


[Dashboard](#) > [Training](#) > [...](#) > [Object Boot Camp](#) > [Collections](#)

Search

ThoughtWorks® Collections

Training

Welcome [Luca Minudel](#) | [History](#) | [Preferences](#) | [Log Out](#) [View](#)[Attachments \(1\)](#)[Info](#)[Review](#) [Browse Space](#)Added by [Rolf Russell](#), last edited by [David S Wood](#) on Dec 07, 2006 ([view change](#))Labels: (None) [EDIT](#)

Collections

Session Objectives

By the end of this session students should ...

- Be familiar with the Java Collections framework
- Understand the differences between Interfaces, Abstract Classes, Concrete Classes
- Understand what it means to buy a collection
- Be familiar with the Collections and Arrays classes
- Understand the usages of Comparable and Comparator and the concept of natural ordering
- Be able to effectively use Iterators
- Understand how to preserve state while testing a potential scenario using a cloned Collection

Session Overview

Activity
Inner Classes
The Java Collections Framework
Lab - Golf

Session Notes

Inner Classes

Describe

- class declared within another class (called the containing class). It does work for the containing class.
- tightly coupled to the containing class. has the full state of the containing class available to it.
- if something outside of the containing class is interested in working directly with the inner class, it probably isn't a good candidate for an inner class and should be promoted to full class.

Examples

- collections framework uses inner classes in several places:
 - iterators which need to know how to iterate over the internal storage mechanism of the collection
 - [Map.Entry] whose sole purpose is to help map do its work. (Note that [Map.Entry] is available outside the Map class. This is a convenience for the Collections framework. You could just use the keys and values directly).
- inner classes are used for state & strategy patterns because they need access to the full state of the containing class and should never be instantiated outside of the scope of the class.
 - strategy - isolating an algorithm like sorting
 - state - managing the states of an object through its lifecycle

The Java Collections Framework

"The Collections framework is a pretty good example of a well written java framework. You will use it frequently."

This active lecture will drive out understanding of the Collection types, their inheritance tree, and the differences between interfaces, abstract classes and concrete classes.

Collection

- Ask what the job of a Collection is? To understand a group of objects.
- Essentially it's a bag of items.
- No concept of order
- Allows duplication
- Discuss the methods available on Collection. The general style is to ask the class *give me a method on Collection* and then ask *what is returned by that method* and *why*
- add(element), remove(element), contains(element), size(), iterator(), isEmpty(), clear(), removeAll(collection), retainAll(collection), toArray()
- Collection is an interface. Ask the class what the characteristics of an interface are. Draw the Interface column of the *trait diagram* below on the board.

Iterator

- Ask what the job of an Iterator is? To understand a traversal of a collection.
- Hand analogy - this is a visual analogy to help the students: If the fingers on your hand represent the 5 elements of a collection, the iterator sits in the spaces between your fingers. It starts before your thumb and you can ask hasNext(). Requesting next() will advance the iterator to the space between your thumb and forefinger and return the thumb. hasNext() now asks about the forefinger. Requesting next() again will advance the iterator between the forefinger and middle finger and return the forefinger. Requesting remove() will remove the forefinger (not the middle finger because you may want to inspect what you remove before removing and you have no way of seeing the middle finger yet).
- Iterators are implemented specific to the type of Collection and thus can be optimised for that type.
- If while you are iterating someone else changes the underlying collection, the iterator will throw a ConcurrentModificationException
- As class why iterator only has next()? Because iterator has no knowledge of

Trait Diagram

Trait	Interface	Abstract Class	Concrete Class
Visibility	Public	Public, Package, (less if inner)	Public, Package, (less if inner)
Can Instantiate	No	No	Yes
Method Types	Abstract	Abstract or Concrete	Concrete
Method Visibility	Public	Any	Any
Can Have Attributes	No	Yes	Yes
Can Have Constants	Yes	Yes	Yes
Multiple Inheritance	Yes	No	No

Multiple Inheritance Discussion

There are 2 types of inheritance:

1. *inheritance of behaviour* - like implemented methods, fields, etc
2. *inheritance of contract* - like defined method signatures

Danger of multiple inheritance comes when you are mixing behaviour from 2 different parents. Draw "diamond of death". The issue is that it is not clear which superclass implementation of the method to call.

Multiple interface inheritance still allows an object to inherit methods and to behave polymorphically on those methods. The inheriting object just doesn't get an implementation free ride.

AbstractCollection

- Default implementations for many methods using iterator()
- Draw Collection and AbstractCollection on the board as in the *class diagram* below. From here on fill out the *class diagram* with each new collection.
- Abstract class. Ask the class what the traits of an abstract class are. Fill in the abstract class column of the *trait diagram*.

Class Diagram



List

- Ask what the job of a List is? To understand an ordered group of objects.
- Ordered, allows duplicates, not sorted
- Allows duplicates
- get(index), indexOf(element), remove(index), add(index, element), listIterator() (can go backwards), set(index, element)

AbstractList

- Default implementations for many methods using listIterator()

ArrayList

- Backed by an array
- Allows null

AbstractSequentialList

LinkedList

- Prefer LinkedList if adding/removing often from anywhere but the end of the list and not doing index based lookup frequently
- What could you use a LinkedList for? elevator of people - last person in is the first person out
- Adds stack operations
- getFirst(), getLast(), peek() look at top of stack, poll() removes top of stack
- Allows null

Vector

- Old implementation
- What is the key difference between a Vector and an ArrayList? Vectors are synchronized. But you can synchronize an ArrayList anyway so there isn't much reason to ever use Vectors

Set

- Ask what the job of a Set is? To understand a group of unique objects.
- no duplicates
- Allows null

HashSet

- Standard implementation of Set
- Does not have predictable iteration order. The iteration order may change over time.

LinkedHashSet

- ordered: backed by a doubly linked list as well as HashSet to make this possible.
- What could you use a LinkedHashSet for? an invoice where you don't want the same item to appear twice, but you want everything to retain the same order

SortedSet

- sorted set
- What could you use a SortedSet for? a customer's address history (which includes dates & address)

TreeSet

- Standard implementation of SortedSet
- You can pass in a comparator to the constructor
- headSet(toElement), tailSet(fromElement), subSet(fromElement, toElement)

Map

- Ask what the job of a Map is? To understand a group of key-value pairs.
- A set of key - value pairs.
- It cannot contain duplicate keys.
- Each key can map to only one value.
- A map is not a Collection
- What could you use a Map for? dictionary
- get(key), put(key, value), remove(key), keySet(), values() returns Collection, entrySet returns Set of [Map.Entries]

HashMap

- Standard implementation of Map
- No guarantee of order

LinkedHashMap

- ordered map

TreeMap

- sorted map

HashTable

- Old implementation. Synchronized like Vector

Comparator vs. Comparable

- Both are interfaces
- Comparable for natural order
 - discuss 'natural order' if it hasn't already been introduced.
 - implement compareTo() on the class (ie. like we implemented betterThan())
 - returns '1', '0' or '-1' which map to 'greater than', 'equal' or 'less than'. Returning '0' allows sorting algorithms to be more efficient by not switching the places of equal objects and also this is guaranteed by [Collections.sort\(\)](#).
 - it is recommended that compareTo() be consistent with equals(). (ie. when compareTo() returns '0' then equals() returns true)
- Comparator for all else
 - for example you might want to compare Dogs by length
 - a new class that knows how to compare 2 objects. For example a comparator of Dogs by length.

- now if we let another class see the length of the Dog, aren't we breaking encapsulation? How can we implement a comparator class without breaking encapsulation? An inner class!!!!

Collections

- a bunch of helper methods that have been separated from the main collection classes to avoid polluting the interface.
- `sort(list)` & `sort(list, comparator)`
- synchronization methods. Return a new collection that is synchronized and backed by the original collection. Synchronized means that the collection is thread safe and multiple threads can act upon the collection at the same time without unexpected behaviour.
- `EMPTY_LIST`, `EMPTY_MAP`, `EMPTY_SET` constants. These are objects you can use as collections but no operations will actually be performed on them. In other words adding to `EMPTY_LIST` won't throw an exception but also won't add anything to the list.
 - these are NullObjects which we will talk more about later.
- `min(collection)` & `max(collection)`: Return min/max of collection.
- `shuffle(list)` & `shuffle(list, random)`: Shuffles the list like a deck of cards.
- `reverse(list)`: Reverses the list in place.
- `reverseOrder()`, `reverseOrder(comparator)`: Returns a Comparator that imposes the reverse of the (natural) order of a collection
- `unmodifiableList(list)`, `unmodifiableSet(set)`, etc: Returns a List/Set/etc that contains the same elements but cannot be modified.

Arrays

- `asList(array)`: Returns a List backed by the array.
- `equals(array, array)`: Calls equals on the objects/primitives in the arrays as opposed to the arrays themselves.
- `fill(array, value)`
- `sort(array)`

Buying a collection

- Let's say we wanted to implement a list of people. Write *public class Employees extends ArrayList* on board. What do you think about the Employees class?
- Discussion
 - ArrayList has methods that do not match the job of Employees. For example what does `toArray()` mean to Employees?
 - ArrayList has methods that Employees may not need/want. For example `removeAll()`. This violates YAGNI and clutters the interface.
 - breaks encapsulation because Employees gives away all of its data.
- How else could we implement Employees? By wrapping the ArrayList in the Employees class (*buying the collection*) and then implementing the Employees methods using the bought collection without exposing it directly. Write the following code on the board:
Error formatting macro: code: Traceback (innermost last): File "<string>", line 1, in ? ImportError: no module named pygments

```
// Understands a group of employees
public class Employees {
    private ArrayList group = new ArrayList();
    public void add(Employee employee)

    Unknown macro: { group.add(employee); }

}
```

- This is better, but I still see an issue in the code. What could we improve? Is employees an ordered list?

No, nothing requires it to be ordered - not the job nor the add() method. Could we use something more generic? Yes, we could declare *group* to be a Collection instead of an ArrayList.

- Using the most generic thing possible hides implementation detail.
- It makes the code easier to change. For example you might want to make each Employee unique in Employees by using a set instead.

Lab - Golf

Have the class model parties showing up to play golf and interacting with a clubhouse attendant. Parties can have 1, 2, 3, or 4 members and will be combined with other parties to obtain tee groups of 1, 2, 3, or 4 players such that golfers get assigned tee times fairly. The attendant wants to know how many groups must be allowed on the course to clear the line because they want to go to Starbucks for some coffee.

Fair is defined as

- first party in line always plays
- remaining parties join group in order they are in line if group won't exceed 4
- search stops when group is full or entire line has been searched

Common Mistakes

- Creating too many objects.
- Using something other than Integer to represent the party.
- Not using an Iterator to delete parties and thus getting a ConcurrentModificationException.

Note: This lab has only been run twice and as such still needs some tweaking. It took quite a while to implement the last time so it would be nice to be able to simplify it, but it is important to need the Collection and the Iterator so just asking who is in the first group to play does not satisfy that.

Equipment Required

- printouts of Collections framework interfaces as instructor cheatsheets

 0 comments |  [Add Comment](#)