Eric Roberts
CS 106A

Handout #33
February 5, 2016

Abstract Data Types

Abstract Data Types
(Lists, Sets, and Maps)

Eric Roberts
CS 106A
February 5, 2016

Outline

1. Type abstraction
2. Lists, sets, and maps
3. Parameterized types, wrapper classes, and autoboxing
4. Creating collections
5. Iteration
6. Applications and exercises
7. Story time: Past Graphics Contest winners

Schedule Change

To give everyone a little more time to create the greatest graphics contest entries of all time, I'm moving the due date for the Graphics Contest to Wednesday, February 10.

Type Abstraction

- One of the most important advantages of the object-oriented paradigm is the idea of ***type abstraction***, in which the goal is to think about types in terms of their high-level behavior rather than their low-level implementation.
- In computer science, types that are defined by their behavior are called ***abstract data types*** or ***ADT***s. Our goal in this class will be to use abstract data types as much as possible, leaving the issues of how those types are implemented to CS 106B.
- In this class, my goal is to introduce three abstract data types:
  - The `List` ADT is used to represent collections of values in which the order is typically important.
  - The `Set` ADT is used to represent unordered collections of distinct values.
  - The `Map` ADT is used to represent collections in which the values are associated with identifying keys.

Examples of Lists, Sets, and Maps

- The conceptual list, set, and map types come up all the time when you are trying to model aspects of the real world.
- Consider, for example, the Enigma machine I talked about on Wednesday. I argue that each of these conceptual types occur:
  - The rotors inserted into the machine form a *list*. The order of the rotors matters, and it makes sense to talk about a first, second, and third rotors.
  - The stock rotors available to the operators form a *set*. Although the stock rotors are numbered for convenience, they can be used in any order and there is no duplication.
  - The permutation implemented by the rotors is a *map*. Each rotor takes an input letter that serves as the *key* and associates it with some other letter that represents the *value*.

Exercise: Real-World Examples

- What real-world examples can you identify for each of the abstract types `List`, `Set`, and `Map`?
- Talk with your neighbors for the next few minutes and come up with at least one example of each type.

Parameterized Types

- The `List`, `Set`, and `Map` ADTs represent collections of values. As with all Java values, the elements contained in a collection have data types, and it is necessary to specify those types whenever you use a collection type. Java calls such types ***parameterized types***.
- In Java, the complete name of a parameterized type includes the data type of the value written in angle brackets after the type name. For example, a list containing strings would have the type `List<String>`. Similarly, a set containing Java colors would have the type `Set<Color>`.
- Maps require two type parameters inside the angle brackets, one for the key and one for the value. If, for example, you wanted to create a map from color names to Java colors, you would use the type `Map<String,Color>`.

Wrapper Classes

- In Java, parameterized types can be used only with object types and not with primitive types. Thus, while it is perfectly legal to use the type name

  `List<String>`

  it is not legal to write

  `List<int>` 

- To get around this problem, Java defines a ***wrapper class*** for each of the primitive types:

`boolean`	`Boolean`	`float`	`Float`
`byte`	`Byte`	`int`	`Integer`
`char`	`Character`	`long`	`Long`
`double`	`Double`	`short`	`Short`

Autoboxing

- The existence of wrapper classes makes it possible to define collections of the primitive types. You can, for example, declare a list of integers using the type name `List<Integer>`.
- When you specify a wrapper class as a type parameter, Java allows you to work with the collection as if it contained values of the corresponding primitive type because Java automatically converts values between the primitive types and their wrapper classes. This facility is called ***autoboxing***.
- Java's autoboxing conversions make it *appear* as if one is storing primitive values in a collection, even though the element type is declared to be a wrapper class.

Creating Collections

- The `List`, `Set`, and `Map` types in Java are defined as *interfaces* rather than as *classes*. Interfaces embody the idea of type abstraction because they define only the behavior of a type and do not specify an implementation at all.
- For the most part, you can ignore entirely the fact that these abstract types are interfaces. The time at which you need to remain conscious of the distinction is when you need to create an instance of a collection. Interfaces in Java do not have constructors, which means that you must call the constructor for some concrete class that implements the interface.
- For now, we'll use the classes `ArrayList`, `TreeSet`, and `TreeMap` as the concrete representations of these abstract types. You'll have a chance to learn about other possible representations of these interfaces later in the quarter.

Iterating over a Collection

- One of the common operations that clients need to perform when using a collection is to iterate through the elements.
- Modern versions of Java defines a special version of the `for` statement that serves just this purpose:

```
for (type variable : collection) {
   . . . code to process the element stored in the variable . . .
}
```

- For example, the following code prints out every value in a `List<String>` stored in the variable `lineList`:

```
for (String str : lineList) {
   println(str);
}
```

Selected Methods in the `List` Interface

`list.size()` Returns the number of values in the list.
`list.isEmpty()` Returns `true` if the list is empty.
`list.set(i, value)` Sets the i^{th} entry in the list to `value`.
`list.get(i)` Returns the i^{th} entry in the list.
`list.add(value)` Adds a new value to the end of the list.
`list.add(index, value)` Inserts the value before the specified index position.
`list.remove(index)` Removes the value at the specified index position.
`list.clear()` Removes all values from the list.

The `readParagraph` Method

```
/**
 * Reads a "paragraph" from the console, which ends with the
 * first blank line.  The method returns a list of strings.
 *
 * @return A list of strings read from the user
 */

private List<String> readParagraph() {
   List<String> paragraph = new ArrayList<String>();
   while (true) {
      String line = readLine();
      if (line.length() == 0) break;
      paragraph.add(line);
   }
   return paragraph;
}
```

Converting a Paragraph to Pig Latin

```
public void run() {
   println("Enter a paragraph ending with a blank line.");
   List<String> paragraph = readParagraph();
   for (String line : paragraph) {
      println(translateLine(line));
   }
}
```

PigLatinParagraph

```
Enter a paragraph ending with a blank line.
Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
Not to praise him.

iendsFray, omansRay, ountrymencay,
endLay emay ouryay earsway;
Iway omecay otay urybay aesarCay,
otNay otay aisepray imhay.
```

Selected Methods in the `Set` Interface

`set.size()`	Returns the number of values in the set.
`set.isEmpty()`	Returns `true` if the set is empty.
`set.add(element)`	Adds a new element to the set.
`set.remove(element)`	Removes the element from the set.
`set.contains(element)`	Returns `true` if the set contains the specified element.
`set.clear()`	Removes all values from the set.

Creating a Set of English Words

- The next few examples and one of the questions on next week's section handout work with a `Set<String>` that contains every word in the English language (or at least those that were in the second edition of the Scrabble dictionary).
- Creating that set requires reading words from a file, which you won't learn until later in the class. For the time being, we'll rely on a class called `WordSet` and a data file called `EnglishWords.txt`, both of which are included with the code for this lecture. You can use the following declaration to create a set containing all English words:

```
Set<String> english = new WordSet("EnglishWords.txt");
```

- The `WordSet` class implements the `Set<String>` interface, so it is legal to initialize the variable `english` in this way.

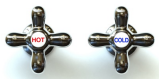
Example: Listing the Two-Letter Words

- A good Scrabble player needs to know the list of two-letter words because they provide the best opportunity to "hook" new words onto existing letter patterns on the board.
- If you use the `WordSet` model defined on the preceding slide, the Java code required to list the legal two-letter words reads almost as fluidly as English:

```
Set<String> english = new WordSet("EnglishWords.txt");
for (String word : english) {
   if (word.length() == 2) println(word);
}
```

Exercise: Finding "S" Hooks

- Another important strategic principle in Scrabble is to conserve your S tiles so that you can hook longer words (ideally, the high-scoring seven-letter plays called ***bingos***) onto existing words.
- Some years ago, I was in a hotel where the shower taps were prominently labeled with HOT and COLD:



- Being a Scrabble player, it immediately occurred to me that each of these words takes an S on *either* end, making them ideally flexible for Scrabble plays.
- Write a Java program that finds all such words.

Selected Methods in the `Map` Interface

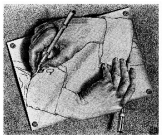
`map.size()`	Returns the number of key/value pairs in the map.
`map.isEmpty()`	Returns `true` if the map is empty.
`map.put(key, value)`	Makes an association between `key` and `value`, discarding any existing one.
`map.get(key)`	Returns the most recent value associated with `key`, or `null` if there isn't one.
`map.containsKey(key)`	Returns `true` if there is a value associated with `key`.
`map.remove(key)`	Removes `key` from the map along with its associated value, if any.
`map.clear()`	Removes all key/value pairs from the map.
`map.keySet()`	Returns a set of all the keys in the map, which you can then use in a `for` loop.

Mapping Color Names to Colors

- The following code creates a `Map<String,Color>` that translates a color name into the appropriate Java color value:

```
Map<String,Color> map = new TreeMap<String,Color>();
map.put("BLACK", Color.BLACK);
map.put("BLUE", Color.BLUE);
map.put("CYAN", Color.CYAN);
map.put("DARK_GRAY", Color.DARK_GRAY);
map.put("GRAY", Color.GRAY);
map.put("GREEN", Color.GREEN);
map.put("LIGHT_GRAY", Color.LIGHT_GRAY);
map.put("MAGENTA", Color.MAGENTA);
map.put("ORANGE", Color.ORANGE);
map.put("PINK", Color.PINK);
map.put("RED", Color.RED);
map.put("WHITE", Color.WHITE);
map.put("YELLOW", Color.YELLOW);
```

Graphics Contest



The CS106A
Graphics Contest
Hall of Fame

Past Winners