Lecture April 9th - Immutable and Advanced Topics

Bentley James Oakes

April 8, 2018

Schedule

- Mutable versus Immutable
- Copying References
- Storing Objects
- HashMaps
- Case Studies
- Final Review
- Programming Possibilities

This Lecture

1 Recap

2 Mutable versus Immutable

3 Copying References

4 Storing Objects

5 HashMaps

Section 1

Recap

The Finally Block

- Let's see the *finally* block, just to finish off our discussion of `Exceptions`
- The finally block is attached to a try block, and it always executes.
- Even if one of the following happens:
  - an unexpected exception occurs in the try block
  - an exception occurs in the catch block
  - There's a return/continue/break statement in the try/catch block.
- You can have a finally even with just a try block (and no catch).
- The finally block is useful for cleaning up file IO operations, where the file needs to be closed

IOException and FileNotFoundException

- File IO operations are so error prone that the code containing them can throw an `IOException`

From the `FileReader` object:

Constructor Detail

FileReader

```
public FileReader(String fileName)
           throws FileNotFoundException
```

Creates a new `FileReader`, given the name of the file to read from.

Parameters:

`fileName` - the name of the file to read from

Throws:

`FileNotFoundException` - if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

File Reading With Throws

```
public static void main(String[] args){
    try{
        readFile("abc.txt");
    }catch(FileNotFoundException e){
        System.out.println("The file was not found.");
    }catch(IOException e){
        System.out.println("Error with the file.");
    }
}
public static void readFile(String filename)
    throws FileNotFoundException, IOException{
    FileReader fr = null;
    BufferedReader br = null;
    try{
        fr = new FileReader(filename);
        br = new BufferedReader(fr);
        String s = br.readLine();
        while (s != null){
            System.out.println(s);
            s = br.readLine();
        }
    }finally{
        if (fr != null){
            fr.close();
        }
        if (br != null){
            br.close();
        }
    }
}
```

Wrapper Classes

- What about storing ints and doubles in `ArrayLists`?
- We can't, because they are primitive types, and `ArrayLists` only store reference types/`Objects`

Boxing

- Java does some magic behind the scenes to make these wrapper classes mostly act like the primitive types
- This is called *boxing* and *unboxing*

```
Double g = new Double(8.4);
Double h = 5.5; //auto-boxing

double k = g;//auto un-boxing
System.out.println(g + " " + h + " " + k);
//8.4 5.5 8.4
```

Wrapper Class Overview

- Used to *wrap* primitive values in a class
- Only use wrapper classes if you have to place values in an `ArrayList`

Section 2

Mutable versus Immutable

Mutable versus Immutable

- We might have classes where the values change, and where they don't change
- Mutable: Values in a class can change
- Immutable: Values can't change

Mutable Class

- A *mutable* class means that values can change
- For example, in the `Student` class, we can change the grade

```
Student s = new Student("Bentley", 87);

System.out.println(s); //Bentley Grade: 87

s.setGrade(100);

System.out.println(s); //Bentley Grade: 100
```

Immutable

- *Immutable* classes mean that their values can't change
- An example is the wrapper classes
- There is no setter to change the value stored
- If you want a `Double` instance with a different value, you must create a new instance

```
Double e = new Double(3.33);
System.out.println("E: " + e);//E: 3.33

double dbl = e.doubleValue();
dbl = dbl * 3;

e = new Double(dbl);
System.out.println("E: " + e);//E: 9.99
```

Writing an Immutable Class

- We can write our own immutable classes
- Just make the variables private, and don't provide setter methods
- Only way to set attribute values is through the constructor

```
public class MyDate{
    private int day;
    private int month;
    private int year;

    public MyDate(int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public String toString(){
        return year + "/" + month + "/" + day;
    }

    public static void main(String[] args){
       MyDate e = new MyDate(1972, 06, 03);
       System.out.println(e); //1972/6/3
    }
}
```

Why Use Immutable?

- If we have an instance of `MyDate`, then we know that another part of our program can't accidentally change the information
- Could be useful in a database program, where transactions shouldn't be deleted or modified

```
public class MyDate{
    private int day;
    private int month;
    private int year;

    public MyDate(int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public String toString(){
        return year + "/" + month + "/" + day;
    }

    public static void main(String[] args){
       MyDate e = new MyDate(1972, 06, 03);
       System.out.println(e); //1972/6/3
    }
}
```

Section 3

Copying References

Copying References

- Let's make sure that values are protected in our immutable classes
- Problems can occur with arrays and `ArrayLists`
- The issue is that other classes have the address of our private array/`ArrayList`

Immutable Student Class

- Let's have an *immutable* `Student` class
- Let's try to protect the list of courses the `Student is taking`

```
import java.util.ArrayList;

public class Student{

    private String name;
    private ArrayList<String> courses;

    public Student(String name, ArrayList<String> courses){
        this.name = name;
        this.courses = courses;
    }

    public String toString(){
        String s = this.name + " Courses: \n";
        s += courses;
        return s;
    }

}
```

In another class:

```
ArrayList<String> courses = new ArrayList<String>();

courses.add("Potions");
courses.add("Defense Against the Dark Arts");
courses.add("Charms");

Student s = new Student("Harry Potter", courses);

System.out.println(s);
//Harry Potter Courses:
//[Potions, Defense Against the Dark Arts, Charms]

courses.clear();

System.out.println(s);
//Harry Potter Courses:
//[]
```

The other class has access to the list, so it can change the data

Immutable Student Class

- Let's change the constructor of `Student` to copy the courses to the attribute

```java
import java.util.ArrayList;

public class Student{

    private String name;
    private ArrayList<String> courses;

    public Student(String name, ArrayList<String> courses){
        this.name = name;
        this.courses = new ArrayList<String>();

        for (int i=0; i < courses.size(); i++){
            this.courses.add(courses.get(i));
        }
    }

    public String toString(){
        String s = this.name + " Courses: \n";
        s += courses;
        return s;
    }
}
```

In another class:

```java
ArrayList<String> courses = new ArrayList<String>();

courses.add("Potions");
courses.add("Defense Against the Dark Arts");
courses.add("Charms");

Student s = new Student("Harry Potter", courses);

System.out.println(s);
//Harry Potter Courses:
//[Potions, Defense Against the Dark Arts, Charms]

courses.clear();

System.out.println(s);
//Harry Potter Courses:
//[Potions, Defense Against the Dark Arts, Charms]
```

Now there are two `ArrayLists`, so a change in one doesn't affect the other

In assignment 5, `setPlanets` must copy elements from one list to the other

Copying References in Getter

- Make sure that when a `private ArrayList` is returned, a copy is made
- Otherwise, another class can obtain the `ArrayList` and change the data
- This code creates a new `ArrayList`, and copies the elements into the new list

```
public ArrayList<String> getCourses(){

    ArrayList<String> newCourses = new ArrayList<String>();
    for (int i=0; i < this.courses.size(); i++){
         newCourses.add(this.courses.get(i));
    }
    return newCourses;
}
```

Section 4

Storing Objects

Creating ArrayLists

- We saw how to create `ArrayLists` to store various types
- `ArrayList<Integer> myList = new ArrayList<Integer>();`
- `ArrayList<String> otherList = new ArrayList<String>();`
- `ArrayList<Book> books = new ArrayList<Book>();`
- `ArrayList<Double> numbers = new ArrayList<Double>();`

What if we want to store both `Strings` and `Integers` in an `ArrayList`?

Storing Objects in an ArrayList

To store multiple object types in an `ArrayList`, we write
`ArrayList<Object> list = new ArrayList<Object>();`
Note that we can't store primitive types, only reference types!

```
ArrayList<Object> list = new ArrayList<Object>();

list.add("hello");
list.add(new Double(4.5));
list.add(new Integer(99));

System.out.println(list);
//[hello, 4.5, 99]
```

Object Inheritance

- We've talked about *objects* in Java, but there's some missing background material
- We've skipped over *inheritance*, which means that all reference types (like `Strings` and `Students`) have some default methods
  - `toString` and `equals`
- We say that all classes *inherit* these methods from the parent class `Object`

Why is This Important?

- This is important because all reference types are guaranteed to have an `equals` method and a `toString` method.
- For example, the `toString` method of an `ArrayList` can just call `toString` on all the elements
- All reference types have a `toString` method, so this will always work

Contains Method

- The `contains` method on `ArrayLists` searches for an element
  - `contains(Object o)` ← returns *true* if o is in the list and *false* otherwise
- The `contains` method will call the `equals` method on each element to see if the element is the same as o
  - Reference types have a built-in `equals` method, but you should write your own
  - This is outside the scope of COMP 202

Section 5

HashMaps

Dictionary Example

- Let's write a program that is a dictionary
- For each word in our dictionary, we have a definition
- For example, the entry for *banana* could be *a yellow fruit*

- We could do this with a new class, but we just need something simpler
- We need a mapping from a `String` to a `String`
- From *banana* to *a yellow fruit*

HashMaps

We'll implement this dictionary using a `HashMap`

HashMaps in Java are a data structure providing an unordered collection of values - indexed by a key whose type is whatever you want

For example, we could have someone's name mapped to their phone number:

```
HashMap<String, Integer> numbers =
new HashMap<String, Integer>();
```

For the dictionary, we will have a `String` mapped to a `String`
Note that we can only store reference types in a `HashMap`

Terminology

If we have a mapping from a word to the definition:
`HashMap<String, String> dictionary = new HashMap<String, String>();`
The word is called the *key*, and the mapping provides us with a *value* for that key
We say that the `HashMap` stores key-value pairs

HashMap Example

Basic `HashMap` methods:

- `put(Object key, Object value)` ← adds a value to the `HashMap` and associates it with that key
- `get(Object key)` ← returns the value associated with that key
- Note: the types must match what the `HashMap` expects

```
import java.util.HashMap;

public class HashMapExample{
    public static void main(String[] args){
        HashMap<String, String> dictionary = new HashMap<String, String>();

        dictionary.put("banana", "a yellow fruit");
        dictionary.put("apple", "a red fruit");

        String def = dictionary.get("banana");
        System.out.println("Entry for banana: " + def);
        //Entry for banana: a yellow fruit
    }
}
```

Other Methods

- `containsKey(Object key)` ← returns whether or not the `HashMap` contains the key
- `containsValue(Object value)` ← returns whether or not the `HashMap` contains the value
- `remove(Object key)` ← removes the key-value pair from the map
- `size()` ← returns the number of key-value pairs in the `HashMap`
- `keySet()` ← Returns a list of the keys contained in this map

Ordering

Note: There is no ordering in a `HashMap`.
This means that keys might not be sorted alphabetically, and not by the order you inserted them in.

Printing a HashMap

We can use the `toString` method for a `HashMap` (but it's ugly):

```
HashMap<String, String> dictionary = new HashMap<String, String>();

dictionary.put("banana", "a yellow fruit");
dictionary.put("apple", "a red fruit");

//overwrite the entry for banana
dictionary.put("banana", "a non-apple fruit");

dictionary.put("watermelon", "round fruit");
dictionary.put("tomato", "a fruit(?)");

System.out.println(dictionary);
//{banana=a non-apple fruit, apple=a red fruit, watermelon=round fruit, tomato=a fruit(?)}
```

Printing a HashMap

It's nicer to use a `for-each` loop to print a `HashMap`:

```
HashMap<String, String> dictionary = new HashMap<String, String>();

dictionary.put("banana", "a yellow fruit");
dictionary.put("apple", "a red fruit");

//overwrite the entry for banana
dictionary.put("banana", "a non-apple fruit");

dictionary.put("watermelon", "round fruit");
dictionary.put("tomato", "a fruit(?)");

for(String key: dictionary.keySet()){
    String value = dictionary.get(key);
    System.out.println(key + " - " + value);
}
//banana - a non-apple fruit
//apple - a red fruit
//watermelon - round fruit
//tomato - a fruit(?)
```

Counting Names

Write a method that takes as input an array of `Strings` and counts how many times each `String` appears in the array. Do this using a `HashMap`.

getCount

```
public static HashMap<String, Integer> getCount(String[] arr){
    HashMap<String, Integer> nameCountMap = new HashMap<String, Integer>();

    for (int i=0; i < arr.length; i++){
        //get the String from the array
        String s = arr[i];

        //if the map doesn't contain the name
        if (!nameCountMap.containsKey(s)){
            //record that we saw the name once
            nameCountMap.put(s, new Integer(1));

        }else{ //map contains the name
            //get how many times we saw the name
            Integer count = nameCountMap.get(s);
            //add one to the count
            Integer newCount = count + 1;
            //store the new count
            nameCountMap.put(s, newCount);
        }

    }
    return nameCountMap;
}
```

getCount Main

```
public static void main(String[] args){

    String[] arr = {"Bentley", "Bentley", "Bentley",
        "Melanie", "Bentley", "Giulia", "Melanie",
        "Giulia", "Bentley", "Bentley"};

    HashMap<String, Integer> nameCount = getCount(arr);

    for(String key: nameCount.keySet()){
        Integer value = nameCount.get(key);
        System.out.println(key + " - " + value);
    }
    //Melanie - 2
    //Giulia - 2
    //Bentley - 6
}
```