CS 5044
Object-Oriented Programming with Java

Q&A Session

Inheritance in general

- What is the whole point of inheritance in OO analysis, design, and programming?
  - Inheritance is a fundamental aspect of object-oriented vs. procedural techniques
    - Reduces redundancies, by reusing implementations throughout a hierarchy
  - However, inheritance somewhat breaks pure encapsulation, so take care not to abuse it
    - Use it only where it naturally meets some design goals
    - Never force-fit inheritance into a solution, just because it's possible to do so
- The textbook cites "data and behavior" being inherited
  - In Java, we inherit fields (internal state) and methods (accessors and mutators)
  - Note that we can also override, whenever something inherited is inappropriate
    - Although if you end up overriding too much, it probably isn't a very good subclass!
- Subclasses are a "specialization" of the superclass
  - The subclass should involve **more** fields and/or **more** methods than its parent
  - Note that this is different from a "special case" in many mathematics fields:
    - In geometry, a square is considered a special case of a rectangle, but...
    - ...Rectangle would be a subclass of Square, because it needs an additional field

Inheritance in Java

- The subclass "extends" the superclass
  - `public class SomeSubClass extends SomeSuperClass { ... }`
- Calling the superclass constructor from the subclass constructor is common:

```
public SomeSubClass() { // subclass constructor
    super(); // calls the constructor of the superclass (may include parameters)
}
```

- Nothing special is required to call methods declared *only* in the superclass

```
public void doSomething() {
    incrementValue(); // calls the method of the superclass
}
```

  - Typically the superclass fields are all declared as private
    - Subclasses must then use superclass getter/setter methods if access is needed
- If you've overridden methods, explicitly specify the superclass if intended

```
@Override
public void someMethod(String s, int x) {
    super.someMethod(s); // calls the method of the superclass
    someField = x; // sets a subclass instance variable
}
public void someOtherMethod(String s) {
    super.someMethod(s);  // calls the method of the superclass
    someMethod(s, -1);    // calls the method of the subclass
    someOtherField = s.toLowerCase(); // sets a subclass instance variable
}
```

Inheritance from Object

- It's often useful to override `equals()`, `hashCode()`, and `toString()`
- From last week: `equals()` and `hashCode()` are required by certain collections
  - You must override these in classes that will be stored in a `Set` (or as *keys* of a `Map`)
  - `Set` uses `equals()` to determine equivalence among elements
    - You should always override `hashCode()` whenever you override `equals()`
      - This is done already in the Coordinate class of Project 4
      - We'll explore this further in an upcoming homework
- The `toString()` method is usually just for convenience, or for user output
  - It's called implicitly to generate the `String` representation of any object:

```
System.out.println(someObject); // implicitly calls someObject.toString()

String display = "value is now: " + anotherObject; // implicitly calls anotherObject.toString()
```

  - It's a bad practice to use the String representation for any program logic
    - Don't assume the String representation is actually meaningful to the code
    - In large scale systems, Strings are often localized based on language preferences

Inheritance and references

- Single inheritance is enforced throughout all classes
  - Every class extends exactly one parent superclass
    - The parent superclass is `Object` if not specified otherwise in the class declaration
  - Any number of subclasses may extend any given class (unlimited child classes)
- Polymorphism: References may point to an object of their type, or of any subtype
  - The type of the reference is an "apparent" type; the type of the object is the "actual" type
  - Cast the reference if you need to call a method from a subclass of the apparent type
    - The compiler won't let you use a field/method that's not in the apparent type
      - The compiler can only prevent blatantly invalid casts (no common sub/super classes)
      - Any other invalid casts will result in a **runtime** `ClassCastException`
    - Examples:

```
Object o = "hello"; // o has actual type String and apparent type Object
String so = (String)o; // OK to cast; apparent type of so is String
String s = o.toLowerCase(); // compile-time error; apparent type of o is Object
String s = ((String)o).toLowerCase(); // OK and equivalent to calling so.toLowerCase()
Integer i = (Integer)o; // run-time ClassCastException due to incompatible types
Box b = new Box(); // b has actual type Box and apparent type Box
String sb = b.toString(); // OK with no cast, even if Box does *not* implement toString()
```

  - When any method is called on any reference, the class hierarchy is searched upward
    - Starting with *actual* type, ending at `Object`, the first implementation found is executed

Interfaces: the other inheritance mechanism

- Classes may implement an unlimited number of interfaces

```
public class Mashup implements AI, DotsAndBoxes {  ... methods from both AI and DotsAndBoxes ...  }
```

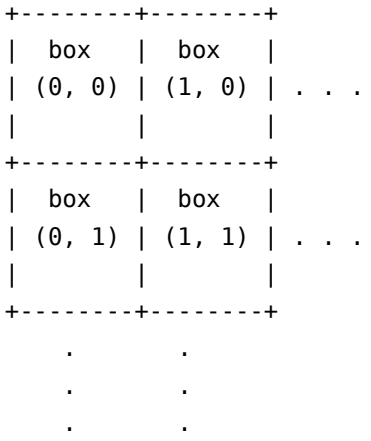
- Each interface is treated *almost* as a superclass
  - Casting, reference, and polymorphism mechanisms are all identical to inheritance:

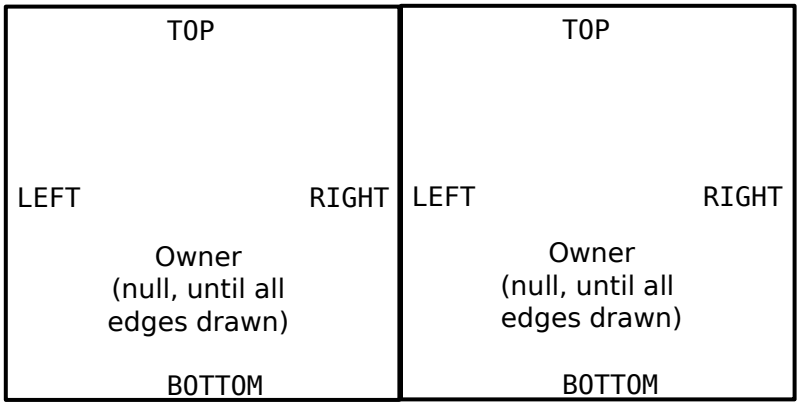
```
Object m = new Mashup();
AI mashAI = (AI)m;
DotsAndBoxes mashDAB = (DotsAndBoxes)m;
```

  - In most situations, interfaces are generally preferred over superclasses
    - Especially true since Java 8, which introduced "default" methods of interfaces
      - Default methods are effectively inherited by all implementations of the interface
  - However, interfaces can never hold state fields (no instance variables allowed)
    - If you need to inherit **fields**, you must inherit from a superclass, not an interface
- Several textbook topics won't be very useful until much later in the semester
  - Callback methods (Chapter 10.4) are prevalent in GUI programming (Module 12)
  - Explicit inner classes (Chapter 10.5) aren't very common at all, however...
    - "Anonymous" inner classes (Special Topic 10.4) allow the use of...
    - Lambda Expressions (Java 8 Note 10.4) which form the foundations of...
    - Functional Programming, which we'll cover later (Module 13)

Project 4: Overview

- Please carefully note the coordinate system (see the `Coordinate` API for details)
  - Location (0, 0) represents the upper-left box





- Adjacent boxes share a common edge, accessed from either coordinate
  - For example, the LEFT edge of (1, 0) is also addressable as the RIGHT edge of (0, 0)
- Note: The boxes, and their edges, are the important objects to model
  - You don't need to store any information about the "dots" in the game
    - Dots represent the box corners; lines are drawn to form the edges of 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 throwing exceptions
  - Your drawEdge() method must use a try-catch structure to handle missing neighbors
    - See next slide for more details about catching 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 (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, some of which 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
- You're required to use at least one try-catch structure
  - This is actually easy to integrate into the edge drawing algorithm
    - We must consider the adjacent box, if there is one; otherwise skip a few steps
      - It can be done with normal if() branches, but try-catch is a much more natural approach