Eric Roberts
CS 106A

Handout #54
February 24, 2010

Slides for the Collections Lecture

Collections

Eric Roberts
CS 106A
February 24, 2010

Extensible *vs.* Extended Languages

- As an undergraduate at Harvard, I worked for several years on the PPL (Polymorphic Programming Language) project under the direction of Professor Thomas Standish.
- In the early 1970s, PPL was widely used as a teaching language, including here in Stanford's CS 106A.
- Although PPL is rarely remembered today, it was one of the first languages to offer syntactic extensibility, paving the way for similar features in more modern languages like C++.
- In a reflective paper entitled "PPL: The Extensible Language That Failed," Standish concluded that programmers are less interested in languages that are *extensible* than they are in languages that have already been *extended* to offer the capabilities those programmers need. Java's collection classes certainly fall into this category.

The `ArrayList` Class Revisited

- You have already seen the `ArrayList` class in Chapter 11.8. The purpose of this lecture is to look at the idea behind the `ArrayList` class from a more general perspective that paves the way for a discussion of the Java Collection Framework.
- The most obvious difference between the `ArrayList` class and Java's array facility is that `ArrayList` is a full-fledged Java class. As such, the `ArrayList` class can support more sophisticated operations than arrays can. All of the operations that pertain to arrays must be built into the language; the operations that apply to the `ArrayList` class, by contrast, can be provided by extension.

The `HashMap` Class

- The `HashMap` class is one of the most valuable tools exported by the `java.util` package and comes up in a surprising number of applications (including FacePamphlet).
- The `HashMap` class implements the abstract idea of a ***map***, which is an associative relationship between keys and values. A ***key*** is an object that never appears more than once in a map and can therefore be used to identify a ***value***, which is the object associated with a particular key.
- Although the `HashMap` class exports other methods as well, the essential operations on a `HashMap` are the ones listed in the following table:

`new HashMap()`	Creates a new `HashMap` object that is initially empty.
map`.put(`*key*`,` *value*`)`	Sets the association for *key* in the map to *value*.
map`.get(`*key*`)`	Returns the value associated with *key*, or `null` if none.

Generic Types for Keys and Values

- As with the `ArrayList` class introduced in Chapter 11, Java allows you to specify the types for names and keys by writing that information in angle brackets after the class name. For example, the type designation `HashMap<String,Integer>` indicates a `HashMap` that uses strings as keys to obtain integer values.
- The textbook goes to some length to describe how to use the `ArrayList` and `HashMap` classes in older versions of Java that do not support generic types. Although this information was important when I wrote those chapters, Java 5.0 and its successors are now so widely available that it doesn't make sense to learn the older style.

A Simple `HashMap` Application

- Suppose that you want to write a program that displays the name of a state given its two-letter postal abbreviation.
- This program is an ideal application for the `HashMap` class because what you need is a map between two-letter codes and state names. Each two-letter code uniquely identifies a particular state and therefore serves as a key for the `HashMap`; the state names are the corresponding values.
- To implement this program in Java, you need to perform the following steps, which are illustrated on the following slide:
  1. Create a `HashMap` containing all 50 key/value pairs.
  2. Read in the two-letter abbreviation to translate.
  3. Call `get` on the `HashMap` to find the state name.
  4. Print out the name of the state.

The `PostalLookup` Application

```
public void run() {
   HashMap<String,String> stateMap = new HashMap<String,String>();
   initStateMap(stateMap);
   while (true) {
      String code = readLine("Enter two-letter state abbreviation: ");
      if (code.length() == 0) break;
      String state = stateMap.get(code);
      if (state == null) {
         println(code + " is not a known state abbreviation");
      } else {
         println(code + " is " + state);
      }
   }
}
```

stateMap

```
PostalLookup
Enter two-letter state abbreviation: HI
HI is Hawaii
Enter two-letter state abbreviation: WI
WI is Wisconsin
Enter two-letter state abbreviation: VE
VE is not a known state abbreviation
Enter two-letter state abbreviation:
```

```
AL=Alabama
AK=Alaska
AZ=Arizona
...
FL=Florida
GA=Georgia
HI=Hawaii
...
WI=Wisconsin
WY=Wyoming
```

Implementation Strategies for Maps

There are several strategies you might choose to implement the map operations `get` and `put`. Those strategies include:

1. ***Linear search in parallel arrays.*** Keep the two-character codes in one array and the state names in a second, making sure that the index numbers of the code and its corresponding state name always match. Such structures are called ***parallel arrays***. You can use linear search to find the two-letter code and then take the state name from that position in the other array.
2. ***Binary search in parallel arrays.*** If you keep the key array sorted by the two-character code, you can use binary search to find the key. Using this strategy improves the performance considerably.
3. ***Table lookup in a two-dimensional array.*** In this specific example, you could store the state names in a 26×26 string array in which the first and second indices correspond to the two letters in the code. You can now find any code in a single array operation, although this performance comes at a cost in memory space.

The Idea of Hashing

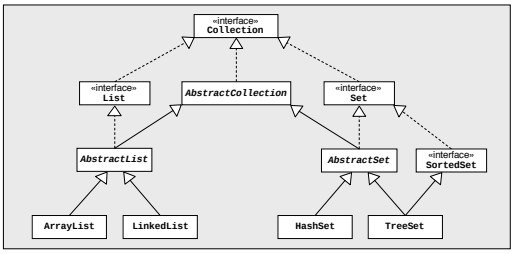
- The third strategy on the preceding slide shows that one can make the `get` and `put` operations run very quickly, even to the point that the cost of finding a key is independent of the number of keys in the table. This level of performance is possible only if you know where to look for a particular key.
- To get a sense of how you might achieve this goal in practice, it helps to think about how you find a word in a dictionary. You certainly don't start at the beginning at look at every word, but you probably don't use binary search either. Most dictionaries have thumb tabs that indicate where each letter appear. Words starting with *A* are in the *A* section, and so on.
- The `HashMap` class uses a strategy called ***hashing***, which is conceptually similar to the thumb tabs in a dictionary. The critical idea is that you can improve performance enormously if you use the key to figure out where to look.

The Java Collections Framework

- The `ArrayList` and `HashMap` classes are part of a larger set of classes called the ***Java Collections Framework***, which is part of the `java.util` package.
- The classes in the Java Collections Framework fall into three general categories:
  1. ***Lists***. Ordered collections of values that allow the client to add and remove elements. As you would expect, the `ArrayList` class falls into this category.
  2. ***Sets***. Unordered collections of values in which a particular object can appear at most once.
  3. ***Maps***. Structures that create associations between keys and values. The `HashMap` class is in this category.
- The next slide shows the Java class hierarchy for the first two categories, which together are called ***collections***.

The `Collection` Hierarchy

The following diagram shows the portion of the Java Collections Framework that implements the `Collection` interface. The dotted lines specify that a class implements a particular interface.



`ArrayList` *vs.* `LinkedList`

- If you look at the left side of the collections hierarchy on the preceding slide, you will discover that there are two classes in the Java Collections Framework that implement the `List` interface: `ArrayList` and `LinkedList`.
- Because these classes implement the same interface, it is generally possible to substitute one for the other.
- The fact that these classes have the same effect, however, does not imply that they have the same performance characteristics.
  - The `ArrayList` class is more efficient if you are selecting a particular element or searching for an element in a sorted array.
  - The `LinkedList` class can be more efficient if you are adding or removing elements from a large list.
- Choosing which list implementation to use is therefore a matter of evaluating the performance tradeoffs.

The `Set` Interface

- The right side of the collections hierarchy diagram contains classes that implement the **`Set`** interface, which is used to represent an unordered collection of objects. The two concrete classes in this category are **`HashSet`** and **`TreeSet`**.
- A set is in some ways a stripped-down version of a list. Both structures allow you to add and remove elements, but the set form does not offer any notion of index positions. All you can know is whether an object is present or absent from a set.
- The difference between the **`HashSet`** and **`TreeSet`** classes reflects a difference in the underlying implementation. The **`HashSet`** class is built on the idea of hashing; the **`TreeSet`** class is based on a structure called a *binary tree*, which you will learn more about if you go on to CS 106B. In practice, the main difference arises when you iterate over the elements of a set, which is described on the next slide.

Iteration in Collections

- One of the most useful operations for any collection is the ability to run through each of the elements in a loop. This process is called ***iteration***.
- The **`java.util`** package includes a class called **`Iterator`** that supports iteration over the elements of a collection. In older versions of Java, the programming pattern for using an iterator looks like this:

```
Iterator iterator = collection.elements();
while (iterator.hasNext()) {
   type element = (type) iterator.next();
   . . . statements that process this particular element . . .
}
```

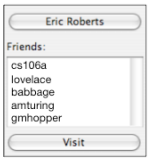
- Java Standard Edition 5.0 allows you to simplify this code to

```
for (type element : collection) {
   . . . statements that process this particular element . . .
}
```

Iteration Order

- For a collection that implements the **`List`** interface, the order in which iteration proceeds through the elements of the list is defined by the underlying ordering of the list. The element at index 0 comes first, followed by the other elements in order.
- The ordering of iteration in a **`Set`** is more difficult to specify because a set is, by definition, an unordered collection. A set that implements only the **`Set`** interface, for example, is free to deliver up elements in any order, typically choosing an order that is convenient for the implementation.
- If, however, a **`Set`** also implements the **`SortedSet`** interface (as the **`TreeSet`** class does), the iterator sorts its elements so they appear in ascending order according to the **`compareTo`** method for that class. An iterator for a **`TreeSet`** of strings therefore delivers its elements in alphabetical order.

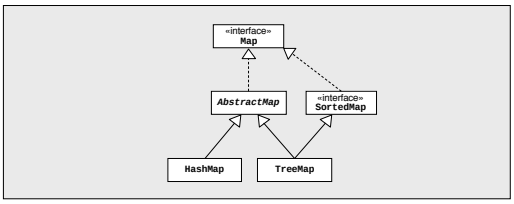
Exercise: Sorting the Friends List



- In the FacePamphlet application, one of the things you have to do to achieve Milestone #4 is update the friends list from the repository.
- When you ask for the friends list, the repository gives it to you in the order in which the friends were added to the list, which makes it harder to find names.
- How would you go about writing an **`UpdateFriendsList`** method that, as part of its operation, made sure that the names in the friends list appear in alphabetical order?

The `Map` Hierarchy

The following diagram shows the portion of the Java Collections Framework that implements the **`Map`** interface. The structure matches that of the **`Set`** interface in the **`Collection`** hierarchy. The distinction between **`HashMap`** and **`TreeMap`** is the same as that between **`HashSet`** and **`TreeSet`**, as illustrated on the next slide.

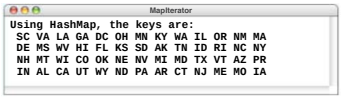


Iteration Order in a `HashMap`

The following method iterates through the keys in a map:

```
private void listKeys(Map<String,String> map, int nPerLine) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   Iterator<String> iterator = map.keySet().iterator();
   for (int i = 1; iterator.hasNext(); i++) {
      print(" " + iterator.next());
      if (i % nPerLine == 0) println();
   }
}
```

If you call this method on a **`HashMap`** containing the two-letter state codes, you get:

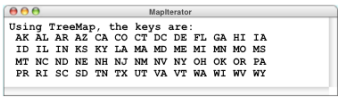


Iteration Order in a `TreeMap`

The following method iterates through the keys in a map:

```
private void listKeys(Map<String,String> map, int nPerLine) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   Iterator<String> iterator = map.keySet().iterator();
   for (int i = 1; iterator.hasNext(); i++) {
      print(" " + iterator.next());
      if (i % nPerLine == 0) println();
   }
}
```

If you call instead this method on a `TreeMap` containing the same values, you get:



The `Collections` Toolbox

- The `Collections` class (not the same as the `Collection` interface) exports several static methods that operate on lists, the most important of which appear in the following table:

`binarySearch(`*list*`, `*key*`)`	Finds *key* in a sorted list using binary search.
`sort(`*list*`)`	Sorts a list into ascending order.
`min(`*list*`)`	Returns the smallest value in a list.
`max(`*list*`)`	Returns the largest value in a list.
`reverse(`*list*`)`	Reverses the order of elements in a list.
`shuffle(`*list*`)`	Randomly rearranges the elements in a list.
`swap(`*list*`, `p_1`, `p_2`)`	Exchanges the elements at index positions p_1 and p_2.
`replaceAll(`*list*`, `x_1`, `x_2`)`	Replaces all elements matching x_1 with x_2.

- The `java.util` package exports a similar `Arrays` class that provides the same basic operations for any array.

Exercise: Trigraph Frequency

In the lecture on arrays, one of the examples was a program to count letter frequencies in a series of lines, which was useful in solving cryptograms. As Edgar Allan Poe explained in his short story *The Gold Bug*, it is often equally useful to look at how often particular sequences of two or three letters appear in a given text. In cryptography, such sequences are called ***digraphs*** and ***trigraphs***.

For the rest of today's lecture, our job is to write a program that reads data from a text file and writes out a complete list of the trigraphs within it, along with the number of times each trigraph occurs. To be included in the list, a trigraph must consist only of letters; sequences of characters that contain spaces or punctuation should not be counted.

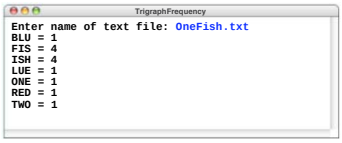
Trigraph Example

For example, if a data file contains the short excerpt

`OneFish.txt`

```
One fish, two fish, red fish, blue fish.
```

the trigraph program should report the following:



Note that the output is ordered alphabetically. Between now and Friday, give some thought as to how you might change the code so that the output appears in order of descending frequency.