CSCI 136
Data Structures & Advanced Programming

Lecture 4

Fall 2019

Instructors: Bill & Sam

Last Time

- Control structures
  - Branching: if – else, switch, break, continue
  - Looping: while, do – while, for, for – each
- Object oriented programming Basics (OOP)
- Strings and String methods

Today's Outline

- String Processing Example: XML
- More on Class Types
  - Music Catalog: A multi-class example
    - Using: Associations
- Technique: Randomizing an array
- Miscellaneous Java
  - Static variables and methods
  - Memory management
  - Access control: public, protected, private

Using Strings

- Application: Parsing an XML file of a CD collection
  - XML = eXtensible Markup Language
  - XML is used for many things
  - Music track info:

```
<TRACK>
   <NAME>Big Willie style</NAME>
   <ARTIST>Will Smith</ARTIST>
   <ALBUM>Big Willie style</ALBUM>
   <GENRE>Pop Rap</GENRE>
   <YEAR>1997</YEAR>
 </TRACK>
```

- How can we find and print just the track names?
  - See TrackTitles.java
  - java TrackTitles < trackList.xml

Catalog: An Extended Example

- Design a program to manage a collection of music tracks, supporting
  - Track objects
  - Collections of tracks (playlist)
  - Collections of playlists (music catalog)
  - Importing of track data from a .XML file
- Goals
  - Better understand basic OOP concepts in Java
  - Foreshadow concepts to come in future lectures
- But first, we'll need a tool: Associations

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 thing as an Object since all classes inherently extend Object class in Java…

Association Class

Association Methods

- public Association (Object key, Object value)
- public Object getKey() : return key
- public Object getValue() : return value
- public Object setValue(Object v)
- public boolean equals(Object other)
  - Return true if keys match; return false otherwise

Example: A Simple Dictionary

```
class Dictionary {

   // A method to print the defs of words from command line.

   public static void main(String args[]) {
       Dictionary dict = new Dictionary();
       System.out.println();

       for (int i = 0; i < args.length; i++) {
           String answer = dict.lookup(args[i]);

           if (!answer.equals(""))
               System.out.println(args[i] + ": " + answer);
           else
               System.out.println("The word '" + args[i] +
                "' was not found.");
       }
       System.out.println();
   }

// implementation continued on next slides…
```

Example: A Simple Dictionary

```
protected Association words[] = new Association[5];

public Dictionary() {
    words[0] = new Association("perception",
        "Awareness of an object of thought");

    words[1] = new Association("person",
        "An individual capable of moral agency");

    words[2] = new Association("pessimism",
        "Belief that things happen for the worst");

    words[3] = new Association("philosophy",
        "Literally, love of wisdom.");

    words[4] = new Association("premise",
        "A statement used to infer truth of others");
}

// implementation continued on next slide…
```

Example: A Simple Dictionary

```
// post: returns the definition of word, or "" if not found.

   public String lookup(String word) {

   // Note: If words array is not "full", this method would crash
   // If a word wasn't found (Why?)

       for (int i = 0; i < words.length; i++) {

           Association a = words[i];

           if (a.getKey().equals(word)) {
               return (String)a.getValue();
               // note the type-cast above to recover type
           }
       }
       return "";
   }

} // End of class declaration
```

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;
   }
// Continued on next slide….
```

Association Class

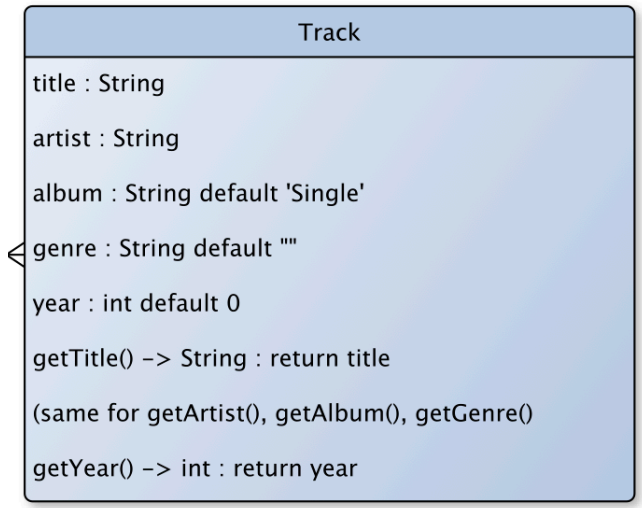
```
    public boolean equals(Object other) {
         if ( other instanceof Association ) {
                Association otherAssoc = (Association)other;
                return getKey().equals(otherAssoc.getKey());
         }
         else return false;
    }
}
```

- Note: The actual structure package code does NOT do the instanceof check (but it should).
- Instead the method has a "pre-condition" comment that says the other must be a non-null Association!
  - We'll return to the topic of pre- (and post-) conditions later

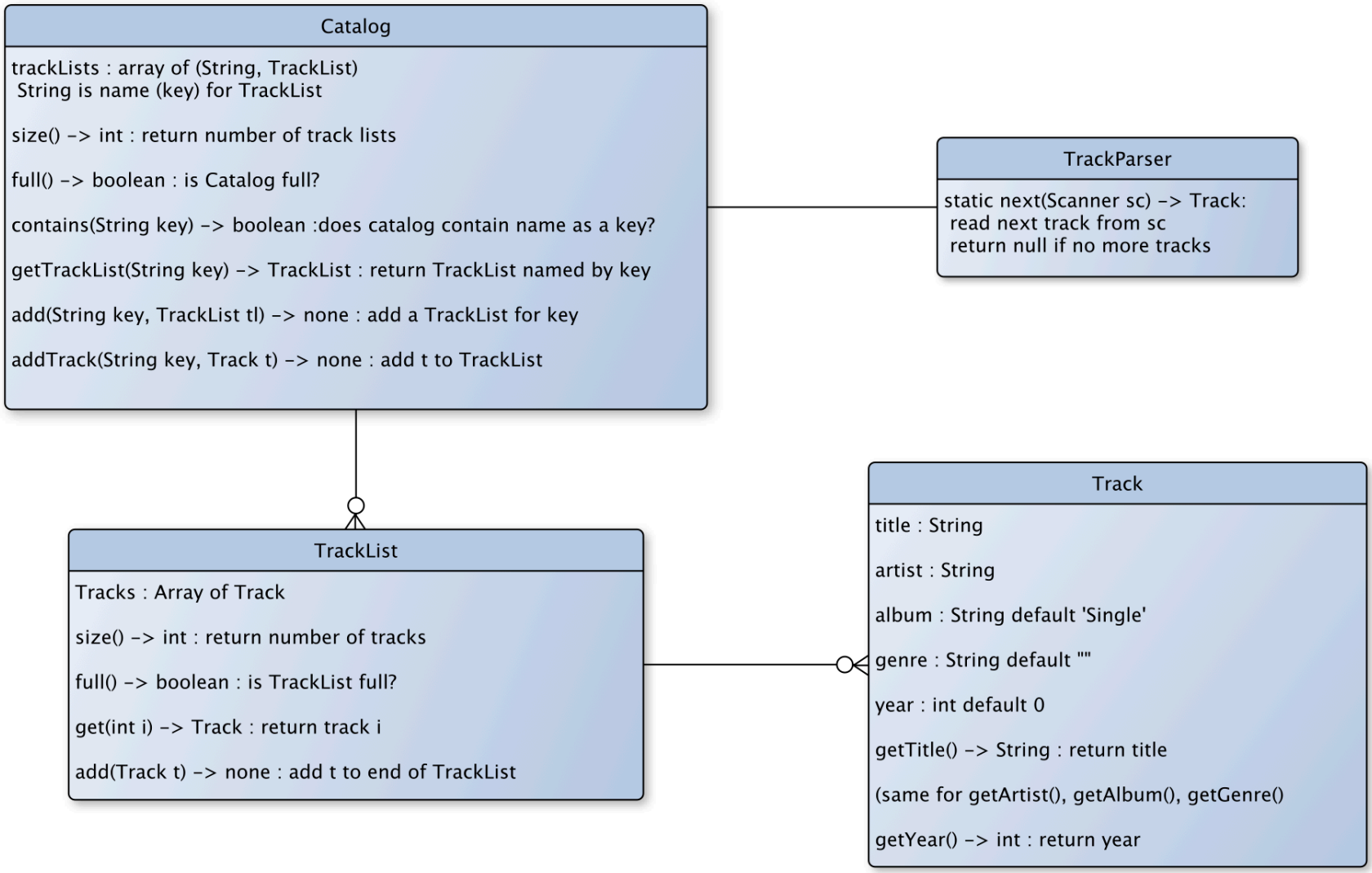
Catalog: Classes

- 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!

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!

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

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

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

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!

Lecture Ends Here