CS 5044
Object-Oriented Programming with Java

Q&A Session

Java Collections Framework

- Standard built-in classes to hold multiple objects ("elements") in various ways
- **Collection** ("a bunch of objects" all of the same type)
  - *Set*: `HashSet` and `TreeSet`
    - No duplicates, very fast lookup for `contains()`, access via iteration only (no index)
    - `TreeSet` is sorted by a natural ordering of the elements; `HashSet` is not sorted at all
  - *List*: `ArrayList`, `LinkedList`, and `Stack`
    - Duplicates allowed, very slow lookup for `contains()`, very fast access via index
    - Always sorted by the order added/inserted (not by comparisons among elements)
    - `Stack` provides direct LIFO operations
  - *Queue*: `LinkedList`
    - A different interface used to access specific methods of a `LinkedList`
    - Exposes support for direct FIFO operations
- **Map** ("a bunch of *pairs* of objects" all of the same types, related as *key*-to-*value*)
  - *Map*: `HashMap` and `TreeMap`
    - The collection of key objects is stored as a `Set` (see above, and also later)
    - Each key object is mapped to a single value object
    - Very fast lookup of any value by its associated key

Using the Collections Framework

- Most common methods:
  - **Collection** methods: `add()`, `remove()`, `size()`, `clear()`, `contains()`, `get()`, `isEmpty()`
    - Note: This `get()` is a lookup of element by *index* (which is not supported by `Set`)
      - You must use an enhanced-for to iterate over `Set` elements
    - Note: `contains()` is typically most useful for `Set`
  - **Map** methods: `put()`, `remove()`, `size()`, `clear()`, `containsKey()`, `get()`, `isEmpty()`, `keySet()`
    - Note: This `get()` is a lookup of *value* by *key*
- The above methods are sufficient for all upcoming projects
  - (Full disclosure: Next week we'll cover a constructor that can help you more conveniently solve one small issue you'll eventually encounter, but its use is entirely optional)

Special considerations for Set (and keys of a Map)

- Elements added to a `Set` (or as keys in a `Map`) *require* uniqueness testing
  - Because duplicates are not allowed, elements must allow meaningful comparisons
    - Built-in classes (such as `String` or `Integer`) generally work exactly as expected
    - Custom classes need additional work (already done in Project 4, where applicable!)
      - You must override the default `equals()` method (Chapter 9; more on this next week)
        - » For example, the `Placement` class from Project 3 follows a very common pattern:

```
@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Placement)) { // if null or incompatible, it's not equal
        return false;
    }
    Placement other = (Placement)obj; // cast to our own type for access to fields
    return (other.column == this.column) && (other.rotation == this.rotation);
}
```

      - You should (always!) also override `hashCode()` to ensure it's consistent with `equals()`
  - For *sorted* implementations (`TreeSet` and `TreeMap`) there may be additional concerns:
    - Primitive wrappers, along with `String` and a few others, work exactly as expected
    - Most classes (including all custom classes) must implement the `Comparable` interface
      - There's just one method required, called `int compareTo(Object obj)`
        - » returns -1, 0, or 1 for less than, equal to, or greater than the specified object
        - » The `compareTo()` method should (always!) be consistent with `equals()`

Two very common mistakes with collections

- When you fetch an element from a collection, you're getting the original object
  - You're not retrieving a copy of the object; you're retrieving a reference to the object
  - You don't need to "put it back" into the collection (even after mutating it)
- Don't use index location as an ID to associate elements of multiple collections
  - This design is extremely fragile and requires far more code complexity
    - Also ignores encapsulation, and many other fundamentals of object-oriented design
  - Example of a very poor practice:
    - `List<Integer> studentIDs;`
    - `List<String> studentNames;`
    - `List<Integer> studentEnrollmentYears;`
    - `List<Double> studentGPAs;`
  - Preferred equivalent (best practice):

```
public class StudentInfo {
    private int id;
    private String name;
    private int enrollmentYear;
    private double gpa;
    // constructor, accessors, and mutators...
}
```

- `Map<Integer, StudentInfo> studentsByID;`

About specifying the generics

- When declaring the collection, specify the "generic" (the object type being stored)
  - For example:
    - `Collection<ElementType>`
    - `Map<KeyType, ValueType>`
  - This allows the compiler to enforce the types of the elements within the collections
    - It's only a warning -- not an error -- to leave the generic unspecified
      - However, you should still ALWAYS specify the generics
  - Only object types are supported, but there are wrappers for primitives:
    - Types `Integer`, `Boolean`, `Double`, and so forth are specified instead of primitive types
    - Auto-boxing and auto-unboxing handles all conversions behind the scenes
- Constructors typically specify the "diamond operator" (just an empty generic)
  - The compiler can infer the generic from the specification in the declaration:

```
Map<String, Boolean> myMap; // variable declaration, specifying the generic

myMap = new HashMap<>(); // definition/initialization (the compiler infers the generic)
```

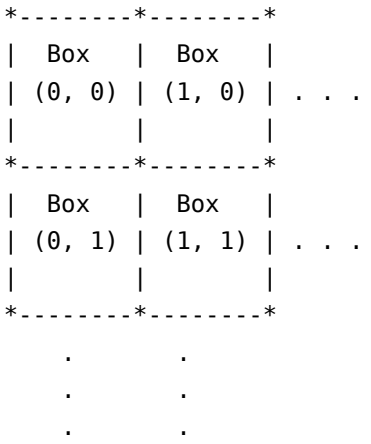
- Best practices regarding level of specificity:
  - Declare *variables* (collection types) as broadly as practical, such as `Map` rather than `HashMap`
  - Specify *generics* (element types) as precisely as possible, such as `String` rather than `Object`

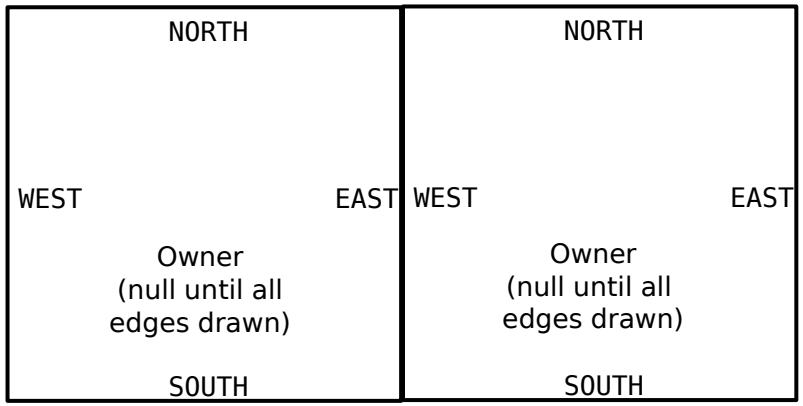
Project 4: Dots and Boxes

- Main goal:
  - Implement an interface (somewhat similar to the Tetris project in this respect)
- Primary areas of focus:
  - Proper use of appropriate collections (particularly `HashMap` and `HashSet`)
  - Developing solutions with more than one class
  - Generating test cases to cover system requirements
  - Continuing to explore the usage of libraries via API
- Follow the many best practices we've learned already throughout the term
  - TDD, minimize redundancies, leverage type-safety, enforce encapsulation, etc.
- Provided classes:
  - DotsAndBoxes *interface* (this is what you need to implement)
  - Coordinate *class* stores a single (x, y) location, with getNeighbor() helper
  - Player *enum* (ONE and TWO) with getOpponent() helper
  - Direction *enum* (NORTH, SOUTH, EAST, WEST) with getOpposite() helper
  - GameException *exception* thrown by your implementation under certain conditions
- Demo: Getting started in Eclipse...

Project 4: Overview

- Adjacent boxes share a common edge, addressable from either box
  - For example, the WEST edge of (1, 0) is the same as the EAST edge of (0, 0)
- Please carefully note the coordinate system (see the `Coordinate` API for details)
  - Location (0, 0) represents the upper-left box





- You do NOT need to store any information about the "dots" in the game
  - Each dot just represents a single corner of one or more of the boxes

Project 4: Notes and additional information

- Overall notes:
  - You're required to develop a separate class to reasonably delegate responsibilities
    - Something like `Box` (see below) is very highly recommended
  - The score `Map` is much easier to generate on demand than to maintain as a field
    - Iterate through all boxes, then tally the scores by box owner
  - Use helper methods, such as `checkInit()` and `findBox()`, to throw `GameException` as appropriate
    - See next slide for more details about exceptions
- Recommended delegation approach:
  - DABGame, the main implementation, holds only the following state fields
    - `private Map<Coordinate, Box> boxGrid;`
    - `private Player currentPlayer;`
    - `private int gridSize;`
  - Each Box object, representing a single box within the grid, holds only these state fields:
    - `private Player owner;`
    - `private Collection<Direction> drawnEdges;`

Project 4: Exceptions

- Exceptions (much more in a few weeks; this just provides some initial exposure)
  - You've probably already experienced `NullPointerException` and `IllegalArgumentException`
  - See sections 11.4.1 and 11.4.2 for additional background, but this is all you need:
    - Throwing exceptions (to indicate that something has gone wrong):

```
if ( /* some condition */ ) {
    throw new GameException();
}
```

    - Catching exceptions (to handle when something has gone wrong):

```
try {
    // some lines that might throw an exception
} catch (GameException ge) {
    // handle the exceptional case here
}
```

- Testing exceptions:
  - Use `@Test(expected=GameException.class)` (or a try-catch structures) to test exceptions
- Demo: Exceptions in Eclipse... (time permitting)