CS11 – Advanced Java

Winter 2011-2012
Lecture 1
Welcome!

- ~8 lectures
- Lab sequence focuses on a larger project
  - Completion will probably take entire term
  - Lots of opportunities to use neat Java features!
  - Opportunities to use other tools and libraries

**Grading**
- All labs must be correct, and of good quality
- Any issues in your work will require fixing
- Must pass all assignments to pass course
Assignments and Grading

- Labs focus on lecture topics
  - ...and lectures cover tricky points in labs
  - Come to class! I give extra hints. 😊

- Labs are given a score in range 0..3, and feedback
  - If your code is broken, you will have to fix it.
  - If your code is sloppy, you will have to clean it up.

- Must have a total score of 18/24 to pass CS11 Java
  - (or 75% of the possible points in the class)
  - Can definitely pass without completing all labs

- Please turn in assignments on time
  - You will lose 0.5 points per day on late assignments
Lab Submissions

- Using csman homework submission website:
  - [https://csman.cs.caltech.edu](https://csman.cs.caltech.edu)
  - Many useful features, such as email notifications
- **Must** have a CS cluster account to submit
  - csman authenticates against CS cluster account
- CS cluster account also great for doing labs!
  - Can easily do the labs on your own machine, since Java works the same anywhere
  - Just make sure you have Java 1.6+
Advanced Java?

- Assumes the following Java knowledge:
  - Familiarity with classes, access-modifiers, inheritance, nested classes
  - Basic familiarity with exceptions and exception-handling
  - Basic familiarity with Swing API, AWT events
  - Basic understanding of Java collection classes
  - Good coding style, Java naming conventions

- Focuses on:
  - Techniques for larger-scale projects
    - Automated build-tools, unit-testing, doc-gen, etc.
  - More esoteric aspects of Java language and API
Advanced Java Project

- We will write a networked Boggle game
- Boggle is a word game
  - 4x4 grid of letters
    - “A” .. “Z” and “Qu”
  - Players form words from the grid
    - Start at a particular cell
    - Take steps in any direction
    - Letters cannot be reused in a word
At the end of each round, players compare their word-lists

- If multiple players found a particular word, it is removed from everybody’s list
- Players get points for the words that only they found.

Words are scored based on their length

- 3-4 letters: 1 point
- 5 letters: 2 points
- 6 letters: 3 points
- 7 letters: 5 points
- 8+ letters: 11 points
- “Qu” is scored as two letters, not one.
This Week: A Warm-Up

- Create a class to represent lists of words
- Each word appears exactly once in the list
- Want efficient add/remove operations and membership tests
- Need to support certain “set operations”
  - Add a word-list into another word-list (set union)
  - Subtract a word-list from another (set difference)
- Need to support loading a word-list from a file
  - For the dictionary of “known valid words”
Implementing the Word-List

- Java provides us with tools to make this easy
  - String manipulation operations
  - Collection classes
  - File IO operations

- Use these tools to make your life easier! 😊
  - Your code for this week should be pretty straightforward.
Java Collections

- Very powerful set of classes for managing collections of objects
  - Introduced in Java 1.2

- Provides:
  - Interfaces specifying different kinds of collections
  - Implementations with different characteristics
  - Iterators for traversing a collection’s contents
  - Some common algorithms for collections

- Very useful, but nowhere near the power and flexibility of C++ STL
Collection Interfaces

- Generic collection interfaces defined in `java.util`
  - Defines basic functionality for each kind of collection

**Collection** – generic “bag of objects”

**List** – linear sequence of items, accessed by index

**Queue** – linear sequence of items “for processing”
  - Can add an item to the queue
  - Can “get the next item” from the queue
  - What is “next” depends on queue implementation

**Set** – a collection with no duplicate elements

**Map** – associates values with unique keys
More Collection Interfaces

- A few more collection interfaces:
  - `SortedSet` (extends `Set`)
  - `SortedMap` (extends `Map`)
  - These guarantee iteration over elements in a particular order

- These require elements to be comparable
  - Must be able to say an element is “less than” or “greater than” another element
  - Provide a total ordering of elements used with the collection
Common Collection Operations

- Collections typically provide these operations:
  - `add(Object o)` – add an object to the collection
  - `remove(Object o)` – remove the object
  - `clear()` – remove all objects from collection
  - `size()` – returns a count of objects in collection
  - `isEmpty()` – returns true if collection is empty
  - `iterator()` – traverse contents of collection

- Some operations are optional
- Some operations are slower/faster
Collection Implementations

- Multiple implementations of each interface
  - All provide the same basic features
  - Different storage requirements
  - Different performance characteristics
  - Sometimes other enhancements too

- Java API Documentation gives the details!
  - See interface API Docs for list of implementers
  - Read API Docs of implementations for performance and storage details
List Implementations

- **LinkedList** – doubly-linked list
  - Each node has reference to previous and next nodes
  - O(N)-time access of \( i^{th} \) element
  - Constant-time append/prepend/insert
  - Nodes use extra space (previous/next references, etc.)
  - Best for when list changes frequently over time
  - Has extra functions for get/remove first/last elements

- **ArrayList** – stores elements in an array
  - Constant-time access of \( i^{th} \) element
  - Append is usually constant-time
  - O(N)-time prepend/insert
  - Best for when list doesn’t change much over time
  - Has extra functions for turning into a simple array
### Set Implementations

- **HashSet**
  - Elements are grouped into “buckets” based on a hash code
  - Constant-time add/remove operations
  - Constant-time “contains” test
  - Elements are stored in no particular order
  - Elements must provide a hash function

- **TreeSet**
  - Elements are kept in sorted order
    - Stored internally in a balanced tree
  - $O(\log(N))$-time add/remove operations
  - $O(\log(N))$-time “contains” test
  - Elements must be comparable
Map Implementations

- Very similar to **Set** implementations
  - These are *associative containers*
  - Keys are used to access values stored in maps
  - Each key appears **only once**
    - (No multiset/multimap support in Java collections)

- **HashMap**
  - Keys are hashed
  - Fast lookups, but random ordering

- **TreeMap**
  - Keys are sorted
  - Slower lookups, but kept in sorted order
Up to Java 1.4, collections only stored `Object` references.

```java
LinkedList points = new LinkedList();
points.add(new Point(3, 5));
Point p = (Point) points.get(0);
```

Could add non-`Point` objects to your `points` collection!
- Retrieval could fail with `ClassCastException`

Also, casting everything just gets annoying
- Older collection code was littered with casts
Java 1.5 Generics

- Java 1.5 introduced **generics**
- Specify the type of objects stored in your collection:
  ```java
  LinkedList<Point> points =
      new LinkedList<Point>();
  points.add(new Point(3, 5));
  Point p = points.get(0);
  ```
- Compiler only allows **Point** objects to be added to the **points** collection
  - Compile-time error if you try to pass another reference type
- No cast is necessary when retrieving **Point** objects from the collection
Collections and Generics

- Lists and sets are easy:

```java
HashSet<String> wordList = new HashSet<String>();
LinkedList<Point> waypoints = new LinkedList<Point>();
```

- Element type must appear in both variable decl. and in `new`-expression.

- Maps are more verbose:

```java
TreeMap<String, WordDefinition> dictionary = 
    new TreeMap<String, WordDefinition>();
```

- First type is key type, second is the value type.

- See Java API Docs for available operations.
Iteration Over Collections

- Often want to iterate over values in collection
- **ArrayList** collections are easy:
  ```java
  ArrayList<String> quotes;
  ...
  for (int i = 0; i < quotes.size(); i++)
      System.out.println(quotes.get(i));
  ```
  - Impossible/undesirable for other collections!
- **Iterators** are used to traverse contents
- **Iterator** is another simple interface:
  - `hasNext()` – Returns `true` if can call `next()`
  - `next()` – Returns next element in the collection
- **ListIterator** extends **Iterator**
  - Provides many additional features over **Iterator**
Using Iterators

- Collections provide an `iterator()` method
  - Returns an iterator for traversing the collection
- Example:
  
  ```java
  HashSet<Player> players;
  ...
  Iterator<Player> iter = players.iterator();
  while (iter.hasNext()) {
    Player p = iter.next();
    ...
    // Do something with p
  }
  ```

  - Iterator should also use generics
  - Can use iterator to delete current element, etc.
Java 1.5 Enhanced For-Loop Syntax

- Setting up and using an iterator is annoying
- Java 1.5 introduces syntactic sugar for this:
  ```java
  for (Player p : players) {
    ... // Do something with p
  }
  ```
  - Can’t access actual iterator used in loop
  - Best for simple scans over a collection’s contents

- Can also use enhanced for-loop syntax with arrays:
  ```java
  float sum(float[] values) {
    float result = 0.0f;
    for (float val : values) {
      result += val;
    }
    return result;
  }
  ```
Collection Elements

- Collection elements may require certain capabilities
- List elements don’t need anything special
  - ...unless `contains()`, `remove()`, etc. are used!
  - Then, elements should provide a correct `equals()` implementation

Requirements for `equals()`:
- `a.equals(a)` returns true
- `a.equals(b)` same as `b.equals(a)`
- If `a.equals(b)` is true and `b.equals(c)` is true, then `a.equals(c)` is also true
- `a.equals(null)` returns false
Set Elements, Map Keys

- Sets and maps require special features
  - Sets require these operations on set-elements
  - Maps require these operations on the keys
- `equals()` must definitely work correctly
- `TreeSet`, `TreeMap` require sorting capability
  - Element or key class must implement `java.lang.Comparable` interface
  - Or, an appropriate implementation of `java.util.Comparator` must be provided
- `HashSet`, `HashMap` require hashing capability
  - Element or key class must provide a good implementation of `Object.hashCode()`
You write this code:

```java
// Helper to print the contents of a list
void printList(List<Object> lst) {
    for (Object o : lst)
        System.out.print("  " + o);
}

List<Point> points = new LinkedList<Point>();
... // Fill in the list with some points.
printList(points);
```

Should Java allow this code?
If this code were allowed, `printList()` could add arbitrary objects to `points`!

// Helper to print the contents of a list
void printList(List<Object> lst) {
    for (Object o : lst)
        System.out.print("  " + o);
}

List<Point> points = new LinkedList<Point>();
...
    // Fill in the list with some points.
printList(points);

Fortunately, Java does not compile this. 😊
Input/Output in Java

- **java.io** package contains classes for reading and writing data
  - File IO – reading/writing individual files on the filesystem
  - Device IO – network sockets, serial ports, other external devices
- A second package was added in Java 1.4
  - **java.nio**, for advanced IO operations
  - Examples:
    - Mapping part of a file into memory for high-performance reading/writing
    - Being able to listen for data on many network sockets at the same time
Basic IO in Java

- In `java.io` package, two major categories of IO operations
  - Reading and writing byte-streams:
    - `InputStream`, `OutputStream`, and (many) subclasses
    - Good for reading/writing raw data
  - Reading and writing character-streams:
    - `Reader`, `Writer`, and subclasses
    - Good for reading/writing text, especially locale-specific text
- Input/output stream and reader/writer classes are abstract base classes
  - Concrete implementations are provided for specific uses
Input-Stream Operations

Input stream and reader base classes provide a set of basic operations:

- **int read()**
  - Reads one byte

- **int read(byte[] b)**
  - Reads into an array of bytes

- **int available()**
  - Estimates how many bytes can be read without blocking

- **long skip(long n)**
  - Skips over, and discards, n bytes from the stream

- **void mark(int rdlimit)**
  - Remembers the “current position” of the stream

- **void reset()**
  - Resets the stream position to the last marked position

- **void close()**
  - Closes the input stream

Readers are nearly identical, but read char values instead of byte values.

Not all streams provide all of these capabilities!
Output-Stream Operations

- Output streams are much simpler:
  - `void write(int b)`
    - Writes one byte
  - `void write(byte[] b)`
    - Writes out an array of bytes
  - `void flush()`
    - Forces any buffered bytes out the stream
  - `void close()`
    - Closes the output stream

- Writers have similar capabilities
  - Again, writers use `char` instead of `byte`
  - Also have a few extra methods, for strings and character sequences
1. Get an input-stream or output-stream for a source or target of data
   // filePath is path and filename of a specific file
   FileInputStream fis = new FileInputStream(filePath);

2. If necessary, wrap the stream with another stream to add any needed capabilities
   // Buffer the stream so small reads are more efficient
   BufferedReader bis =
      new BufferedReader(new FileInputStream(fis));

3. Use the outermost stream for IO operations.
   // Read some data from the input file.
   byte[] buf = new byte[1024];
   bis.read(buf);
Some Useful Stream Classes

- `java.io.FileInputStream` and `FileOutputStream` for reading and writing data files
- `java.net.Socket` has `getInputStream()` and `getOutputStream()` methods
- `java.util.zip` package has compression libraries
  - Can open an input-stream or output-stream on an entry within a .zip file, for example
- `java.io.ByteArrayInputStream` and `ByteArrayOutputStream`
  - Provide stream operations for growable arrays of bytes
Streams and Readers

- Most input/output stream providers don’t also provide readers/writers
- Two classes to convert to reader/writer:
  - `java.io.InputStreamReader`
    - Constructor takes an `InputStream` object
  - `java.io.OutputStreamWriter`
    - Constructor takes an `OutputStream` object
- Very useful when you need to read/write text over an input/output stream
Several ways to represent a file or directory
- A `String` containing the path to the file/directory
- A `java.io.File` object
  - Provides many useful features!
  - Convert a relative path to an absolute path, or vice versa
  - Get `File` objects for all root directories of the filesystem
  - Test if a file exists, if it’s readable or writable, etc.

Java has classes for opening file input/output streams, as well as opening readers/writers on files
- Makes it easy to work with binary files or text files
- Can pass these classes a `String` path, or a `File` object
API Documentation

- Documenting code is extremely important
  - Specify requirements and expected behaviors
  - Record design-decisions in the code
  - Any important usage details, error conditions, etc.
- Best practice is to put these docs into the code itself
  - Good commenting practices…
  - *Much* easier to keep up-to-date if in same place
- Automatic doc-gen tools can process your source-files and generate useful/pretty API-docs
  - *Exactly* how the Sun Java API-Docs are produced!
Javadoc!

- Sun provides `javadoc` tool with Java Developer Kit
- `javadoc` processes your source-files
  - Comments starting with `/**` are javadoc comments
  - Must precede classes, fields, methods, etc.
  - Comments inside method-bodies are ignored.

- Example:
  ```java
  /**
   * A class to represent a player's spaceship.
   */
  public class PlayerShip {
    /** Location of the ship's center. */
    Point2D.Float loc;
    ...
  }
  ```
Javadoc Comments

- Javadoc generates a “brief” comment and a “detailed” comment
- Brief comment is first sentence of javadoc comment
  - Used in lists of classes, methods, fields, etc.
- Detailed comment is everything in the comment
  - Used in docs for a particular class, method, field, etc.
- Make that first sentence count!
  - A brief summary statement, containing essential details.
  - Other details go in subsequent sentences, and will appear in detailed docs.
Javadoc Tags!

- Can embed tags in your javadoc comments
  - Link to other relevant classes
  - Associate special meaning with specific notes
  - Tag format is @tag, or {@inlinetag}

Example:

```java
/**
 * A class to represent a player's spaceship.
 * @author Donnie Pinkston
 * @version 1.0
 */
public class PlayerShip {
```
Javadoc Tag Usage

- Different tags can be used in different places
- Can be used only on classes and interfaces:
  - @author – person who wrote the class/interface
  - @version – current version information
- Can be used only on constructors and methods:
  - @param – describe individual parameters
  - @return – describe what a method returns
  - @throws – what exceptions are thrown, and when
- Can appear on anything:
  - @see – refer to another class, interface, method, etc.
  - @since – version where this thing was introduced
  - @deprecated – mark as “shouldn’t be used anymore”
Referring to Other Classes, etc.

- **@see** tag lets you refer to another class, etc.
- Refer to another class:
  - `@see TargetZone`
- Refer to a field or method in another class:
  - `@see TargetZone#loc`
  - `@see TargetZone#intersects(PlayerShip)`
- Refer to another field or method in this class:
  - `@see #dirAngle`
  - `@see #turnLeft()`
- Can also embed `{@link ...}` tags in comments
  - Syntax is similar to `@see` tags
Running Javadoc

- Can run javadoc from command-line
  
  javadoc -d docs *.java

- `-d` option specifies where to put the results
  - Can specify a relative or absolute path
  - Directory is created automatically
  - Default target is the current directory! Yuck.
  - Entry-point for API-docs is `index.html` file.

- Javadoc has many more details and options!
  - Will dig into these in subsequent weeks

  [http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html](http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html)
This Week’s Assignment

- Write a basic class for representing word-lists
  - Support all necessary operations for Boggle game
  - Support ability to load a list of words from a file
  - Write a simple test class to try out your code

- Comment your code!
  - Use javadoc-style comments
  - Run javadoc to generate results
  - Comment every class and method, at least briefly
  - Easier to do this as you go!
Next Week

- Specifying metadata for classes and methods using Java annotations
- Creating automated test suites for your classes