CS 106A, Lecture 27
Final Exam Review 1

Plan for today

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

Plan for today

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

Final exam

- Is the final exam cumulative?
- What will be tested on the final exam?
- What about all this stuff you aren't covering today?
  - Expressions and Variables
  - Java Control Statements
  - Console Programs
  - Methods, parameters, returns
  - Randomness
  - Strings and chars
  - Scanners and file processing
  - Memory

RESOURCES

- Lecture Videos
- Eclipse
- Course Staff
- Textbooks
- Pair Programming
- LaIR Help Hours
- Stanford Library Docs
- Blank Karel Project
- Blank Java Project

Midterm review session was the recorded section on Friday of Week 4

- Is the final exam going to be difficult/curved?
- How can I practice for the final?

Practicing for the final

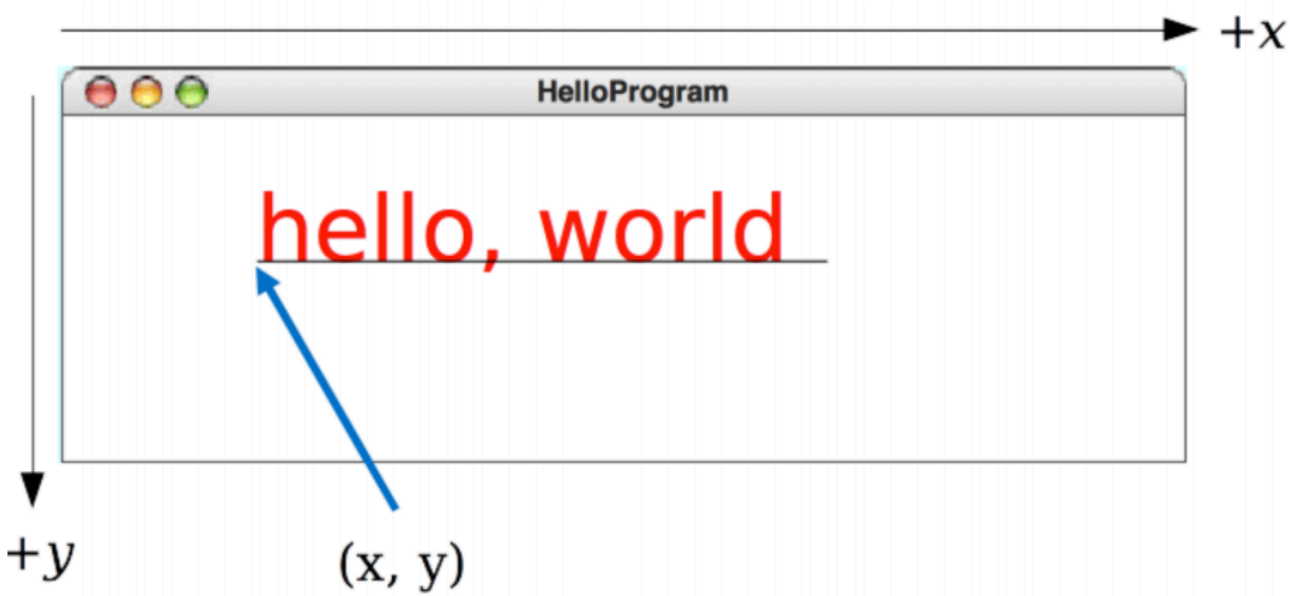
- Review concepts you're unsure of
- Review concepts from previous assignments
- Do section problems
- Do practice final under real conditions
- codestepbystep.com

Plan for today

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

Graphics

- Look at lecture slides for lists of different GObject types and their methods
- Remember: the x and y of GRect, GOval, etc. Is their **upper left corner**, but the x and y of GLabel is its **leftmost baseline coordinate**.
- Remember for labels: **getHeight() = getAscent() + getDescent()**.



Animation

Standard format for animation code:

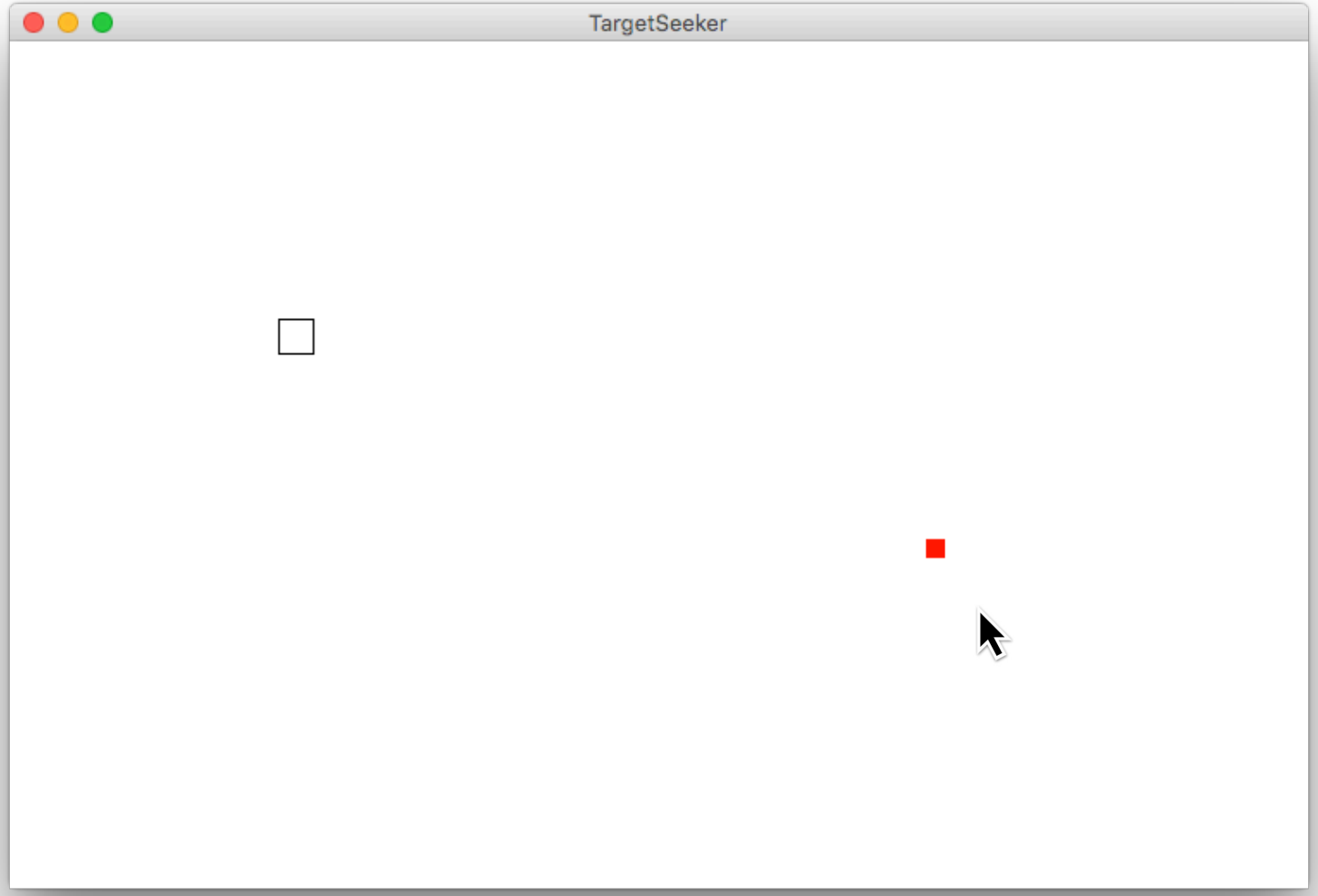
```
while (condition) {
      update graphics
      perform checks
      pause(PAUSE_TIME);
}
```

Events

- Two ways for Java to run your code: from run() and from event handlers (mouseClicked, mouseMoved, actionPerformed, etc.)
- Event handlers must have exactly the specified signature otherwise they won't work!

  e.g. **`public void mouseClicked(MouseEvent e)`**
- If you need to modify something in an event handler that you use elsewhere in your code, it should be an instance variable (e.g. paddle in Breakout!)

Demo: Seeker



Seeker

- Red square is the **target**
- Transparent square is the **seeker**
- The seeker should move towards and engulf the target
- Can change target location by clicking on the screen

```
/* Constants */
private static final int TARGET_SIZE = 10;
private static final int SEEKER_SIZE = 20;
private static final int PAUSE_TIME = 10;
```

Instance variables

```
/* Private instance variables */
private int targetMidX;
private int targetMidY;
private GRect targetSquare;
private GRect seeker;
```

run()

```
public void run() {
    initTarget();
    initSeeker();

    // Always keep seeking the target
    while (true) {
        seek();
        pause(PAUSE_TIME);
    }
}
```

run()

```
public void run() {
    initTarget();
    initSeeker();

    // Always keep seeking the target
    while (true) {
        seek();
        pause(PAUSE_TIME);
    }
}
```

initTarget() and initSeeker()

```
// Target is filled red square that starts in center
// of screen
private void initTarget() {
    targetSquare = new GRect(TARGET_SIZE, TARGET_SIZE);
    targetSquare.setColor(Color.RED);
    targetSquare.setFilled(true);
    targetMidX = getWidth() / 2;
    targetMidY = getHeight() / 2;
    add(targetSquare,
        targetMidX - TARGET_SIZE/2, targetMidY - TARGET_SIZE/2);
}
```

initSeeker()

```
// Seeker is unfilled black square that starts at origin
private void initSeeker() {
    seeker = new GRect(SEEKER_SIZE, SEEKER_SIZE);
    add(seeker, 0, 0);
}
```

run()

```
public void run() {
    initTarget();
    initSeeker();

    // Always keep seeking the target
    while (true) {
        seek();
        pause(PAUSE_TIME);
    }
}
```

seek()

```
// Seek target by taking a step toward its direction
private void seek() {
    // See if target is to left or right
    double seekerMidX = seeker.getX() + SEEKER_SIZE / 2;
    int dx = moveAmount(seekerMidX, targetMidX);

    // See if target is above or below
    double seekerMidY = seeker.getY() + SEEKER_SIZE / 2;
    int dy = moveAmount(seekerMidY, targetMidY);

    // move seeker toward target
    seeker.move(dx, dy);
}
```

moveAmount()

```
// Determine direction for seeker to move to get
// closer to targetPos
private int moveAmount(double seekerPos, double targetPos) {
    int amount = 0;
    if (targetPos > seekerPos) {
        amount = 1;
    } else if (targetPos < seekerPos) {
        amount = -1;
    }
    return amount;
}
```

mouseClicked()

```
// move center of target to position of mouse click
public void mouseClicked(MouseEvent e) {
    targetMidX = e.getX();
    targetMidY = e.getY();
    targetSquare.setLocation(targetMidX - TARGET_SIZE / 2,
                             targetMidY - TARGET_SIZE / 2);
}
```

Plan for today

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

1D Arrays

- An **array** is a fixed-length list of a single type of thing.
- An array can store **primitives** and **objects**.
- You cannot call methods on arrays i.e. no myArray.contains().
- Get the length by saying myArray.length. (No parentheses!)
- Print array with Arrays.toString(myArray), **not** println(myArray)!

```
[2, 4, 6, 8]      [I@4ddced80
```

Program traces

- Local variables are *separate* across methods
- Parameters are just assigned names by the order in which they're passed
- Draw changes to variables as you go through the program
- Objects vs primitive behavior for parameters

```
public void run() {
    int[] myArray = {5, 10, 15};
    int x = 5;
    foo(myArray, x);
    println(Arrays.toString(myArray) + " " + x);
}

private void foo(int[] anArray, int x) {
    anArray[1] = 20;
    x = 7;
}
```

A. Prints **[5, 10, 15] 3**

B. Prints **[5, 10, 15] 7**

C. Prints **[5, 20, 15] 5**

D. Prints **[5, 20, 15] 7**

Trace

```
public void run() {
    int[] a = {2, 0, 1};
    int b = 3;
    mystery(a, b, a[0]);
    println(Arrays.toString(a) + " " + b);

    b = a[0] + a[1] + a[2];
    mystery(a, a[1], a[2]);
    println(Arrays.toString(a) + " " + b);
}

public void mystery(int[] a, int b, int c) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
    b++;
    c--;
    println(Arrays.toString(a) + " " + b + " " + c);
}
```

Trace

```
public void run() {
    int[] a = {2, 0, 1};
    int b = 3;
    mystery(a, b, a[0]);
    println(Arrays.toString(a) + " " + b);

    b = a[0] + a[1] + a[2];
    mystery(a, a[1], a[2]);
    println(Arrays.toString(a) + " " + b);
}

public void mystery(int[] a, int b, int c) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
    b++;
    c--;
    println(Arrays.toString(a) + " " + b + " " + c);
}
```

Output:

```
[4, 0, 2] 4 1
[4, 0, 2] 3
[8, 0, 4] 1 1
[8, 0, 4] 6
```

Extra 1D Array problem

Write the method `int longestSortedSequence(int[] array)`

e.g. `int[] array = {3, 8, 10, 1, 9, 14, -3, 0, 14, 207, 56, 98, 12}`

Sorted in this case means nondecreasing, so a sequence could contain duplicates:

e.g. `int[] array = {17, 42, 3, 5, 5, 5, 8, 2, 4, 6, 1, 19}`

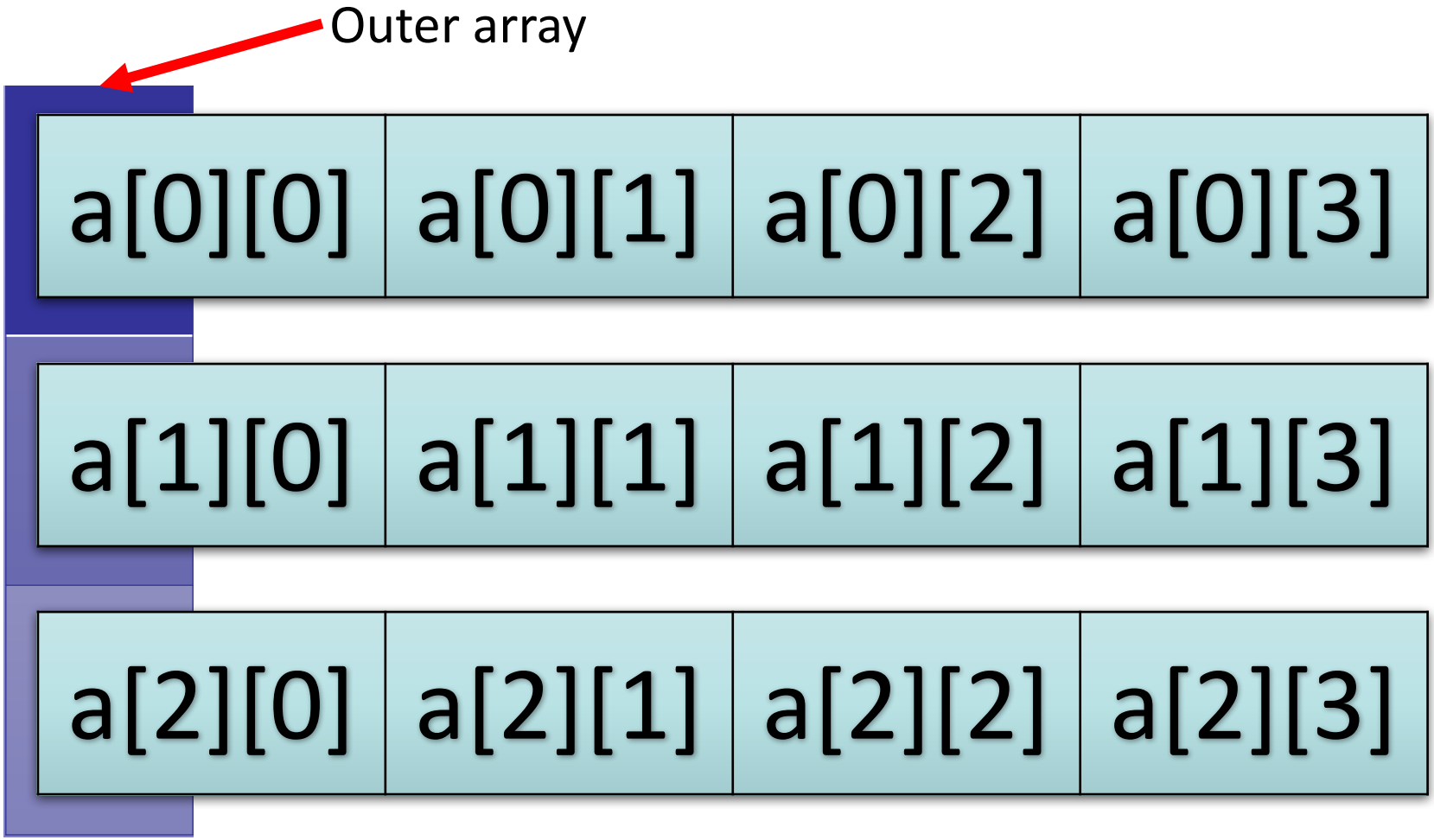
Link: http://www.codestepbystep.com/problem/view/java/arrays/longestSortedSequence

Plan for today

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

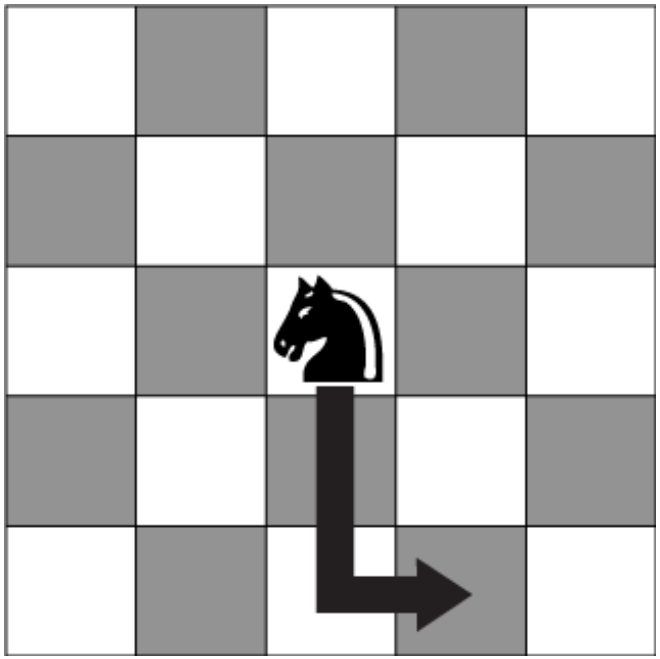
2D Arrays = Arrays of Arrays!

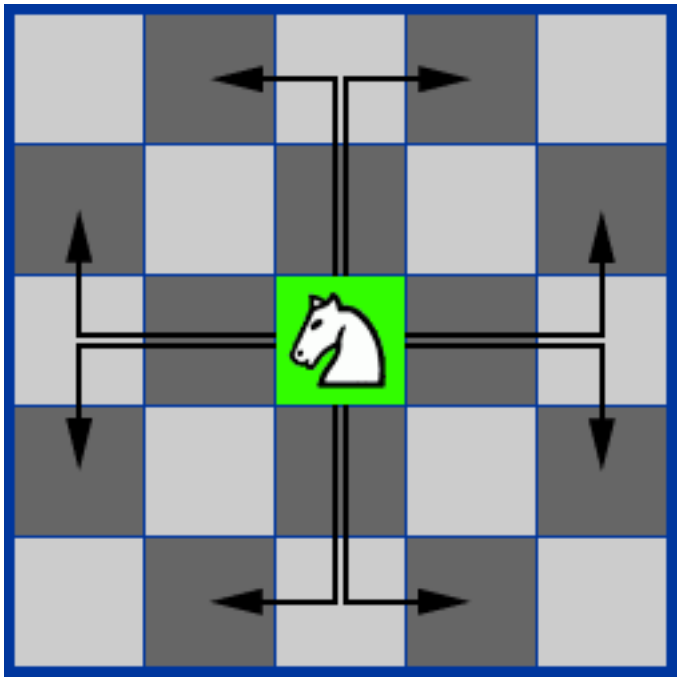
```
int[][] a = new int[3][4];
```



Chess

- Knight: moves in an ”L”-shape (two steps in one direction, one step in a perpendicular direction)





knightCanMove()

```
boolean knightCanMove(String[][] board,
                      int startRow, int startCol,
                      int endRow, int endCol)
```

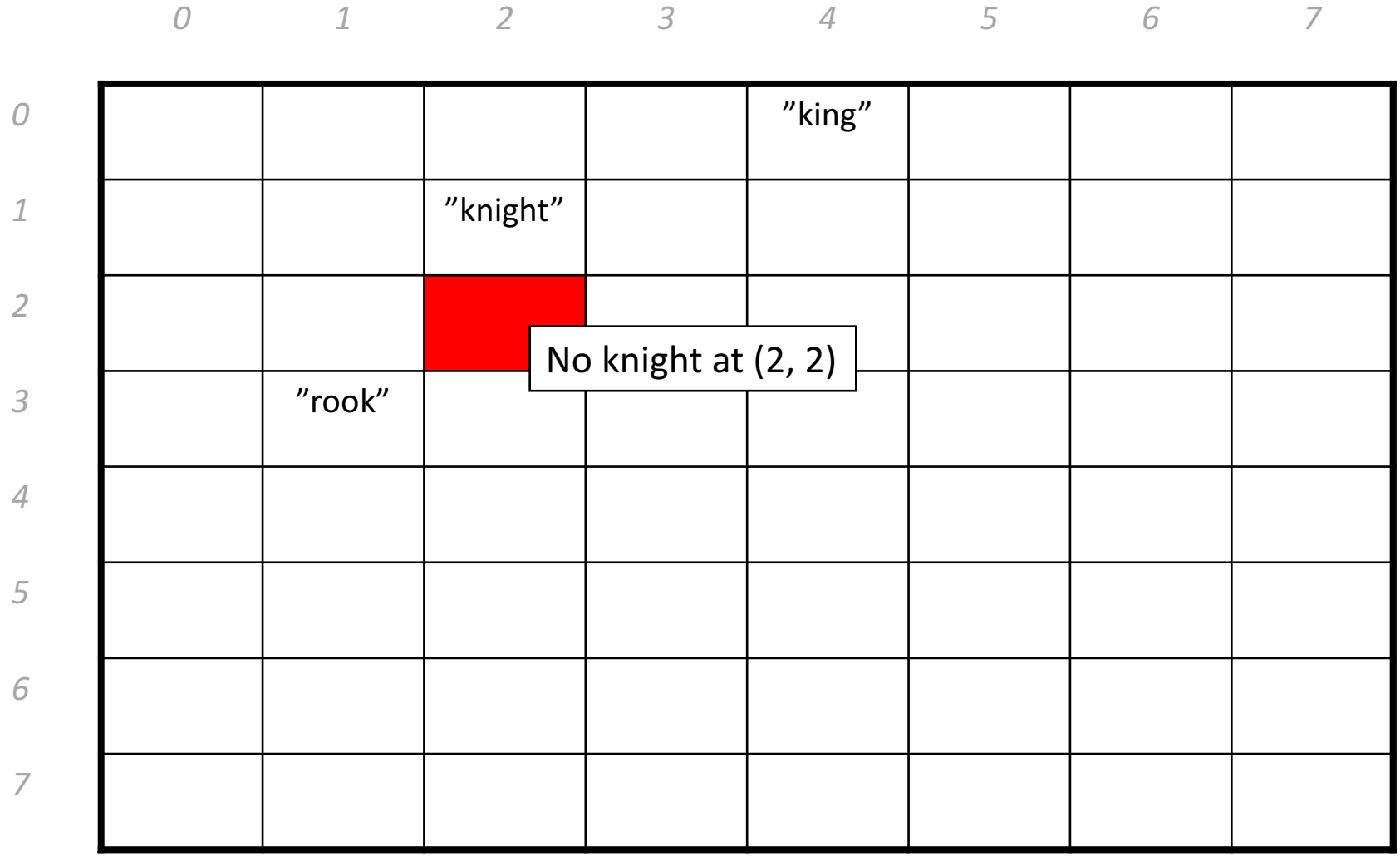
- (startRow, startCol) must contain a knight
- (endRow, endCol) must be empty
- (endRow, endCol) must be reachable from (startRow, startCol) in a single move
- Assume that (startRow, startCol) and (endRow, endCol) are within bounds of array

knightCanMove()

	0	1	2	3	4	5	6	7
0					”king”			
1			”knight”					
2								
3		”rook”						
4								
5								
6								
7								

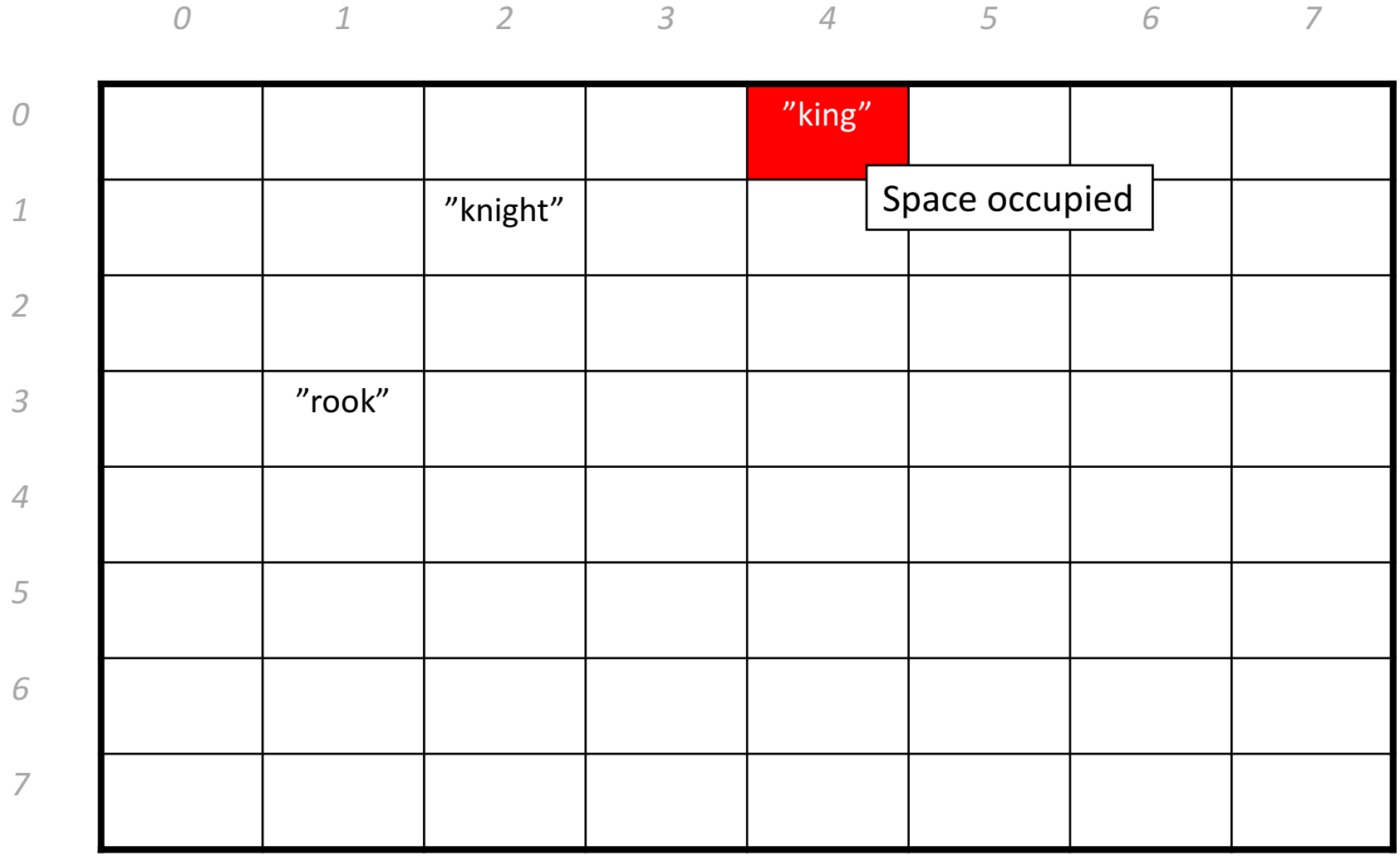
knightCanMove()

`knightCanMove(board, 2, 2, 3, 4)` returns **`false`**



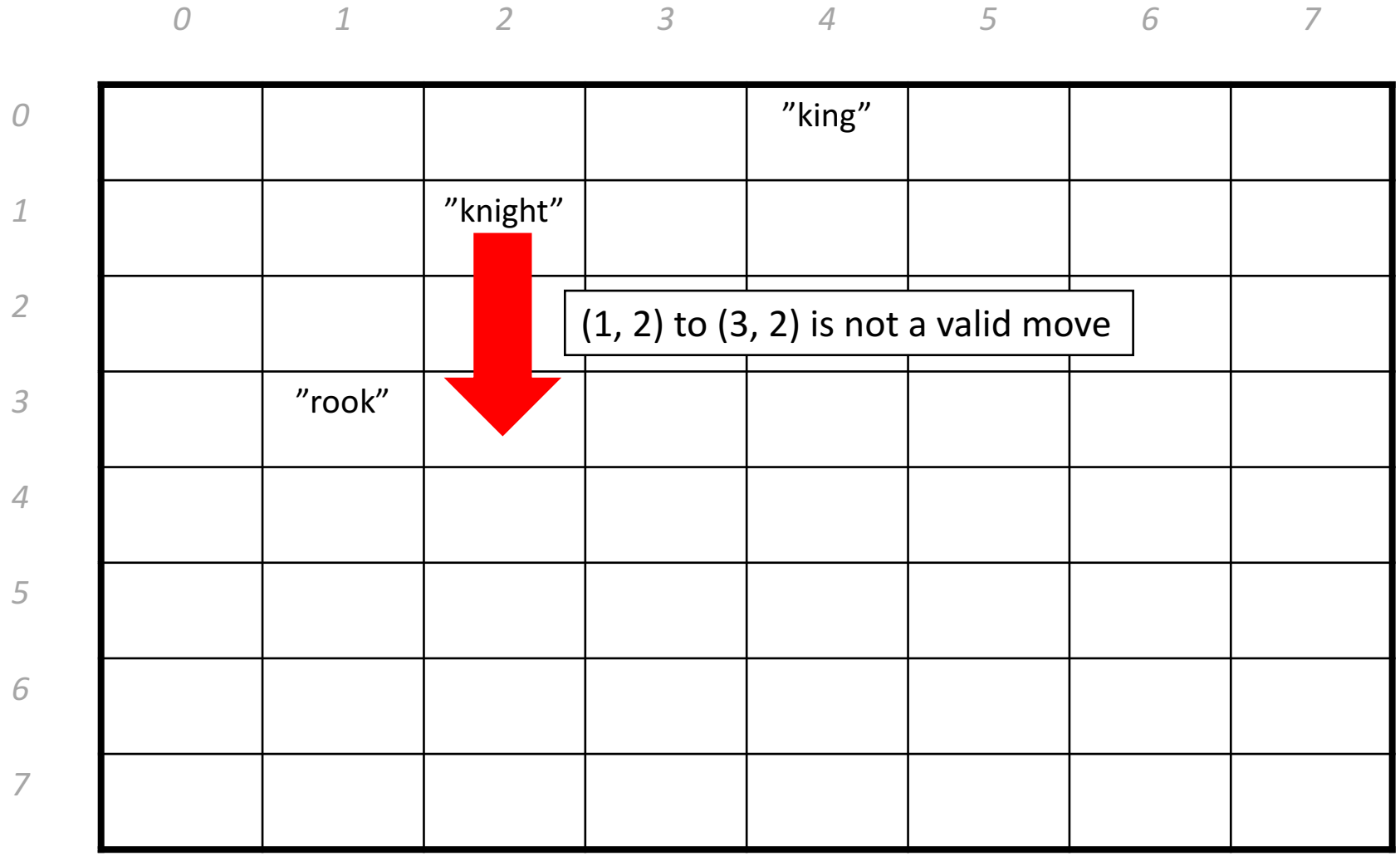
knightCanMove()

`knightCanMove(board, 1, 2, 0, 4)` returns **`false`**



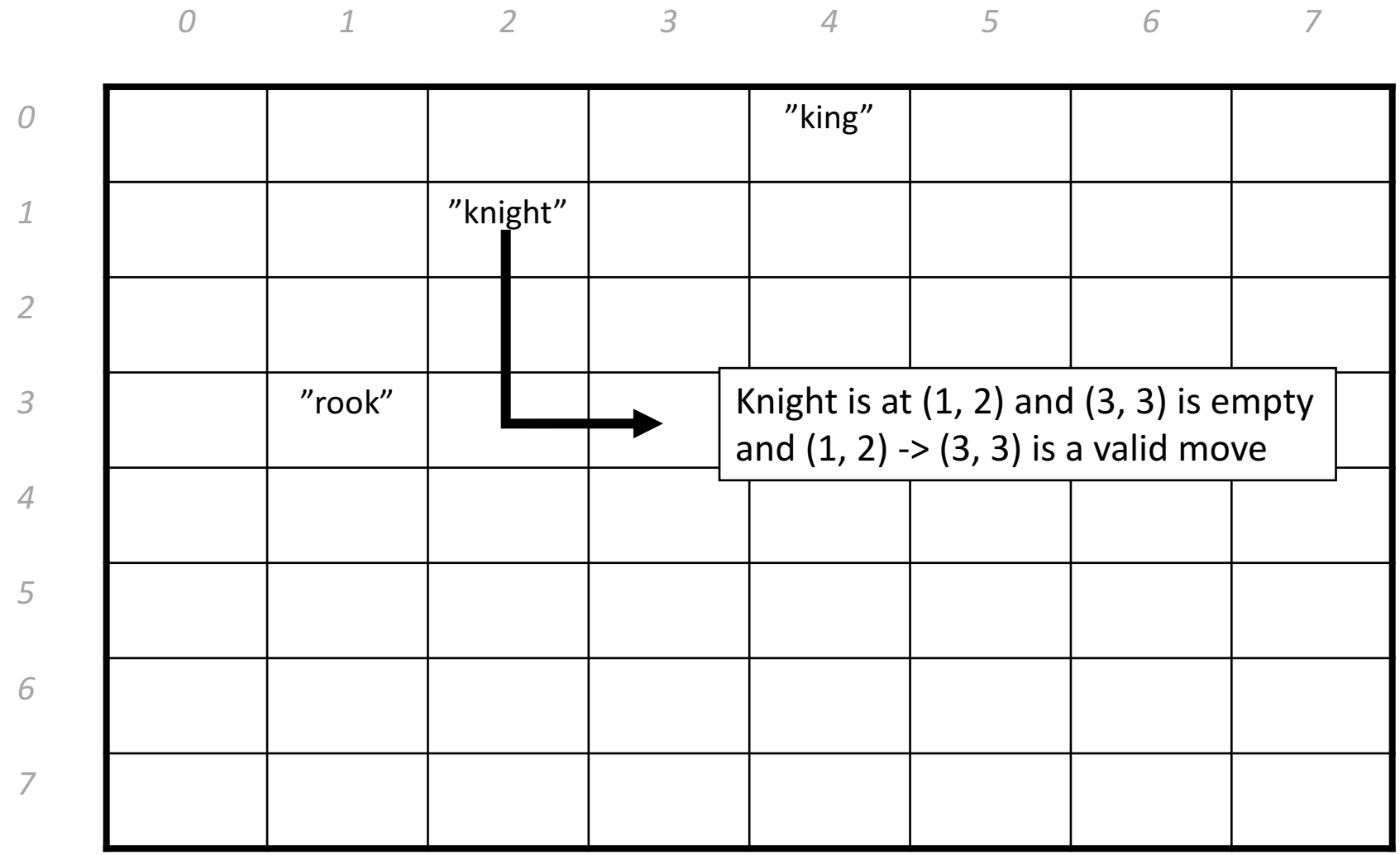
knightCanMove()

`knightCanMove(board, 1, 2, 3, 2)` returns **`false`**



knightCanMove()

`knightCanMove(board, 1, 2, 3, 3)` returns **`true`**



knightCanMove()

```
// This method returns true if the starting square contains a knight,
// the end square is empty, and the knight can legally move from the
// start square to the end square.
private boolean knightCanMove(String[][] board, int startRow,
                              int startCol, int endRow, int endCol) {



}
```

knightCanMove()

```
// This method returns true if the starting square contains a knight,
// the end square is empty, and the knight can legally move from the
// start square to the end square.
private boolean knightCanMove(String[][] board, int startRow,
                              int startCol, int endRow, int endCol) {
    if (board[startRow][startCol].equals("knight")) {





    }

}
```

knightCanMove()

```
// This method returns true if the starting square contains a knight,
// the end square is empty, and the knight can legally move from the
// start square to the end square.
private boolean knightCanMove(String[][] board, int startRow,
                              int startCol, int endRow, int endCol) {
    if (board[startRow][startCol].equals("knight")) {
        if (board[endRow][endCol].equals("")) {





        }
    }

}
```

knightCanMove()

```
// This method returns true if the starting square contains a knight,
// the end square is empty, and the knight can legally move from the
// start square to the end square.
private boolean knightCanMove(String[][] board, int startRow,
                              int startCol, int endRow, int endCol) {
    if (board[startRow][startCol].equals("knight")) {
        if (board[endRow][endCol].equals("")) {
            int rowDifference = Math.abs(startRow - endRow);
            int colDifference = Math.abs(startCol - endCol);
            if ((rowDifference == 1 && colDifference == 2) ||
                (rowDifference == 2 && colDifference == 1)) {
                return true;
            }
        }
    }

}
```

knightCanMove()

```
// This method returns true if the starting square contains a knight,
// the end square is empty, and the knight can legally move from the
// start square to the end square.
private boolean knightCanMove(String[][] board, int startRow,
                              int startCol, int endRow, int endCol) {
    if (board[startRow][startCol].equals("knight")) {
        if (board[endRow][endCol].equals("")) {
            int rowDifference = Math.abs(startRow - endRow);
            int colDifference = Math.abs(startCol - endCol);
            if ((rowDifference == 1 && colDifference == 2) ||
                (rowDifference == 2 && colDifference == 1)) {
                return true;
            }
        }
    }
    return false;
}
```

Plan for today

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

ArrayList

- An **ArrayList** is a flexible-length list of a single type of thing.
- An ArrayList can only store **objects**.
  - For primitives use e.g. **ArrayList<Integer>** instead of ArrayList<int>. (**Integer** is a wrapper class for int)
  - Other wrapper classes: **Double** instead of double, **Character** instead of char, **Boolean** instead of boolean.
- An ArrayList has a variety of methods you can use like *.contains, .get, .add, .remove, .size*, etc.

Array vs ArrayList

- Array
  - Fixed size
  - Efficient (not a concern in this class)
  - No methods, can only use myArray.length (no parentheses!)
  - Can store any object or primitive
- ArrayList
  - Expandable
  - Less efficient than Array (not a concern in this class)
  - Convenient methods like .add(), .remove(), .contains()
  - Cannot store primitives, so use their wrapper classes instead

deleteDuplicates()

```
private void deleteDuplicates(ArrayList<String> list)
```

- Guaranteed that list is in sorted order
- {"be", "be", "is", "not", "or", "question", "that", "the", "to", "to"} becomes {“be”, “is”, “not”, “or”, “question”, “that”, “the”, “to”}

- Solution strategy:
  - Loop through ArrayList
  - Compare pairs of elements
  - If element.equals(nextElement), remove element from the list

deleteDuplicates

- Loop through ArrayList
- Compare pairs of elements
- If element.equals(nextElement), remove element from the list

```
private void deleteDuplicates(ArrayList<String> list) {
    for (int i = 0; i < list.size() - 1; i++) {
        String elem = list.get(i);
        // If two adjacent elements are equal
        if (list.get(i + 1).equals(elem)) {
            list.remove(i);
            i--;
        }
    }
}
```

deleteDuplicatesReverse

- Loop through ArrayList **in reverse**
- Compare pairs of elements
- If element.equals(**previous**Element), remove element from the list

```
private void deleteDuplicatesReverse(ArrayList<String> list) {
    for (int i = list.size() - 1; i > 0; i--) {
        String elem = list.get(i);
        // If two adjacent elements are equal
        if (list.get(i - 1).equals(elem)) {
            list.remove(i);
        }
    }
}
```

Recap

- Announcements/Exam logistics
- Graphics, Animation, Events
- 1D Arrays
- 2D Arrays
- ArrayList

Next time: Final Exam Review 2