CSCI 136
Data Structures & Advanced Programming

Lecture 5

Fall 2019

Bill & Sam

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

Organization

- Before: Java review
- This week: using multiple data structures together

Last Time

- String Manipulation Example: XML parsing
- More on Java Program Organization

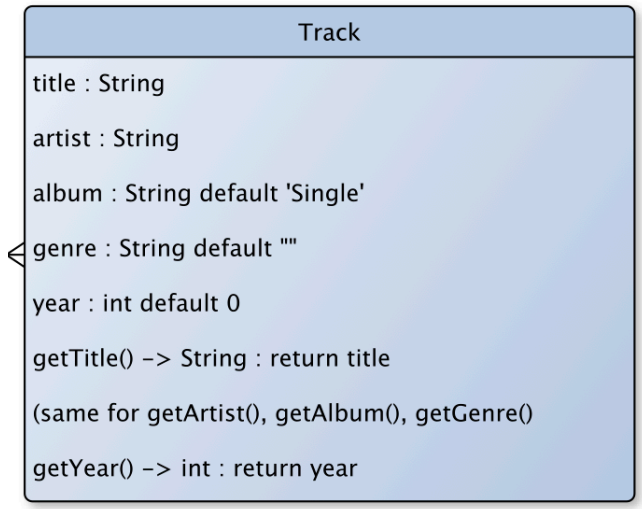
Today

- Finish up some discussion on objects
  - Formalize some of the issues we've been having: how does Java handle memory?
- Vectors
- Code Samples
  - WordFreq (Vectors, Associations, histograms)
  - Dictionary (Associations, Vectors)

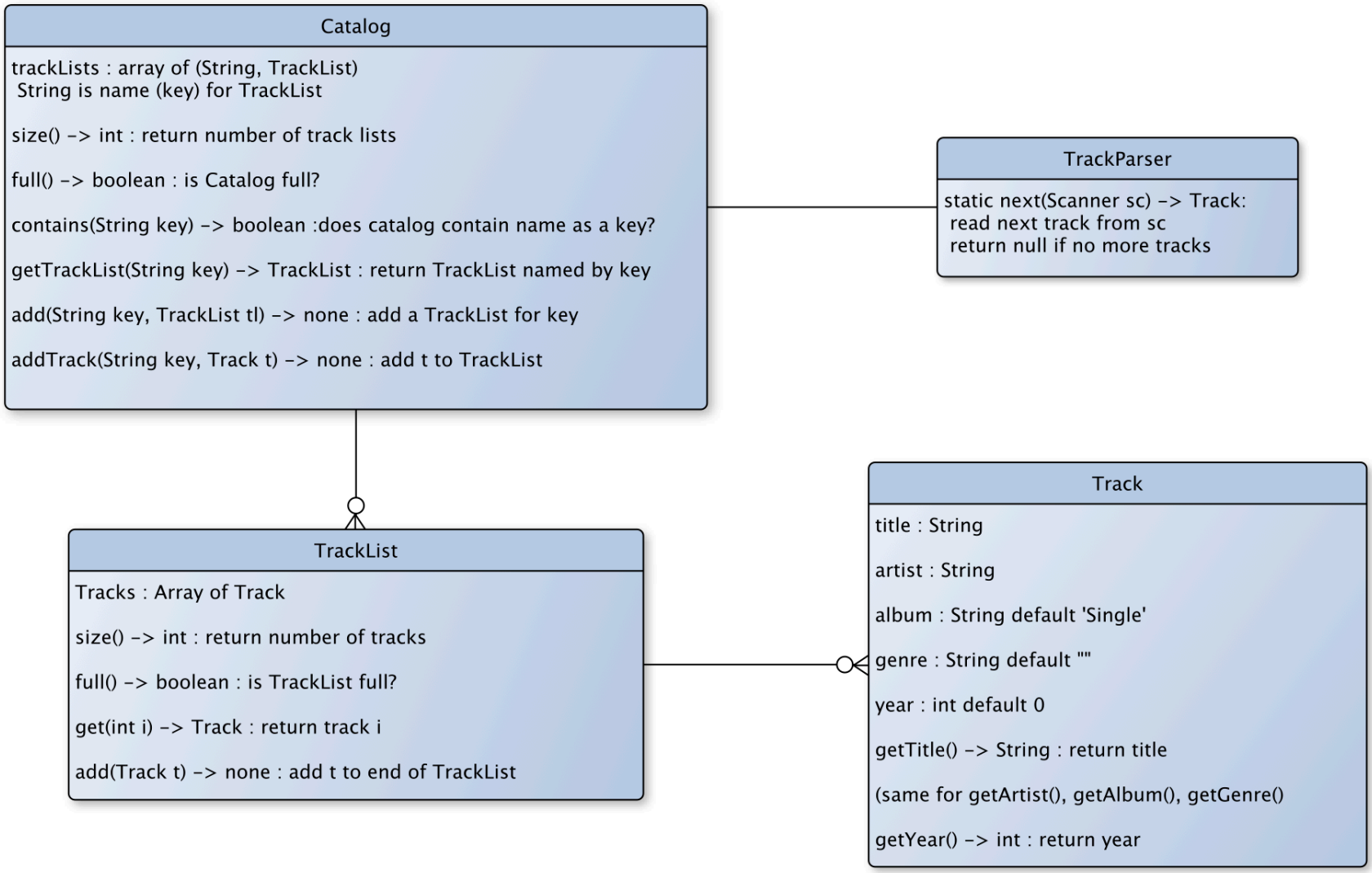
Catalog: Classes

- Track
  - Store data about a single music track
  - Allow access (not updating) to that data
- TrackList
  - Store a set of tracks
  - Allow access to i^{th} track, add new tracks
- Catalog
  - Store a set of ***named*** track lists
  - Allow access to track list by name, add a track list, add a track
- TrackParser : utilities to parse XML track file

Catalog: Class Diagram



Catalog: Class Diagram



Implementation Notes

- Track
  - Object data is private, methods are public
  - Use of "this" to disambiguate names
  - Special methods: constructors and toString
- TrackList
  - DEFAULT_SIZE
    - final : a constant—value can't be changed
    - static : one copy of variable is shared among all Tracks
  - Array capacity (length) not same as current size
    - contains & toString need to use current size
  - Contains uses a problematic equality test!

Implementation Notes

- Catalog
  - Use an Association to pair name with TrackList
    - Stores a pair of objects as a (key, value) pair
    - Supports getKey() and getValue() methods
    - But these methods return type Object
      - Must *cast* the type back to actual type
      - Use instance of method to check for correct type in equals()
- TrackParser
  - Assumes one XML tag per line
  - Minimal error-checking
  - Uses private parseLine() method for modularity
  - Uses *switch* statement on tag

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/array types, copies the reference

Types and Memory: Copying

```
int a = 20;
int b = a;
a++;
System.out.println(b);

Student s1 = new Student("Mary",20,'A');
Student s2 = s1;
s1.setGrade('B');
System.out.println(s2.getGrade());
```

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

Memory Management in Java

- Where do "old" objects go?

```
Track t = new Track("Hey, Jude", "The Beatles", … );
…
t = new Track ("Blowin' in the Wind", "Bob Dylan", … );
```

- What happens to Hey, Jude?
- 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!

Class Object

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

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

  - Student, Track, TrackList are *subclasses* of type Object
- 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 our previous examples
  - Let's override equals() for the Track class….

Object Equality

- Suppose we have the following code:

```
Track t1 = new Track("A song", "An Artist", "An Album");
Track t2 = new Track("A song", "An Artist", "An Album");
if (t1 == t2) { System.out.println("SAME"); }
else { System.out.println("Not SAME"); }
```

- What is printed?
- How about:

```
Track t3 = t2;
if (t2 == t3) { 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?
  - Ideally, all fields should be the same
  - But sometimes genre/year is missing, so skip them
- How?

```
if (t1.getTitle().equals(t2.getTitle()) &&
    t1.getArtist().equals(t2.getArtist()) &&
    t1.getAlbum().equals(t2.getAlbum()) ) {

   System.out.println("SAME");
}
```

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

equals()

- We use:

```
if (t1.equals(t2)) { … }
```

- We can define equals() for our Track class

```
public boolean equals(Object other) {
   if ( other instanceof Track ) {
      Track ot = (Track) other;
      return getTitle().equals(ot.getTitle()) &&
         getArtist().equals(ot.getArtist()) &&
         getAlbum().equals(ot.getAlbum())          }
   else
      return false;
  }
```

- Notes
  - Must cast `other` to type `Track`
  - Should add toLower() for upper/lower-case mismatches!

Multi-Dimensional Arrays

- Syntax for 1-D array:

```
Card deck[] = new Card[52]; // array of 52 “nulls”
Card[] deck= new Card[52]; // same
```

- Syntax for 2-D array:

```
int [][] grades = new int[10][15];
String[][] deck = new String[4][13];
String[][] wordLists = new String[26][ ]
```

- Determine size of array?

```
deck.length;  //not deck.length()!!
wordLists.length vs wordLists[3].length?
```

About "static" Variables

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

```
public class A {
     static private 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 do not access/mutate objects 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
     }
}
```

When to Use `static`

For class X having instance variable v and method m()

- Instance variable v
  - If distinct objects of the class might have different values for v, v *cannot* be declared static
  - If all objects of the class will always have the same value for v, v *should be* declared static
    - In particular, constants should be made static
- Method m()
  - If you want to be able to invoke m() without needing an object of class X, m() *must be* declared static
  - If you m() to be able to access/update instance variables of objects of class X, m() *cannot be* declared static

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

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

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*

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

Implementing Vectors

- A Vector holds an array of Objects
- Key difference is that the number of elements can grow and shrink dynamically
- How are they implemented in Java?
  - What instance variables do we need?
  - What methods? (start simple)
- Let's explore the implementation….

Class Vector : Instance Variables

```
public class Vector<E> {
private Object[] elementData;     // Underlying array
protected int elementCount;       // Number of elts in Vector
protected final static int defaultCapacity;
protected int capacityIncrement;  // How much to grow by
protected E initialValue;         // A default elt value
}
```

- Why Object[]?
  - Java restriction: Can't use type variable, only actual type
- Why elementCount?
  - size won't usually equal capacity
- Why capacityIncrement?
  - We'll "grow" the array as needed

Core Vector Methods

```
public class Vector {
    public Vector()  // Make a small Vector

    // Make Vector of given capacity
    public Vector(int initCap)

    // Add elt to (high) end of Vector
    public void add(Object elt)

    // Add elt at position I
    public void add(int i, Object elt)

    // Remove (and return) elt
    public Object remove(Object elt)

    // Remove (and return) elt at pos I
     public E remove(int i) //
```

Core Vector Methods

```
   public int capacity()   // Return capacity
   public int size()       // Return current size
   public boolean isEmpty()// Is size == 0?

   // Is elt in Vector?
   public boolean contains(Object elt)

   // Return elt at position I
   public Object get(int i)

   // Change value at position I
   public Object set(int i, Object elt)

   // Return earliest position of elt
   public int indexOf(Object elt)
}
```

Class Vector : Basic Methods

- Much work done by few methods:
  - indexOf(Object elt, int i)
    - Find first occurrance of elt at/after pos. I
    - Used by indexOf(Object elt)
    - remove methods use indexOf(Object elt)
  - firstElement(), lastElement() use get(int i)
- Method names/functions in spirit of Java classes
  - indexOf has same behavior as for Strings
- Methods are straightforward except when array is full
- How do we add to a full Vector?
  - We make a new, larger array and copy values to it

Extending the Array

- How should we extend the array?
- Possible extension methods:
  - Grow by fixed amount when capacity is reached
  - Double array when capacity is reached
- How could we compare the two techniques?
  - Run speed tests?
    - Hardware/system dependent
  - Count operations!
  - We'll do this soon

ensureCapacity

- How to implement `ensureCapacity(int minCapacity)`?

```
// post: the capacity of this vector is at least minCapacity
public void ensureCapacity(int minCapacity) {
   if (elementData.length < minCapacity) {
      int newLength = elementData.length; // initial guess
      if (capacityIncrement == 0) {
      // increment of 0 suggests doubling (default)
         if (newLength == 0) newLength = 1;
            while (newLength < minCapacity) {
              newLength *= 2;
            }
       } else {
      // increment != 0 suggests incremental increase
         while (newLength < minCapacity) {
            newLength += capacityIncrement;
         }
      }
```

```
  // assertion: newLength > elementData.length.
     Object newElementData[] = new Object[newLength];
     int i;

  // copy old data to array
     for (i = 0; i < elementCount; i++) {
        newElementData[i] = elementData[i];
     }

     elementData = newElementData;
        // garbage collector will pick up old elementData
  }
  // assertion: capacity is at least minCapacity
}
```

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
  - Each has a valueOf() method to return primitive

Vector Summary & Notes

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");
}
```

Example: Randomizing a Vector

- How would we shuffle the elements of a Vector?
- `shuffle(Vector v)`
  - Many ways to implement.
  - An efficient way
    - Randomly move elements to "tail" of vector
    - Do this by swapping random element with last element
- swap is a little tricky
  - Three step process, not two!

Lab 2 Preview

- 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