COMP241
Software Engineering Development
Lecture 11: Container classes 2

Mark Hall

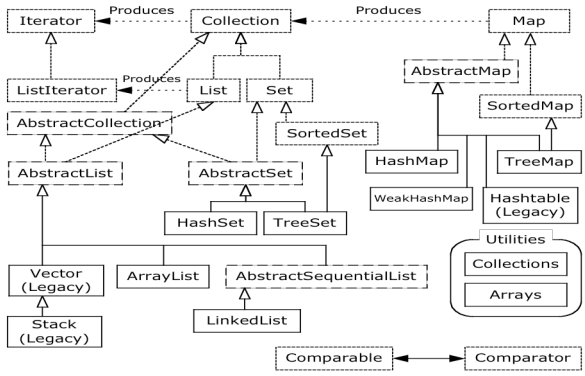
- Container taxonomy
- Collection functionality
- Iterators
- Set functionality
- List functionality
- Adapter design pattern
- Map functionality

THE UNIVERSITY OF WAIKATO
DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Beyond Arrays

- Containers in Java encompass two distinct concepts (interfaces):
  - `Collection:` a group of individual elements, often with some rule applied to them
    - A `List` must hold the elements in a particular sequence
    - A `Set` cannot have any duplicate elements
  - `Map`: a group of key-value object pairs
    - Could have been implemented as a `Collection` of pairs, but is clearer as a separate concept
    - Can be convenient to look at portions of a `Map` by creating a `Collection` to represent that portion
      - Eg. can get a `Set` of keys or `Collection` of values

Container Taxonomy



Container Taxonomy

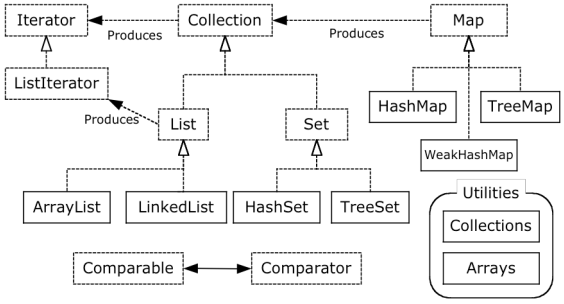
- Three container components
  - `Map`, `List` and `Set`—each with two or three implementations
- Dotted boxes represent **interfaces**
- Dashed boxes represent **abstract** classes
- Solid boxes are **concrete** classes
- Interfaces concerned with holding objects
  - `Collection`, `Map`, `List` and `Set`
  - Typically write most of your code to talk polymorphically to these interfaces
  - Specify precise type at point of creation:

```
List x = new LinkedList();
```


Container Taxonomy

- In the class hierarchy there are a number of abstract classes
  - They are tools that partially implement a particular interface
  - To implement your own container you would inherit from one of these abstract classes and do the minimal necessary work to make your new class
- The containers library is sufficiently powerful to satisfy almost all needs
  - Therefore, we can ignore abstract classes
- Can simplify the class hierarchy by focusing on interfaces and concrete classes

Simplified Container Taxonomy



Collection Functionality

- `Collection` interface (i.e. everything you can do with a `List` or `Set`)
  - `boolean add(Object)`—ensure that container holds the argument (returns false if not added)
  - `boolean addAll(Collection)`—adds all the elements in the argument (returns true if *any* are added)
  - `void clear()`—remove all elements
  - `boolean contains(Object)`—true if the container holds the argument
  - `boolean containsAll(Collection)`
  - `boolean remove(Object)/removeAll(Collection)`
  - `boolean retainAll(Collection)`—performs an intersection with the elements in the argument
  - `Object [] toArray()/ <T> T[] toArray(T[] a)`—return array containing all elements in the container
  - `Iterator iterator()` — returns an Iterator that can be used access the elements in the collection

Collection Functionality

- Notice that there is no `get()` method for random access in `Collection`
  - `Collection` also includes `Set`, which maintains its own internal ordering (making random access lookup meaningless)
- `Iterators` are the only way to fetch things back from a `Collection`

Iterators

- An `Iterator` provides a useful abstraction for moving through the sequence of objects held in a container
  - Allows the programmer to obtain objects *without* knowing or caring about the underlying structure of the sequence
  - Write generic code that can work on may types of containers

Iterators

- The Java `Iterator` is very simple, you can:
  - Ask a container to hand you an `Iterator`
  - Get the next object in the sequence by calling the `next()` method
  - See if there are any more objects in the sequence with `hasNext()`
  - Remove the last element returned by the iterator with `remove()`
- There is a more powerful `ListIterator` for `Lists`

Set Functionality

- `Set` has exactly the same interface as `Collection`, so there isn't any additional functionality
- `Sets` have different *behaviour* from `Lists`
  - Hold only one instance of each object value
  - `Set` interface does not guarantee it will maintain its elements in any particular order
- Two concrete implementations:
  - `HashSet`—for `Sets` where fast lookup time is important
    - Requires that stored `Objects` also define **`hashCode()`**
    - Provides constant time performance for basic operations (add, remove, contains), provided the hash function disperses elements properly
  - `TreeSet`—an ordered set backed by a tree structure
    - Provides log(n) time performance for basic operations

List Functionality

- Most commonly used methods in `List`
  - `add(int index, Object o)`—insert at the specified position
  - `set(int index, Object o)`—replace at the specified position
  - `get(int index)`—return the object at specified position
  - `iterator()`—get an iterator to the sequence
- Other useful methods:
  - `indexOf(Object o)`—return the index of the first occurrence of the element or -1
  - `subList(int fromIndex, int toIndex)`—return a view of this list from `fromIndex` (inclusive) to `toIndex` (exclusive)
  - `listIterator()`—get a `ListIterator` for this list

List Types

- There are two concrete implementations of List
  - `ArrayList`
    - Excels at randomly accessing elements
    - Slow when inserting and removing elements from the middle of the `List`
  - `LinkedList`
    - Provides optimal sequential access (slow for random access)
    - Inexpensive insertions and deletions from the middle of the `List`
    - Also has `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst` and `removeLast` methods
      - Allow `LinkedList` to be used as a **stack**, a **queue**, and a **deque**

Adapter Design Pattern

- Since the interface of `LinkedList` allows for the concepts of stack, queue and deque it is not great in terms of *cohesion*
- We can apply the **Adapter** design pattern and make our own Stack and Queue classes
  - Make use of LinkedList, but **adapt** its interface to a more conventional interface that clients expect

Adapter Design Pattern

- Adapter is similar to *Decorator*
  - Both make use of object **composition**
- Composition is preferable to inheritance because enclosing objects can only manipulate the enclosed object via its interface
  - This results in *loose* coupling between objects
- Inheritance results in classes that are *tightly* coupled to their base class
  - Internals of the base class are visible to the sub-classes

Loose coupling results in more flexible systems

Adapter Design Pattern

- Adapter is very similar to the Decorator design pattern
  - Both delegate work to another object
  - With Adapter, the relationship is typically set at **compile time**
  - With Decorator, the relationship is dynamically set at **runtime**
- However, the **intent** is different:

Adapter converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

- `InputStreamReader` and `OutputStreamWriter` apply the Adapter design pattern
  - Provide a bridge (or adapt) between byte streams and character streams

Adapting LinkedList to a Stack

- Stack—LIFO (last in, first out) container

```
// Adapt a linked list to a Stack interface using
// composition
public class Stack {
  private LinkedList mList = new LinkedList();
  public void push(Object o) {
    mList.addFirst(o);
  }
  public Object top() {
    return mList.getFirst();
  }
  public Object pop() {
     return mList.removeFirst();
  }
  public boolean isEmpty() {
    return mList.isEmpty();
  }
}
```

Adapting LinkedList to a Stack

- To have only stack behaviour, inheritance would have been inappropriate here
  - Result in a class with all the rest of the `LinkedList` methods
  - This was the very mistake made made by the Java 1.0 library designers with the `Stack` class (legacy class)
    - Java 1.0 `Stack` extends `Vector` (another legacy class, superceded in the new libraries by `ArrayList`)

Map Functionality

- ArrayList allows you to select from a sequence using a number (i.e. associates numbers to objects)
- A *map* (or *dictionary* or *associative array*) allows you to look up an object by using **another object**

Map Functionality

- Some methods in Map:
  - `put(Object key, Object val)`—adds a *value* and associates it with a *key*
  - `get(Object key)`—returns the value with the corresponding *key* (or null if the key is not in the map)
  - `containsKey(Object key)`, `containsValue(Object val)`—test to see if map contains key/value

Map Functionality

- There are two different types of `Map`
  - `HashMap`—implementation based on a hash table
    - Provides constant time performance for inserting and locating pairs (key, value)
    - Performance can be adjusted via constructors that allow you to set the *capacity* and *load factor* of the hash table
  - `TreeMap`
    - Maintains pairs in sorted order
    - Guarantees log(n) time performance for basic operations by using a balanced binary tree
    - Keys must implement `Comparable` or a `Comparator` must be supplied when constructing a `TreeMap`

Map Example

- Program to check the randomness of Math.random()
  - Random numbers *should* be uniformly distributed
  - Generate a bunch of random numbers and count the ones that fall into the various ranges
  - HashMap will help us associate <random num, count> pairs

Map Example

```
class Counter {
  int mCount = 1; // initialize count to 1
  public String toString() { return ""+mCount; }
}
public class Statistics {
  public static void main(String [] args) {
    Map<Integer, Counter> hm = new HashMap<Integer, Counter>();
    for (int i = 0; i < 10000; i++) {
      // produce a number between 0 and 20
      Integer r = new Integer((int)(Math.random() * 20));
      if (hm.containsKey(r)) {
        hm.get(r).mCount++;
      } else {
        hm.put(r, new Counter());
      }
    }
    System.out.println(hm);
  }
}
```

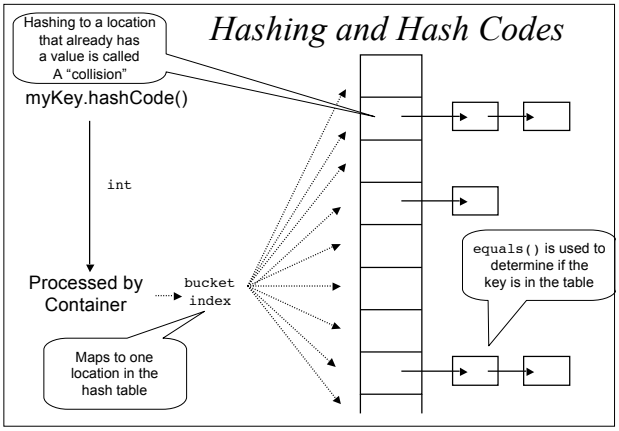
Map Example

- Why use `Counter` instead of `int` or `Integer` to hold values?
  - Can't use `int` because containers can only store objects
  - Can't use `Integer` because we need to update counts and Integer is an *immutable* class
- We can use `TreeMap` (which is a `SortedMap`) instead of `HashMap` in order to print out sorted pairs. `SortedMap` gives us:
  - `Object firstKey()/lastKey()`—lowest and highest key values
  - `SortedMap subMap(fromKey, toKey)`
  - `SortedMap headMap(toKey)`
  - `SortedMap tailMap(fromKey)`

Hashing and Hash Codes

- In the previous example we used Integer as a key for HashMap
  - Library classes (such as Integer and String) work fine as keys in hashed containers because **`hashCode()`** and **`equals()`** have already been implemented for us
- To use our own classes as keys we must provide suitable `hashCode` and `equals` methods
- Can't just use `hashCode` and `equals` inherited from `Object` as these use object *addresses*
  - An object that we use for lookup will not have the same address (and therefore will not hash to the same location) as the one stored in the container
  - Furthermore, equals is used to determine if the lookup key is equal to any in the table—again, this won't work for addresses

Hashing and Hash Codes



Hashing and Hash Codes

- The most important factor in creating a `hashCode()` is that it produces the same value for a given object regardless of when it is called
  - Otherwise you won't be able to retrieve the objects from a hashing-based container
  - If your `hashCode` depends on *mutable* data in the object then the user must be made aware that changing the data will change the key

Hashing and Hash Codes

- Don't use unique object info for `hashCode` (eg. object address), use info that identifies the object in a meaningful way
  - Eg. `String` uses the contents of the string to compute `hashCode`:

```
System.out.println("Hello".hashCode());
System.out.println("Hello".hashCode());
// two separate "Hello" String objects, same
// hashCode
```

- A good `hashCode` *should* result in an even distribution of values for optimal performance
  - Worst case: all hashCodes are equal—`HashMap` or `HashSet` degenerates to a linked list!

HashMap Performance Factors

- Terminology
  - *Capacity*—the number of buckets (locations) in the table
  - *Initial capacity*—the number of buckets when the table is created
    - `HashMap` and `HashSet` have constructors that allow you to specify the initial capacity
  - *Size*—the number of entries currently in the table

HashMap Performance Factors

- Terminology (cont.)
  - *Load factor*—size/capacity (empty table = 0, half full = 0.5 etc)
    - A lightly loaded table will have fewer *collisions* and so is optimal for lookups and insertions, but is slower for traversing with an `Iterator`
    - `HashMap` and `HashSet` have constructors that allow you to specify the load factor
    - When the load factor is reached the container automatically increases the capacity (roughly doubles it) and re-distributes the contents (re-hashing
    - High load factors decreases space requirements but increases the lookup cost
    - The default load factor (0.75) is a reasonable tradeoff between time and space costs

The Collections Class

- There are a number of useful static utility methods in the `Collections` class
  - Methods to sort and search `Lists`—have same names and signatures as those in `Arrays`
  - `max/min(Collection)`—produces the maximum or minimum element in the argument using the natural comparison method of objects in the `Collection`
  - `max/min(Collection, Comparator)`—produces the maximum or minimum element in the `Collection` using the supplied `Comparator`
  - `reverse(List)`—reverses the elements of a `List`
  - `copy(List dest, List src)`
  - `shuffle(List l, Random r)`—randomly permute the list