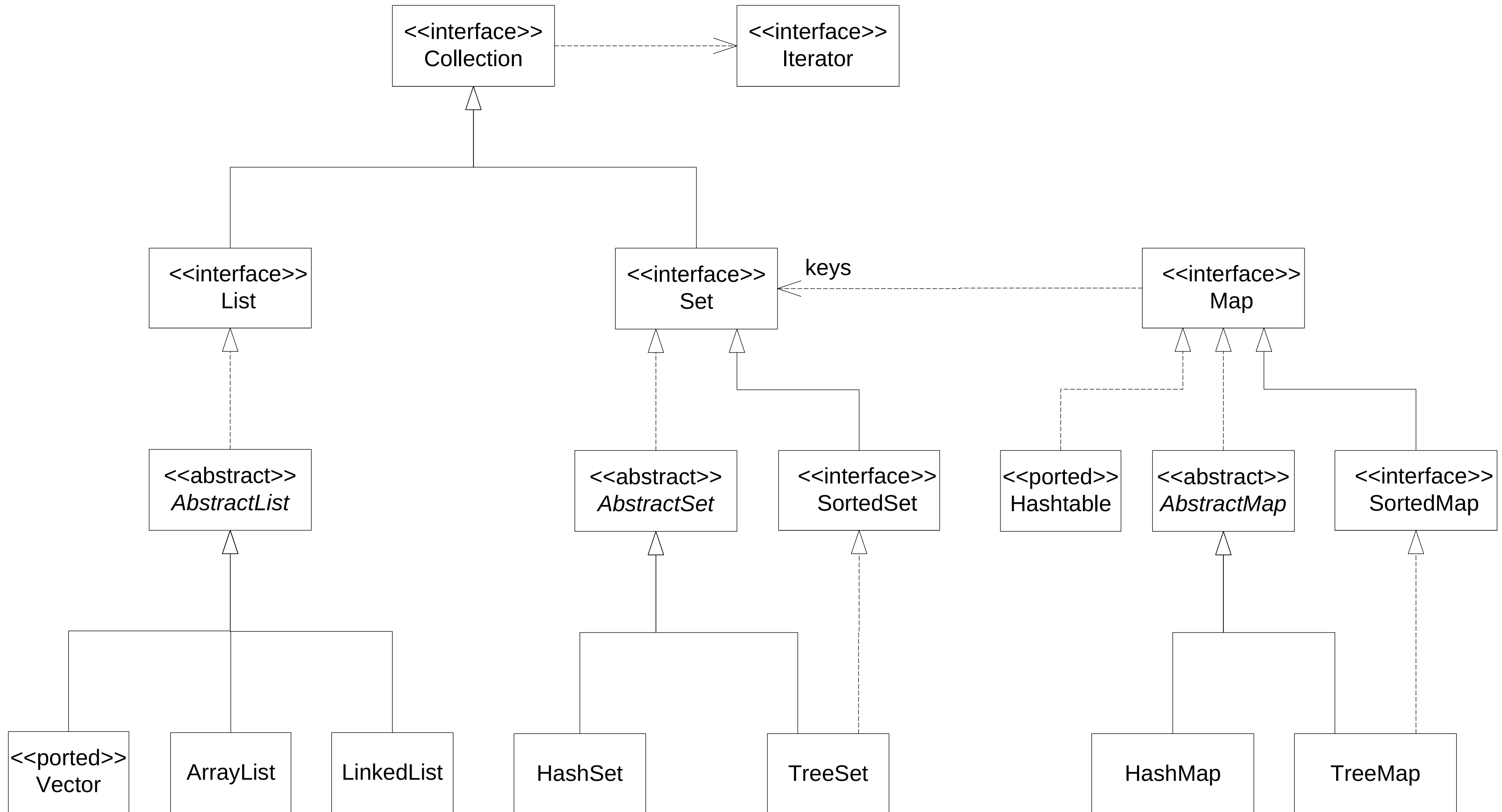
- Each one-to-many association implies a collection attribute.



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 must ***downcast*** the results of operations on them.
  - More uniform class structure
  - Wider variety of collections
  - Introduces ***Iterator*** design pattern
- JDK 1.5 +
  - **Generics**! (analogous to C++ ***templates***)
  - It’s about time :-)

Java Collections (JDK 1.2 to 1.4)



Collection Considerations

- There are many different types of collections.
- Choosing the best one for the job requires careful consideration.
- Not all OO languages support all types of collections.
  - 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 operations you need to make on the collection, because each type performs some operations better (more efficiently) than others:
  - Adding / removing / searching for an element
  - Sorting
  - Iterating

The Java Collections Framework

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

Classes with "convenience" methods for functions such as sorting:

- **`Collections`**
- **`Arrays`**

Iterfaces for *ordered* collections (TreeSet & TreeMap):

- **`Comparable`**
- **`Comparator`**

One other important interface:

- **`Iterator`**

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();
...
}
```

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 :-)
```

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();
  … … …
}
```

Primitive Arrays vs. Collections

- 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”, “C#”, “JADE”, “Python”, “Lisp”};
```

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

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
```

java.util.ArrayList (JDK 1.5)

```
public class ArrayList< E > extends AbstractList< E >
     implements List< E >, . . . 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 >
```

`// < ? 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 !!
```

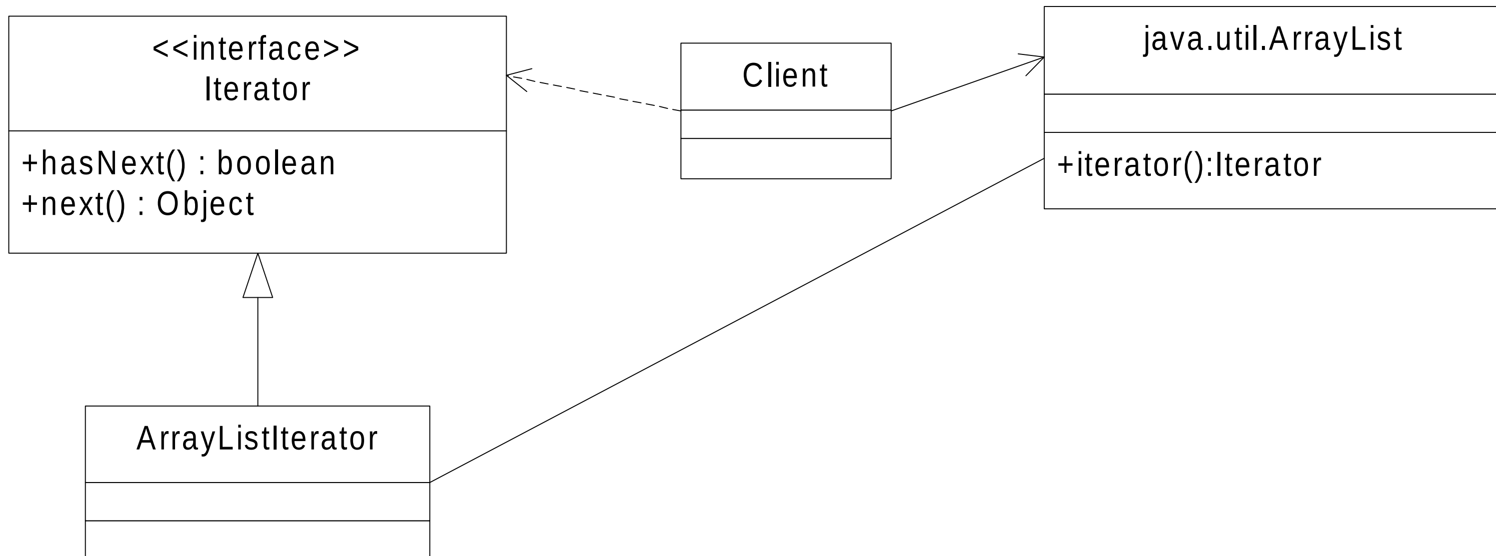
The Iterator Design Pattern

Intent: Isolate the iteration over a collection, so that ***multiple iterations can progress concurrently***, and so that ***different iteration strategies might be used interchangeably***.

- Iterators are tightly bound to a single collection, allowing you to move through the collection sequentially.
  - Searching for an element may return an iterator if more than one element satisfies the search criteria. This is the case for example, with database queries using Java's JDBC (refer to `java.sql.ResultSet`).
- Iterators *can* enable more than one client to *safely* operate on a collection at once. Use `java.util.concurrent.*`

Iterator as good design

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



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();
}
```

java.util.Iterator Example

```
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
    }
  }
}
```

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.
- Support in Java:
  - `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” );
```

Associative Arrays – JDK 1.2 usage

```
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 - Pre-1.5 usage
    System.out.println( “key = a; value = “ + s );
  }
}
// outputs:
key = a; value = c
```

Hash Table – data structure

- Hash table data structures have been around for decades, mainly as a fast associative array, and in places other than OO containers.
- A ***hash function*** is defined on the keys.
  - The hash function assigns the value of each key to one of a fixed number of buckets (computed in constant time – no search).
  - Based on the key, the hash function computes a integer value between 0 and the number of buckets - 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 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.

Hashtable and HashMap data structures

- Pros:
  - Useful dictionary class with a simple API: put() & get()
  - Performs well if you have a good hash function, especially if there are a very large number of keys (compared to searching).
- Cons:
  - Has poor performance if the hash function produces many key collisions (mapping to the same bucket).
  - Not sortable, so Iterators created from the keySet() return entries in random order - Compare to TreeMap.

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();
}
```

java.util.HashMap

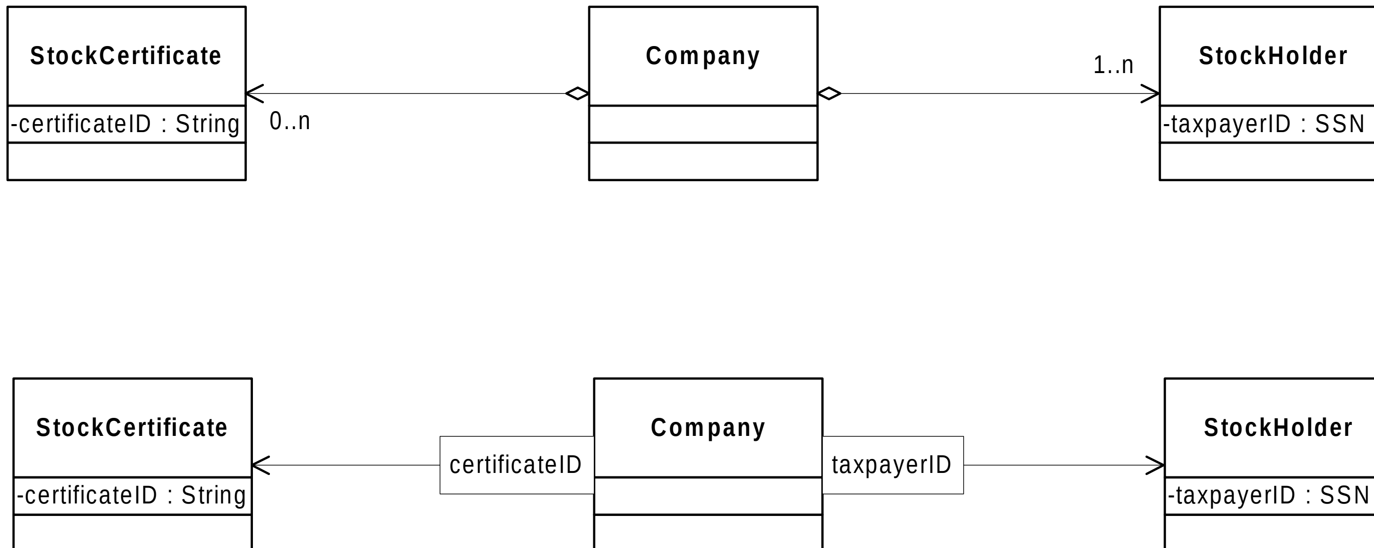
- HashMap doesn’t implement the Collection interface, so to get an Iterator, you must call `keySet()`, `entrySet()`, or `values()` to get a Collection, and then iterate over that Collection.
- Not ***thread-safe***, for increased performance. But the `Collections` utility class provides a `synchronizedMap( Map m )` method, wrapping the given Map, providing *optional* thread safety.
- Or use `java.util.concurrent.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 are 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 are based on value equality. This is also what you usually want.

Map Considerations

- In Java, keys and values must be Objects, not primitive types.
  - Although with Java 5 there is “auto-boxing.”
  - For Objects that are used as keys in hash maps, *a test for equality is needed*. 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



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 useful. Sorting requires that all elements implement the ***Comparable*** interface, or that the TreeMap be constructed with an appropriate ***Comparator***.

- 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 HashMap and TreeMap.

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 );
}
```

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;
  }
```

Comparable Example (cont.)

```
  private void setValue( int value ) {
    this.value = value;
  }
  public String toString() {
    return "( " + value + " )";
  }
  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 );
} }
```

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 );
} }
```

Comparable Example - output

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

- TreeMap uses `compareTo()` to test for equality.
- HashMap relies on `equals()`, which CollectionTest does not override; it therefore tests for object identity, not the numeric value.

```
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
```

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 );
} } }
```

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;
}
```

Other Java Collections

- **LinkedList** – A doubly-linked list.
- **Stack**
- **WeakHashMap** – Entries are 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: