CSCI 136
Data Structures & Advanced Programming

Lecture 5

Fall 2018

Bill Lenhart & Bill Jannon

Administrative Details

- Read and prepare for Lab 2
  - Bring a design document!
  - We'll collect them
  - We'll also hand out one of our own for comparison

Last Time

- String Manipulation Example: XML parsing
- More on Java Program Organization
  - Enums
  - Interfaces
  - Multiple implementations of an interface

Today

- Miscellaneous Java
  - modifiers for variables and methods
  - Variable storage and memory management
- The class Object
  - Provides default toString() and equals() methods
- Card Deck: Array and Vector versions
- Associations and Vectors
- Code Samples
  - WordFreq (Vectors, Associations, histograms)
  - Dictionary (Associations, Vectors)

Access Levels

- public, private, and protected variables/methods
- What's the difference?
  - public – accessible by all classes, packages, subclasses, etc.
  - protected – accessible by all objects in same class, same package, and all subclasses
  - private – only accessible by objects in same class
- Generally want to be as "strict" as possible

Access Modifiers

	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
none	Y	Y	N	N
private	Y	N	N	N

A package is a named collection of classes.

- Structure5 is Duane's package of data structures
- Java.util is the package containing Random, Scanner and other useful classes
- There's a single "unnamed" package

About Static Variables

- Static variables are shared by all instances of class
- What would this print?

```
public class A {
     static protected int x = 0;
     public A() {
         x++;
         System.out.println(x);
     }
     public static void main(String args[]) {
         A a1 = new A();
         A a2 = new A();
     }
}
```

- Since static variables are shared by all instances of A, it prints 1 then 2. (Without `static`, it would print 1 then 1.

About Static Methods

- Static methods are shared by all instances of class
  - Can only access static variables and other static methods

```
public class A {
     public A() { … }
     public static int tryMe() { … }
     public int doSomething() { … }
     public static void main(String args[]) {
            A a1 = new A();
            int n = a1.doSomething();
            A.doSomthing(); //WILL NOT COMPILE
            A.tryMe();
            a1.tryMe();      // LEGAL, BUT MISLEADING!
            doSomething();   // WILL NOT COMPILE
            tryMe();         // Ok
     }
}
```

Memory Management in Java

- Where do "old" cards go?

```
Card c = new Card(ACE, SPACES);
…
c = new Card (ACE, DIAMONDS);
```

- What happens to the Ace of Spades?
- Java has a *garbage collector*
  - Runs periodically to "clean up" memory that had been allocated but is no longer in use
  - Automatically runs in background
- Not true for many other languages!

Variables and Memory

- Instance variables
  - Upon declaration are given a default value
  - Primitive types
    - 0 for number types, false for Boolean, \u000 for char
  - Class types and arrays: null
- Local variables
  - Are NOT given a default when declared
- Method parameters
  - Receive values from arguments in method call

Types and Memory

- Variables of primitive types
  - Hold a value of primitive type
- Variables of class types
  - Hold a *reference* to the location in memory where the corresponding object is stored
- Variable of array type
  - Holds a *reference*, like variables of class type
- Assignment statements
  - For primitive types, copies the value
  - For class types, copies the reference

Class Object

- At the root of all class-based types is the type `Object`
- All class types implicitly *extend* class `Object`
  - Card52, Student, … `extend Object`

```
Object ob = new Card52(); // legal!
Card52 c = new Object(); // NOT legal!
```

- Class `Object` defines some methods that all classes should support, including

```
public String toString()
public boolean equals(Object other)
```

- But we usually *override* (redefine) these methods
  - As we did with toString() in the various CardXYZ classes
  - What about equals()?

Object Equality

- Suppose we have the following code:

```
Card c1 = new CardRankSuit(Rank.ACE, Suit.SPADES);
Card c2 = new CardRankSuit(Rank.ACE, Suit.SPADES);
if (c1 == c2) { System.out.println("SAME"); }
else { System.out.println("Not SAME"); }
```

- What is printed?
- How about:

```
Card c3 = c2;
if (c2 == c3) { System.out.println("SAME"); }
else { System.out.println("Not SAME"); }
```

- '==' tests whether 2 names refer to same object
  - Each time we use "new" a new object is created

Equality

- What do we really want?
  - Check both rank and suit!
- How?

```
if (c1.getRank() == c2.getRank() && c1.getSuit() == c2.getSuit()){
    System.out.println("SAME");
}
```

- This works, but is cumbersome…
- `equals()` to the rescue....

equals()

- We use:

```
if (c1.equals(c2)) { … }
```

- We can define equals() for each CardXYZ class

```
public boolean equals(Object other) {
  if ( other instanceof Card ) {
      Card oc = (Card) other;
      return this.getRank() == oc.getRank() &&
          this.getSuit() == oc.getSuit();
  }
  else
      return false;
}
```

- Note: Must cast `other` to type `Card`

Array Manipulation: Shuffling

- How would we shuffle our deck of cards?
- We could write `shuffleDeck()`
  - Many ways to implement.
  - An efficient way
    - Randomly move cards to "tail" of deck
    - Do this by swapping random card with card from tail
- swap is a little tricky
  - Three step process, not two!

Vector: A Flexible Array

A Limitation of Arrays

- Must decide size when array is created
- What if we fill it and need more space?
  - Must create new, larger array
  - Must copy elements from old to new array

Enter the Vector class

- Provides functionality of array
  - Sadly, can't use [] syntax…
- Automatically grows as needed
- Can hold values of any class-based type
  - Not primitive types---but there's a work-around

Example: Vector-Based Card Deck

- A Vector holds the cards

```
cards = new Vector();
```

- Cards are added one by one to Vector

```
cards.add( new Card52v2( r, s ) );
```

- Swap uses the Vector's get and set methods

```
Card toMove = (Card) cards.get(i);
cards.set( i, cards.get( remaining-1 ) );
cards.set( remaining-1, toMove );
```

- Note: Constant NUMCARDS not needed!
- Note: A Vector can hold any Object
- Note: Must include structure package

```
import structure.*;
```

Vectors

- Vectors are collections of Objects
- Methods include:
  - `add(Object o), remove(Object o)`
  - `contains(Object o)`
  - `indexOf(Object o)`
  - `get(int index), set(int index, Object o)`
  - `remove(int index)`
  - `add(int index, Object o)`
  - `size(), isEmpty()`
- Remove methods preserve order, close “gap”

Example: Word Counts

- Goal: Determine word frequencies in files
- Idea: Keep a Vector of (word, freq) pairs
  - When a word is read…
  - If it's not in the Vector, add it with freq = 1
  - If it is in the Vector, increment its frequency
- How do we store a (word, freq) pair?
  - An *Association*

Associations

- Word → Definition
- Account number → Balance
- Student name → Grades
- Google:
  - URL → page.html
  - page.html → {a.html, b.html, …} (links in page)
  - Word → {a.html, d.html, …} (pages with Word)
- In general:
  - Key → Value

Association Class

- We want to capture the "key → value" relationship in a general class that we can use everywhere
- What type do we use for key and value instance variables?
  - Object!
  - We can treat any class as an Object since all classes inherently extend Object class in Java…

Association Class

```
// Association is part of the structure package
class Association {
   protected Object key;
   protected Object value;

   //pre: key != null
   public Association (Object K, Object V) {
        Assert.pre (K!=null, “Null key”);
        key = K;
        value = V;
   }

   public Object getKey() {return key;}
   public Object getValue() {return value;}
   public Object setValue(Object V) {
        Object old = value;
        value = V;
        return old;
   }
}
```

WordFreq.java

- Uses a Vector
  - Each entry is an Association
  - Each Association is a (String, Integer) pair
- Notes:
  - Include structure.*;
  - Can create a Vector with an initial capacity
  - Must *cast* the Objects removed from Association and Vector to correct type before using

Notes About Vectors

- Primitive Types and Vectors

```
Vector v = new Vector();
v.add(5);
```

  - This (technically) shouldn't work! Can't use primitive data types with vectors…they aren't Objects!
  - Java is now smart about some data types, and converts them automatically for us -- called *autoboxing*
- We used to have to "box" and "unbox" primitive data types:

```
Integer num = new Integer(5);
v.add(num);
…
Integer result = (Integer)v.get(0);
int res = result.intValue();
```

- Similar wrapper classes (Double, Boolean, Character) exist for all primitives

Vector Summary So Far

- Vectors: "extensible arrays" that automatically manage adding elements, removing elements, etc.
  1. Must cast Objects to correct type when removing from Vector
  2. Use wrapper classes (with capital letters) for primitive data types (use "Integers" not "ints")
  3. Define equals() method for Objects being stored for contains(), indexOf(), etc. to work correctly

Application: Dictionary Class

- What is a Dictionary
  - Really just a *map* from words to definitions…
  - We can represent them with Associations
  - Given a word, lookup and return definition
  - Example: java Dictionary some_word
    - Prints definition of some_word
- What do we need to write a Dictionary class?
  - A Vector of Associations of (String, String)

Dictionary.java

```
protected Vector defs;
public Dictionary() {
   defs = new Vector();
}

public void addWord(String word, String def) {
   defs.add(new Association(word, def));
}

// post: returns the definition of word, or "" if not found.
public String lookup(String word) {
   for (int i = 0; i < defs.size(); i++) {
       Association a = (Association)defs.get(i);
       if (a.getKey().equals(word)) {
           return (String)a.getValue();
       }
   }
   return "";
}
```

Dictionary.java

```
public static void main(String args[]) {
   Dictionary dict = new Dictionary();
   dict.addWord("perception", "Awareness of an object of
        thought");
   dict.addWord("person", "An individual capable of moral
        agency");
   dict.addWord("pessimism", "Belief that things generally
        happen for the worst");
   dict.addWord("philosophy", "Literally, love of
        wisdom.");
   dict.addWord("premise", "A statement whose truth is used to
        infer that of others");
}
```

Lab 2

- Three classes:
  - FrequencyList.java
  - Table.java
  - WordGen.java
- Two Vectors of Associations
- toString() in Table and FrequencyList for debugging
- What are the key stages of execution?
  - Test code thoroughly before moving on to next stage
- Use WordFreq as example

Lab 2: Core Tasks

- FreqencyList
  - A Vector of Associations of String and Int
  - Add a letter
    - Is it a new letter or not?
    - Use indexOf for Vector class
- Pick a random letter based on frequencies
  - Let total = sum of frequencies in FL
  - generate random int r in range [0…total]
  - Find smallest k s.t r >= sum of first k frequencies

Lab 2: Core Tasks

- Table
  - A Vector of Associations of String and FrequencyList
  - Add a letter to a k-gram
    - Is it a new k-gram or not?
  - Pick a random letter given a k-gram
    - Find the k-gram then ask its FrequencyList to pick
- WordGen
- Convert input into (very long) String
  - Use a StringBuffer---see handout