Eric Roberts
CS 106A

Handout #48
May 18, 2012

Collections and Maps

Collections and Maps

Eric Roberts
CS 106A
May 18, 2012

Extensible *vs.* Extended Languages

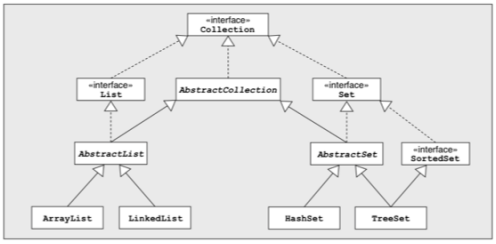
- As an undergraduate at Harvard, I worked for several years on the PPL (Polymorphic Programming Language) project under the direction of Professor Thomas Standish.
- In the early 1970s, PPL was widely used as a teaching language, including here in Stanford's CS 106A.
- Although PPL is rarely remembered 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 Java Collections Framework

- The `ArrayList` class from Wednesday's lecture is part of a larger set of classes called the ***Java Collections Framework*** defined as part of the `java.util` package.
- The classes in the Java Collections Framework fall into three general categories:
  1. ***Lists*** are ordered collections of values that allow the client to add and remove elements. As you would expect, `ArrayList` falls into this category.
  2. ***Sets*** are unordered collections of values in which a particular object can appear at most once.
  3. ***Maps*** are structures that define an association between keys and values.

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`.
- Both sets and lists allow you to add and remove elements, but sets do not support the notion of an index position. All you can know is whether an object is present or absent from a set.
- The difference between the `HashSet` and `TreeSet` classes is primarily a difference in implementation. The `HashSet` class is built on an structure called a *hash table,* while the `TreeSet` class is based on a structure called a *binary tree.* In practice, the only difference arises when you iterate over the elements of a set, as described on the next slide.
- You will have a chance to learn the details of these techniques if you take CS 106B.

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.

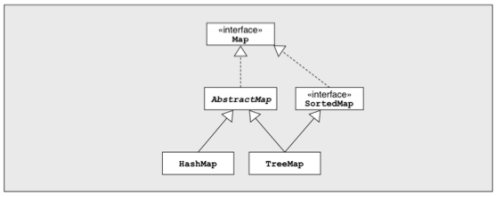
Maps

- A ***map*** is a data structure that implements 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.
- Maps are one of the most useful tools in the Java Collections Framework and come up in all sorts of applications, including NameSurfer.
- Although maps export other methods as well, the following operations are the most useful:

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.
map`.containsKey(`*key*`)`	Returns `true` if the map contains an association for *key*.

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` and affects the iteration order.



Constructing Maps

- Even though it represents the root of the hierarchy on the previous slide, the `Map` type in Java is an *interface* rather than a *class*. One implication of this design decision is that you cannot use the identifier `Map` as a constructor. You must instead create either a `HashMap` or a `TreeMap`.
- As with the `ArrayList` class, Java allows you to specify the types for keys and values by enclosing 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 `HashMap` in older versions of Java that do not support generic types. Although this strategy was necessary when the book first came out, 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

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

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:
```

Iteration and Maps

- In Java, the classes in the `Map` hierarchy don't support iteration directly. If you want to go through all the keys in a map, you have to call the `keySet` method, which returns a set consisting of all the keys in the map. Once you have the set of keys, you can then iterate through that.
- The iteration order for a map depends on whether you have created a `HashMap` or a `TreeMap`. The `keySet` method for a `HashMap` returns a `HashSet`, which has an undefined iteration order. The `keySet` method for a `TreeMap` returns a `TreeSet`, which guarantees that the keys will be returned in increasing order.
- The `HashMap` class is slightly more efficient than `TreeMap` and is much more common in practice. As a Java programmer, you do have both options.

Iteration order in a `HashMap`

The following method iterates through the keys in a map:

```
private void listKeys(Map<String,String> map) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   for (String key : map.keySet()) {
      print(key + " ");
   }
   println();
}
```

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

MapIterator

```
Using HashMap, the keys are:
RI VT HI ME VA MI DE ID IA MD MA AR IL UT IN MN
AZ MO MT MS NH NJ NM AK AL TX NC ND NE NY GA NV
TN CA OK OH WY FL SD SC CT WV KY WI KS OR LA WA
CO PA
```

Iteration Order in a `TreeMap`

The following method iterates through the keys in a map:

```
private void listKeys(Map<String,String> map) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   for (String key : map.keySet()) {
      print(key + " ");
   }
   println();
}
```

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

MapIterator

```
Using TreeMap, the keys are:
AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS
KY LA MA MD ME MI MN MO MS MT NC ND NE NH NJ NM
NV NY OH OK OR PA RI SC SD TN TX UT VA VT WA WI
WV WY
```

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:

TrigraphFrequency

```
Enter name of text file: OneFish.txt
BLU = 1
FIS = 4
ISH = 4
LUE = 1
ONE = 1
RED = 1
TWO = 1
```