MASTERS OF INFORMATION TECHNOLOGY

CS 5044
Object-Oriented Programming with Java

Q&A Session

Array and ArrayList

- Array is a special type of object, with special syntax rules
  - Arrays contain a fixed number of elements, which is nearly always inefficient
- VarArgs (Special Topic 7.1) is an interesting usage for arrays
  - You can develop a method that takes any number of arguments of the same type

```
public int findMin(int... values) { // Note that Math.min() only takes 2 values
    int temp = Integer.MAX_VALUE;
    for (int i = 0; i < values.length; i++) { // could use an enhanced-for loop instead!
        if (values[i] < temp) { temp = values[i]; }
    }
    return temp;
}
```

    - Called as `findMin(z)` or `findMin(a, b, c, d, e, f)` or `findMin(1, 2, 5, 3)` etc.
    - We'll need to call -- but not develop -- a VarArgs method in Project 3
- ArrayList is just a normal object, with normal object syntax
  - Probably the most useful methods: `add()`, `get()`, `size()`, `isEmpty()`, `remove()`, and `clear()`
  - We used ArrayList in Project 2, but we will NOT need to use it in Project 3!
- Why would we ever use an array, rather than an `ArrayList`?
  - When implementing methods with VarArgs
  - When the number of elements is fundamentally fixed (such as vector/matrix math)
  - When interacting with older/legacy code is required

Enhanced: a better for() loop

- The enhanced for() loop is quite convenient, for both collections and arrays

```
for (LogEntry entry : eventLog) {
    if (entry != LogEntry.NO_ACTION_TAKEN) {
        System.out.println("Entry: " + entry);
    }
}
```

- Suitable for fetching all elements in order, when we don't care about the index
  - Generally you should *always* prefer the enhanced for(), *unless* you need the index
- What if the index is needed? Just convert it to a regular for() loop:

```
for (int i = 0; i < eventLog.size(); i++) {             // note the change
    LogEntry entry = eventLog.get(i);                   // note the addition
    if (entry != LogEntry.NO_ACTION_TAKEN) {            // no changes here...
        System.out.println("Entry " + i + ": " + entry); // ...but we can access the index i
    }
}
```

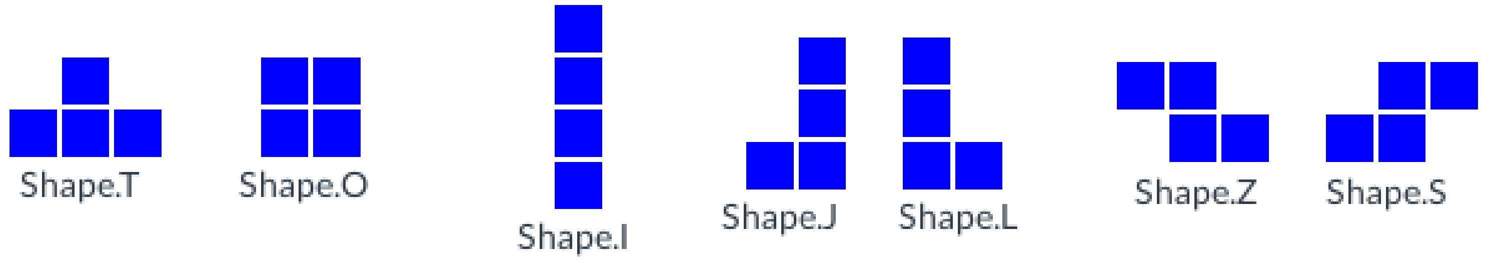
- We will practice both kinds of loops in Project 3
  - Cost calculation methods will need only conventional for()
  - Find best placement method will need one enhanced for() and one conventional for()

Project 3

- Tetris!
  - We've been tasked to develop an artificial intelligence (AI) that can play Tetris
- An "interface" has been provided; you will *implement* the interface
  - We'll learn much more about interfaces in a few weeks, but for now:
    - An interface is just a way to formally specify a set of methods, without any code
    - To implement the interface, we develop a class with actual code for all the methods
  - Eclipse can generate all the placeholder methods for us automatically
    - Once that's done, it becomes *exactly* like our previous projects
    - Plus, the compiler will ensure we don't accidentally change the method headers!
- Let's review the project setup in Eclipse…

Meet the enums

- `enum Shape`
  - One value for each of the seven tetromino pieces:



  - Each Shape provides a Set of distinct Rotation values (default orientations shown above)
    - In Java, a `Set` is very similar to an `ArrayList`, but without any index values
      - There are a few other differences, but we'll explore those next week
      - For now, just use an *enhanced for()* loop to iterate over each element of the Set
  - You can also query the width of the shape (in blocks) after applying any valid Rotation
- `enum Rotation`
  - One value for each 90° rotation:
    - `NONE`
    - `CCW_90`
    - `CCW_180`
    - `CCW_270`

Game on

- `class Board`
  - Represents the state of the playing board at any given time
    - Class is immutable, meaning there are no public mutator methods
  - Provides public static constants: WIDTH and HEIGHT
    - Use these fields to avoid "magic numbers" (Programming Tip 4.1)
  - Contains a collection of fixed blocks we can query via getColumn(col)
    - Returns a boolean array, where true indicates the presence of a fixed block
  - We can ask the board to show us the hypothetical result of placing another piece
    - The piece will be placed and dropped, then any full rows will be cleared
    - Creates a new Board object; does not mutate the existing Board
  - We can also construct new Board objects with arbitrary blocks for testing
- `class Placement`
  - A Placement just holds together a Rotation value and a column index
  - This is what our AI needs to return to the game engine, for the given Shape:
    - First, the specified Rotation will be applied to the Shape
    - Next, the rotated Shape will be moved horizontally, such that...
      - ...the left-most block of the rotated Shape will be in the specified column
    - Invalid Placement objects are ignored by the game engine (with message to console)

Mind games

- `interface AI` (this is the interface we need to implement)
  - The primary method of interest is called by the game engine for each Shape
    - `public Placement findBestPlacement(Board currentBoard, Shape shape)`
  - The remaining methods must compute specific "cost" factors for a given board:
    - `public int getColumnHeightVariance(Board board)`
    - `public int getColumnHeightRange(Board board)`
    - `public int getAverageColumnHeight(Board board)`
    - `public int getTotalGapCount(Board board)`
  - The individual cost factor methods must be developed (and tested!) first
  - AFTER completing/testing the cost factor methods, start working on `findBestPacement()`
    - For every possible Placement (Rotation and column) of this Shape:
      - Get the board that would result from this hypothetical placement
      - Calculate cost factors for result board and combine with weights*
      - If this is the lowest overall cost so far, consider this placement as the new best
    - Return the best (minimum cost) Placement to the game engine
  - *Weights can all start at 1; this places **62.5** pieces, on average, over the 4 TEST modes
    - Then adjust the weights manually, using combinations of 0, 3, 6, and 9
      - The requirement is to place at least **125** pieces, on average, over the 4 TEST modes

Project 3: Hints and tips

- Other classes:
  - `ShapeStream` and `RandomMode` are needed for the OPTIONAL challenge only
  - `Tetris5044` is used only by the game engine itself; you won't ever need to use it
- Use meaningful variable names; it really makes a difference (and will be graded!)
  - For example, loop variables should be named `col` and `row` rather than `i` and `j`
- Beware of off-by-one errors in all loop bounds (use enhanced-for where possible)
  - The bottom-most row is 0, and the left-most column is 0 (beware `<` vs `<=` and similar)
- Account for *all* of the rows of the board when calculating costs
  - There are `Board.HEIGHT` rows to be considered
    - `Board.HEIGHT_LIMIT` is only useful for the OPTIONAL challenge
- Develop helper methods to reduce redundancy (and greatly simplify your code)
  - Consider a `getColumnHeight(Board board, int col)` helper:
    - Used by `getTotalGapCount()`, `getColumnHeightVariance()`, and `getColumnHeightRange()`
  - Consider a `getColumnBlockCount(Board board, int col)` helper:
    - Used to simplify `getTotalGapCount()`
- Let's share Eclipse again, time permitting...