Reading:
1. Dale, Chapter 11 (Section 6)

Objectives:
1. Introduce searching
2. Discuss class project
3. Quiz #1

Concepts:
1. Searching
1. Introduce searching
   - Briefly describe the concept of searching and its need
   - Demonstrate a couple of searching algorithms

2. Discuss class project
   - Provide project description to the students
   - Discuss the project itself
   - Discuss what programming should be used for the projects
   - Ask the students to prepare questions for the next lecture

3. Quiz #1
Objective:
1. Become familiar with binary search

Student Activities:
1. If an array is sorted binary search technique could be used. Look through the following code and identify possible errors. You won’t be running the code in lab this week but you will be testing in next week. Try to understand what happens at each stage of the search:

```java
// Binary search
public int binarySearch ( int key, Graphics gg) {
    gg.drawstring ("Portions of array searched", xPosition, yPosition);
yPosition += 15;

    int low = 0;    // low subscript
    int high = a.length – 1;  // high subscript
    int middle;    // middle subscript

    while (low <= high ) {
        middle = ( low + high );

        printRow ( low, middle, high, gg);

        if (key == a [middle] )   // match
            return middle;
        else if (key < a [middle] )
            high = middle + 1;  // search low end of array
        else
            low = middle – 1;  // search high end of array
    }
    return –1;    // search key not found
}
```
The code should be:

// Binary search

public int binarySearch ( int key, Graphics gg)
{
    gg.drawstring ("Portions of array searched", xPosition, yPosition);
yPosition += 15;

    int low = 0;    // low subscript
    int high = a.length -1;  // high subscript
    int middle;    // middle subscript

    while (low <= high ) {
        middle = ( low + high ) / 2;

        printRow ( low, middle, high, gg);

        if (key == a [middle] )   // match
            return middle;
        else if (key < a [middle] )
            high = middle - 1;  // search low end of array
        else
            low = middle + 1;  // search high end of array
    }

    return -1;    // search key not found
}
PROJECT – TETRIS
This is the Readme for the CS 116 Tetris package -- suitable for playing around with tetris or creating a tetris based project.

Materials
Readme.txt -- this introduction
JTetris.jar -- a functioning Tetris

TetrisProject.zip -- things you need to compile tetris code
Tetris-Architecture.html -- the architecture of the classes
TetrisAssignment -- handout for the full assignment
*.java -- starter files and support files

This Readme introduces the Tetris materials, explains how to run the program, and gives some assignment ideas. Tetris-Architecture.html describes the overall design architecture of the tetris classes. The .java starter files are also heavily commented.

Running
The provided JTetris.jar contains a runnable Java version of tetris which supports the brain features (below). The program needs Java 1.2 or later -- the free download from Sun is http://java.sun.com/j2se/1.3/jre/ JTetris.jar can be run by double clicking it, or with the command line incantation "java -jar JTetris.jar".

The program includes a little slider that controls the speed. The "Brain Active" button will turn on the AI brain to play for you. The "Adversary" slider uses the AI brain to figure out which next piece will be hardest for the player. Set the adversary slider to 25%, and 25% of the pieces will be very hard to play instead of being chosen randomly. It's fun to put the adversary at about 25%, and then ask a victim to play without telling them about the adversary. The "Load Brain" button can load in a user-written brain.

Projects
1. Fun: Brain

A fun project that's not too hard is designing a tetris brain. The code to implement a brain is surprisingly short. The default brain in JTetris is called LameBrain, and it's game playing logic is less than a page long. Writing a brain requires just basic Java and OOP skills. Look at the Brain.java and LameBrain.java files to get an idea of how the system works. Write your own subclass of LameBrain and override the rateBoard() method with your own rating logic.

Run JTetris and use the "Load Brain" button with the name of your class, such as "MyBrain", to load your brain into JTetris. By default, Java finds classes that are in the current directory, so just put your .class file in the same directory as the program so it will be found when you click the Load Brain button.

See the Tetris-Architecture.html document for ideas on Brain design and some basic information on how to message the Piece and Board classes. Designing your own brain is a creative and wide-open sort of challenge.

If you run the program with the command line option "test", then it will play the same sequence of pieces every time -- seeing the same sequence of pieces each time makes it easier to interpret what the brain is doing. (perhaps this should be a control in the GUI.)

2. Hard: The Whole Thing

The obvious tetris project is to implement the whole thing -- most of the work is in the Piece and Board classes which implement the core tetris logic. The implementation is more complex than you might think, since it needs to support the brain efficiently.

It's a pretty difficult project, but it's fun. We have our 2nd year CS undergraduates do this project.

See the TetrisAssignment.rtf for the assignment handout we use.

The basic steps are...

a. Read the Tetris-Architecture.html to get a feel for the overall architecture. Also, the .java files are heavily commented.
(For each of the following classes, there is a "starter" file that has some of the basic syntax done for you.)

b. Build the Piece class in Piece.java

c. Build the JPieceTest class in JPieceTest.java. 
Use this to test the functionality of your Piece. 
(This requires a little understanding of Java drawing.)

d. Build the Board class in Board.java (this is the hardest part)

e. Use the provided JBoardTest class to test your Board implementation

f. Use the provided JTetris class to play tetris using your Piece and Board implementations.

g. (optional) Implement your own JBrainTetris subclass of JTetris that combines JTetris with the Brain AI to implement self-playing tetris and the adversary.

h. (optional) You can also implement your own Brain -- the provided LameBrain is quite simple.

It's a challenging project, but it's fun once it works. It's a nice example of using OOP to divide and conquer a complex system.

**Tetris Architecture Overview**

This is the architectural overview that describes the Tetris classes.

These Tetris classes provide a framework for a few different Tetris applications. For a fun little assignment, students can write their own "brain" AI code (surprisingly easy), and then load it into the provided JTetris GUI to see how it plays. For 2nd year CS undergraduates, we have a fairly difficult assignment where they implement all the main tetris classes. This makes a nice exercise in OOP decomposition -- dividing the rather large Tetris problem into several non-trivial classes that cooperate to solve the whole thing.

The classes that make up Tetris are...

- Piece -- a single tetris piece
- Board -- the tetris board
- JPieceTest / JBoardTest -- tester classes for Piece and Board
- JTetris -- present the GUI for tetris in a window and do the animation
- LameBrain -- simple heuristic logic that knows how to play tetris
• JBrainTetris -- a subclass of JTetris that uses a brain to play the game without a human player

As an additional feature, the tetris classes implement the logic for tetris in a way that runs quickly. The speed is needed for the later "brain" parts of the project.

Piece

The Piece class defines a tetris piece in a particular rotation. Each piece is defined by a number of blocks known as its "body". The body is represented by the (x, y) coordinates of its blocks, with the origin in the lower-left corner.

```
|   |   |
| 2 |   |
| 1 |   |
| 0 |   |
```

So the body of this piece is defined by the (x, y) points: (0, 0), (1, 0), (1, 1), (2, 1).

The Piece class and the Board class (below) both measure things in this way -- block by block with the origin in the lower-left corner. As a design decision, the Piece and Board classes do not know about pixels -- they measure everything block by block. Or put another way, all the knowledge of pixels is isolated in the JTetris class.

Each piece responds to the messages like getWidth(), getHeight(), and getBody() that allow the client to see the state of the piece. The getSkirt() message returns an array that shows the lowest y value for each x value across the piece ({0, 0, 1} for the piece above). The skirt makes it easier to compute how a piece will come to land in the board. There are no messages that change a piece -- it is "immutable". To allow the client to see the various piece rotations that are available, the Piece class builds an array of the standard tetris pieces internally -- available through the Piece.getPieces(). This array contains the first rotation of each of the standard pieces. Starting with any piece, the nextRotation() message returns the "next" piece object that represents the next counter-clockwise rotation. Enough calls to nextRotation() gets the caller back to the original piece.

Board

The Board class stores the 2-d state of the tetris board. The client uses the place() message to add the blocks of a piece into the board. Once the blocks are in the board, they are not connected to each other as a piece any more; they are just 4 adjacent blocks that will eventually get separated by row-clearing.

• place(piece, x, y) -- add a piece into the board with its lower-left corner at the given (x, y)
• clearRows() -- compact the board downwards by clearing any filled rows
Board has many methods that allow the client to look at the Board state. These all run in constant time -- makes things easy and fast for the client, but harder for the board implementation...

- `int getWidth()` -- how many blocks wide is the board
- `int getHeight()` -- how many blocks high is the board
- `int getRowWidth(y)` -- the number of filled blocks in the given horizontal row
- `int getColumnHeight(x)` -- the height the board is filled in the given column. This is $1 + \text{the y value of the highest filled block}$
- `int getMaxHeight()` -- the max of the getColumnHeight() values
- `int dropHeight(piece, x)` -- the y value where the origin (lower left corner) of the given piece would come to rest if the piece dropped straight down at the given x

**Board Undo**

The Board also supports a 1-deep undo() facility that allows the client to undo the most recent placement and/or row clearing. The undo facility is rather limited -- the client can do a single place() and a single clearRows(), and then use undo() to return to the original state. Although simple, the undo facility is fast. It turns out that the Brain (below) needs exactly this facility of a simple but fast undo. Here's how undo works...

- We'll designate the state of the board as "committed" -- a state that can be returned to.
- From a committed state, the client can do a place() operation that modifies the board. An undo() will remove the change so the board is back at the committed state.
- Then the client can do a clearRows() operation which further modifies the board. An undo() will remove all the changes so the board is back at the committed state.
- Finally, the client can do a commit() operation which marks the current state as the new committed state. The previous committed state can no longer be reached via undo().

A client can use the undo facility to animate a piece moving in the board like this: place() piece with y=15, pause, undo(), place() with y=14, pause, undo(), place() with y=13, ...

**JPieceTest, JBoardTest**

These are simple test classes that exercise the Piece and Board respectively. Tetris is complex enough that it's important to build and test the components separately.

**JTetris**

The JTetris class presents the GUI for a playable tetris game in a window. It uses Piece and Board to do the real work. Usually, I have the students write Piece and Board, but I provide JTetris.
**JBrainTetris**

JBrainTetris is a subclass of JTetris that uses the LameBrain (below) or another loaded brain to play tetris without a human player. JBrainTetris can also implement an "adversary" feature. The adversary is a cruel yet hilarious feature where the game figures out what the worst possible next piece is (using the brain), and then gives that piece to the player.

**LameBrain**

LameBrain includes heuristic logic that knows how to play the game of tetris. The LameBrain algorithm is very simple -- it knows that height is bad and creating holes is bad. Students can write their own brains. Here is the code...

```java
/*
   A simple brain function.
   Given a board, produce a number that rates
   that board position -- larger numbers for worse boards.
   This version just counts the height
   and the number of "holes" in the board.
   See Tetris-Overview.html for brain ideas.
*/
public double rateBoard(Board board) {
    final int width = board.getWidth();
    final int maxHeight = board.getMaxHeight();

    int sumHeight = 0;
    int holes = 0;

    // Count the holes, and sum up the heights
    for (int x=0; x<width; x++) {
        final int colHeight = board.getColumnHeight(x);
        sumHeight += colHeight;

        int y = colHeight - 2; // addr of first possible hole
        while (y>=0) {
            if (!board.getGrid(x,y)) {
                holes++;
            }
            y--;
        }
    }

    double avgHeight = (double)sumHeight / width;
```
// Add up the counts to make an overall score
// The weights, 8, 40, etc., are just made up numbers that appear to work
return (8*maxHeight + 40*avgHeight + 1.25*holes);
}

It's pretty easy to write better brain logic, and that alone can be the basis of an assignment. Here's some suggestions for building a better brain (or don't look at these if you want to puzzle it out yourself) ...

- Height is bad
- Holes are bad
- Stacking more blocks on top of holes is bad
- Holes that are horizontally adjacent to other holes are not quite as bad
- Holes that are vertically aligned with other holes are not quite as bad
- Tall 1-wide troughs are bad
- 1-wide troughs are not so bad if they are only 1 or 2 deep. Think about which pieces could fill a 2-deep trough -- 1, 2, or 3 out of the 7 pieces depending on the two sides of the trough.
- Concentrate on issues that are near the current top of the pile. Holes that 10 levels below the top edge are not as important as holes that are immediately below the top edge.
- At some point, my brain code always has some arbitrary constants like 1.54 and -0.76 in it that I can only lamely optimize by hand. To get the best possible brain, use a separate genetic algorithm to optimize the constants. This is another reason why the design here has such an emphasis on speed -- the genetic optimizer needs to be able to rip through millions of board positions

**Tetris Project - Instructor's Guide**

The project is made of Piece and Board classes that implement the core of the tetris game, a JTetris class (provided) that manages the game and does the animation, and Brain classes that add in game playing AI.

**Assignment Niche**

Tetris is an advanced CS2 assignment. Putting the Piece, Board, and Brain systems all together creates something larger than the typical CS2 assignment, and some of the algorithms are tricky to get right. The complexity makes for a more convincing demonstration of the benefits of a modular OOP design. From a teaching point of view, the main theme of the assignment is using OOP decomposition to divide a complex system into more manageable pieces.

We require the students to have separate test code for both Piece and Board before the two are used in the full tetris. Partly this is just an excuse to teach them about modular
test code, and partly because it really is the best way to get the whole thing to work. The project is too big to debug all at once.

Engineering issues aside, tetris is at its heart a fun and visually engaging project. The brain and adversary features add novelty and further ways to play with the system. It's the sort of assignment that the students play with for hours after the required functionality has been done. (Unless their Board doesn't quite work -- nothing sucks the fun out of game of tetris quicker than a board that, say, doesn't do row clearing right.)

I see tetris being used in a CS2 course or later to show off OOP decomposition on a large project. Programming maturity is required to code and debug the algorithms. The students probably should get 2 or 3 weeks to complete the project, and it should be due in stages, which fits well with its modular theme.

**Strengths and Weaknesses**

The best feature of the assignment is that it attacks some real complexity with OOP decomposition. It also builds something visual and fun that the students really enjoy playing with. The bad part of the assignment is that it is large, and some of the algorithms are tricky. This can be very frustrating for the students if they don't have the skills to deal with that scale of program yet. It also means that it's going to take 2 or 3 weeks out of the term.

**Alternative**

As an alternative to the full tetris project, the students can just work on their own brain code and load it in to the off-the-shelf JTetris program to try it out. This makes a fun little assignment. Brain code is surprisingly easy to write -- it requires only a basic understanding of Java and OOP. The problem itself is creative and open-ended. There is no right answer; instead the students can code up their heuristic ideas pretty easily and see what they do. The readme has some suggested brain strategies. The LameBrain that ships with the project is extremely simple -- I found that when I shipped a better brain, it seemed to inhibit the students from trying their own. With the LameBrain, it's obvious to the students that they could easily write a better one (or perhaps watching it play poorly is just too offensive to allow to pass without improvement!).

**Implementation Variants**

- The assignment can be given as a non-OOP ADT assignment. There's not really any inheritance going on -- I've given a verison of this assignment in Pascal with Piece and Board ADTs.
- The undo algorithm and interface could be simplified without compromising the assignment by relaxing the constraint that the Board and its undo system be so efficient. In particular, implement undo just by copying the whole board instead of the efficient/stingy strategy. In this way, the client can just copy the board before making changes. This is slower since it copies more, and in Java, it will
tend to introduce memory churn in the algorithm, where the stingy strategy never calls new after the initial allocation.

- The undo algorithm could work without any copying, but instead by remembering the piece and location of the last play, and then backing it out block by block. Row clearing would have to be handled separately.
- The board could use something other than a 2-d array as its internal representation. I suspect storing each column as a single 32 bit int and using a bit to represent a block would run faster. Another idea is to store the booleans in a 1-d array and do the offset arithmetic explicitly instead of using a 2-d array.
1) Answer each of the following:
   a) Lists and tables of values are stored in __________.
   b) The elements of an array are related by the fact that they have same 
      ______ and ______.
   c) The number used to refer to a particular element of an array is called its 
      ______.

2) State whether the following are true or false. If the answer is false explain why.
   a) An array can store many different types of values
   b) An array subscript should normally be of the data type float
   c) Assume you have a two-dimensional array called “a”. The following code 
      is executed: a[1, 1] = 5; is it correct? If not how should it be written

1) a) Arrays b) name, type c) subscript
2) a) False. An array can store only values of the same type
2) b) False. An array subscript should normally be an integer or an integer expression
2) c) False. It is not correct the correct way will be: a[1][1] = 5;