CS 106A, Lecture 27
Final Exam Review 1

Plan for today

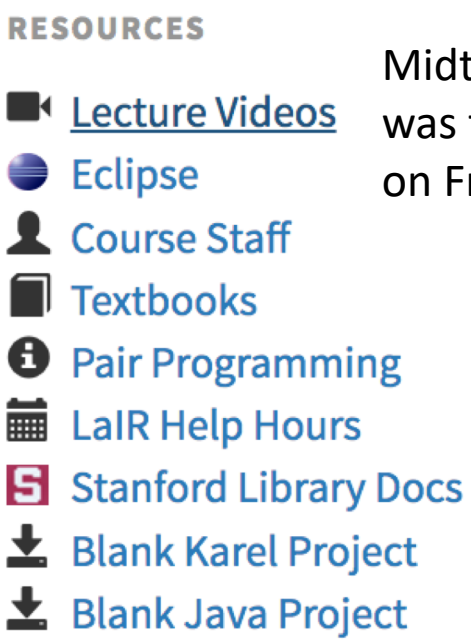
- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

Plan for today

- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

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



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 programs we wrote in lecture
- Do section problems
- Do practice final under real conditions
- codestepbystep.com

- Colin's secret test-taking strategy:
  - Using BlueBook's timer, give yourself 3-5 minutes to read and start writing pseudocode for each problem
  - Once you've thought about every problem, go back to the one that seemed easiest and start coding for real
  - This is not about finishing every problem; it is about collecting points

Plan for today

- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

Learning Goals

"After this lecture, I want you to be able to…"

- Lectures 1-3 (Karel): Apply programmatic thinking and decomposition to logical tasks
- Lecture 4 (Intro to Java): Create variables of primitive types, perform console I/O, and evaluate expressions using primitive types
- Lecture 5 (Booleans and Control Flow): Use loops to perform repeated tasks, use conditions to decide which tasks to perform

Learning Goals

"After this lecture, I want you to be able to…"

- Lecture 6 (Scope): Identify a variable's scope
- Lecture 7 (Parameters and Return): Write functions that pass parameters and leverage return values to overcome the limitation of scope in program decomposition

Learning Goals

"After this lecture, I want you to be able to…"

- Lecture 8 (Characters and Strings): Use randomness to write interesting programs, recall that Java understands chars as ASCII values (ints from 0 - 255), create String variables, recall that Strings are immutable

Learning Goals

"After this lecture, I want you to be able to…"

- Lecture 9 (Problem-Solving with Strings): Identify situations where common String methods like length and substring are useful, solve problems that involve manipulating Strings (often through creating new Strings)
- Lecture 10 (File Reading): Write programs that use files as sources of input data

Learning Goals

"After this lecture, I want you to be able to…"

- Lectures 11 and 12 (Graphics): Write programs using five types of graphical objects (rectangles, ovals, lines, labels, and images), call methods on Objects
- Lecture 13 (Animation): Use loops and pausing to animate graphical programs

Learning Goals

"After this lecture, I want you to be able to…"

- Lectures 14 (Events): Write programs that respond to mouse events, identify when it is appropriate to use instance variables
- Lecture 15 (Memory): Recall that primitives are passed by value while Objects are passed by reference in Java, apply that knowledge to know which variables' values change when they are modified in other methods

Learning Goals

"After this lecture, I want you to be able to…"

- Lectures 16 (Arrays): Describe the purpose of data structures in programming, know how to store data in and retrieve data from arrays
- Lecture 17 (2D Arrays): Recognize 2D arrays as grids or arrays of arrays, apply nested for loops to working with 2D arrays
- Lecture 18 (More Arrays): Identify uses for arrays in writing complex programs

Learning Goals

"After this lecture, I want you to be able to…"

- Lectures 19 (ArrayLists): Know how to store data in and retrieve data from ArrayLists
- Lecture 20 (ArrayLists and HashMaps): Know how to store data in and retrieve data from HashMaps, identify the most appropriate data structure between arrays, ArrayLists, and HashMaps for different storage needs

Learning Goals

"After this lecture, I want you to be able to…"

- Lectures 23 (Interactors and GCanvas): Know how to create graphical user interfaces (GUIs) with Java's interactive components
- Lecture 24 (GCanvas): Write richer graphical programs leveraging multiple classes
- Lectures 24-26 (BiasBars, Life After CS106A): Identify real-world challenges where 106A-level programming knowledge can help

Learning Goals

- Assignments gave you practice synthesizing lots of different topics from lecture
- Exams assess the extent to which you are able to recall and synthesize learning goals
  - Because exams are high-pressure, timed situations, you don't need to score spectacularly for me to believe that you understand the course's material

Plan for today

- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

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**

Animation

Standard format for animation code:

```
while (condition) {
      update graphics
      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 access to a variable in an event handler that you use elsewhere in your code, it should be an instance variable (e.g., paddle in Breakout)

Plan for today

- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

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, e.g., 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
```

1D Array Practice

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

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

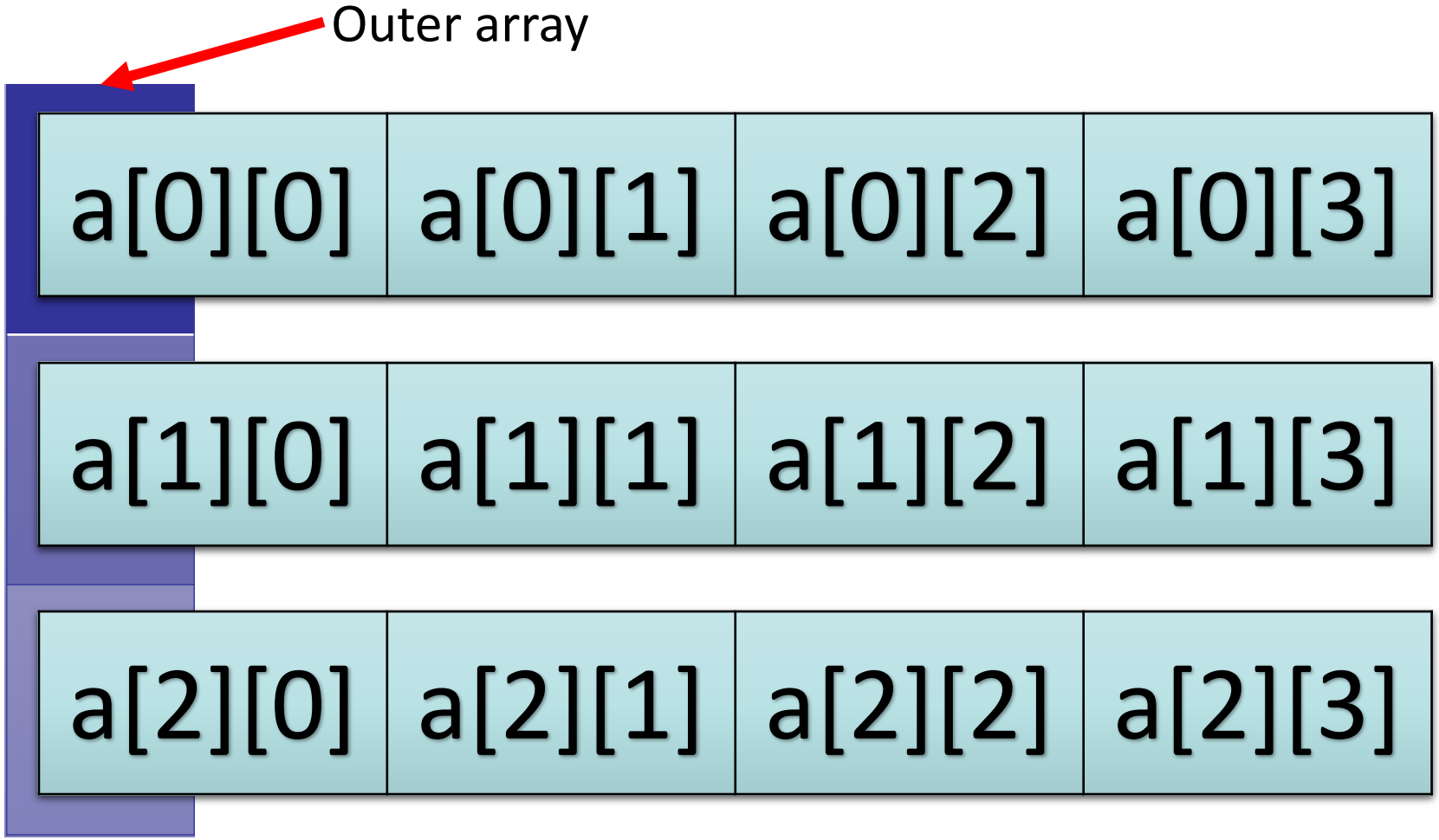
Sorted in this case means nondecreasing, so a sequence could contain duplicates:

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

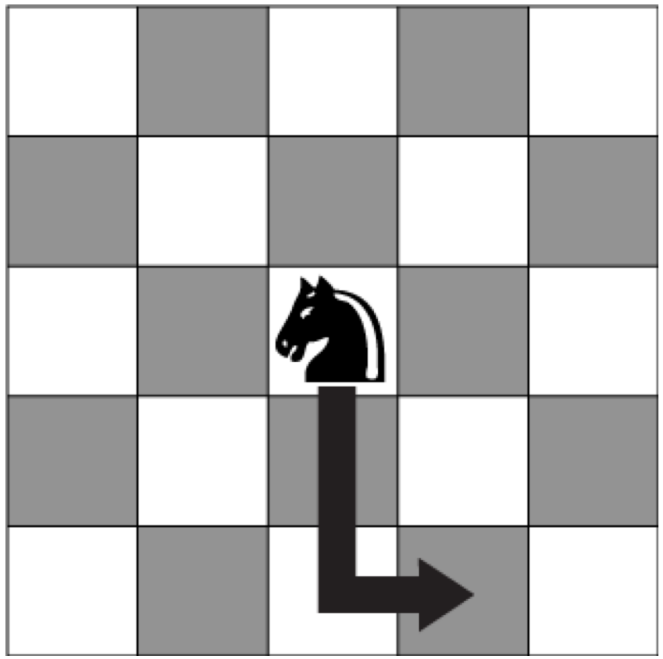
2D Arrays = Arrays of Arrays!

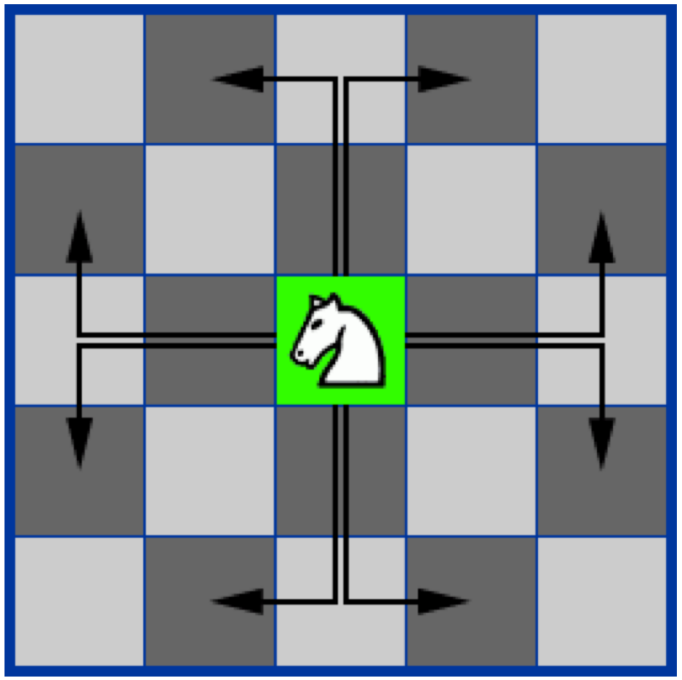
```
int[][] a = new int[3][4];
```



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)
```

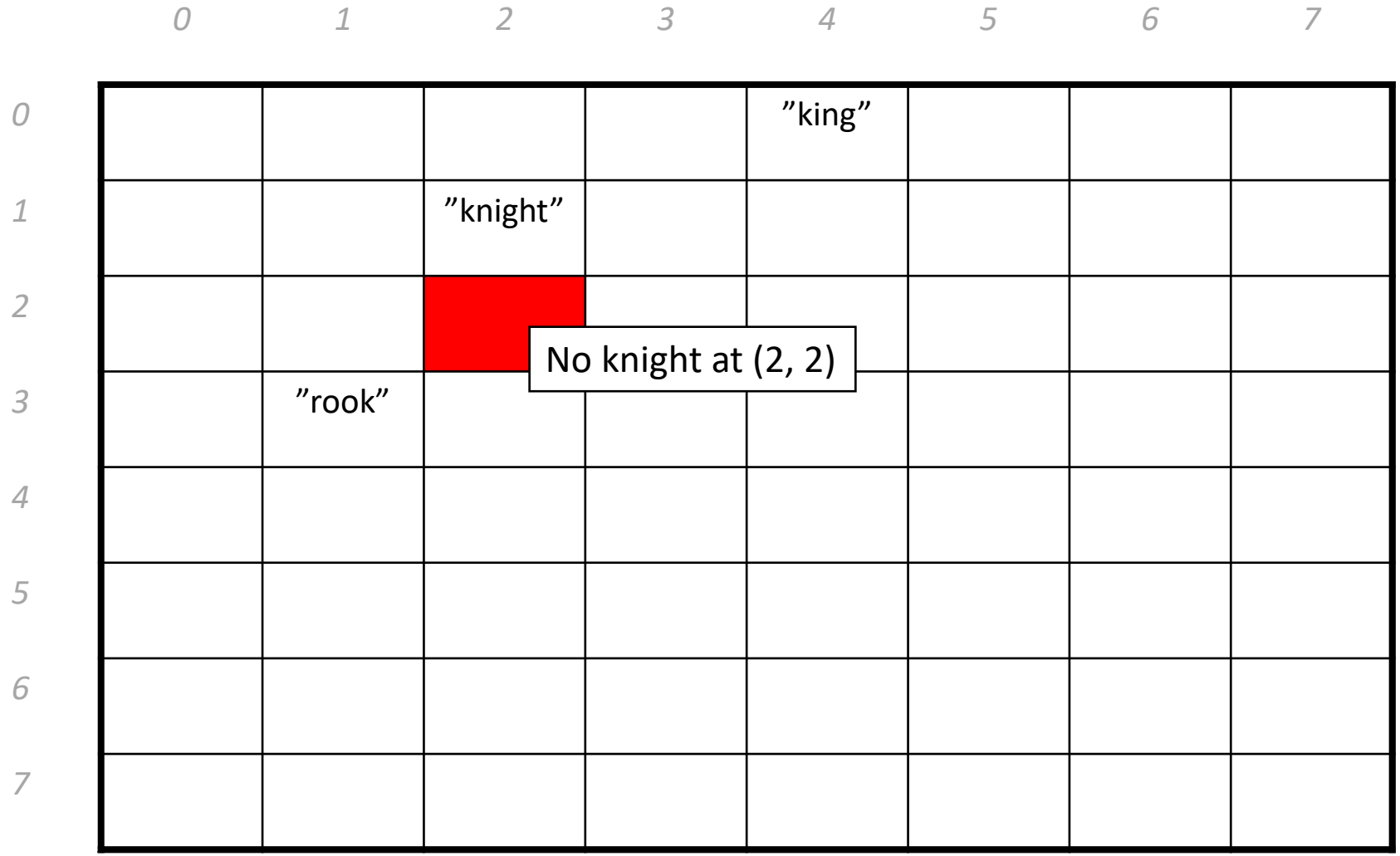
- (startRow, startCol) must contain a knight
- (endRow, endCol) must be empty
- (endRow, endCol) must be reachable from (startRow, startCol) in a single move
- Assume that (startRow, startCol) and (endRow, endCol) are within bounds of array

knightCanMove()

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

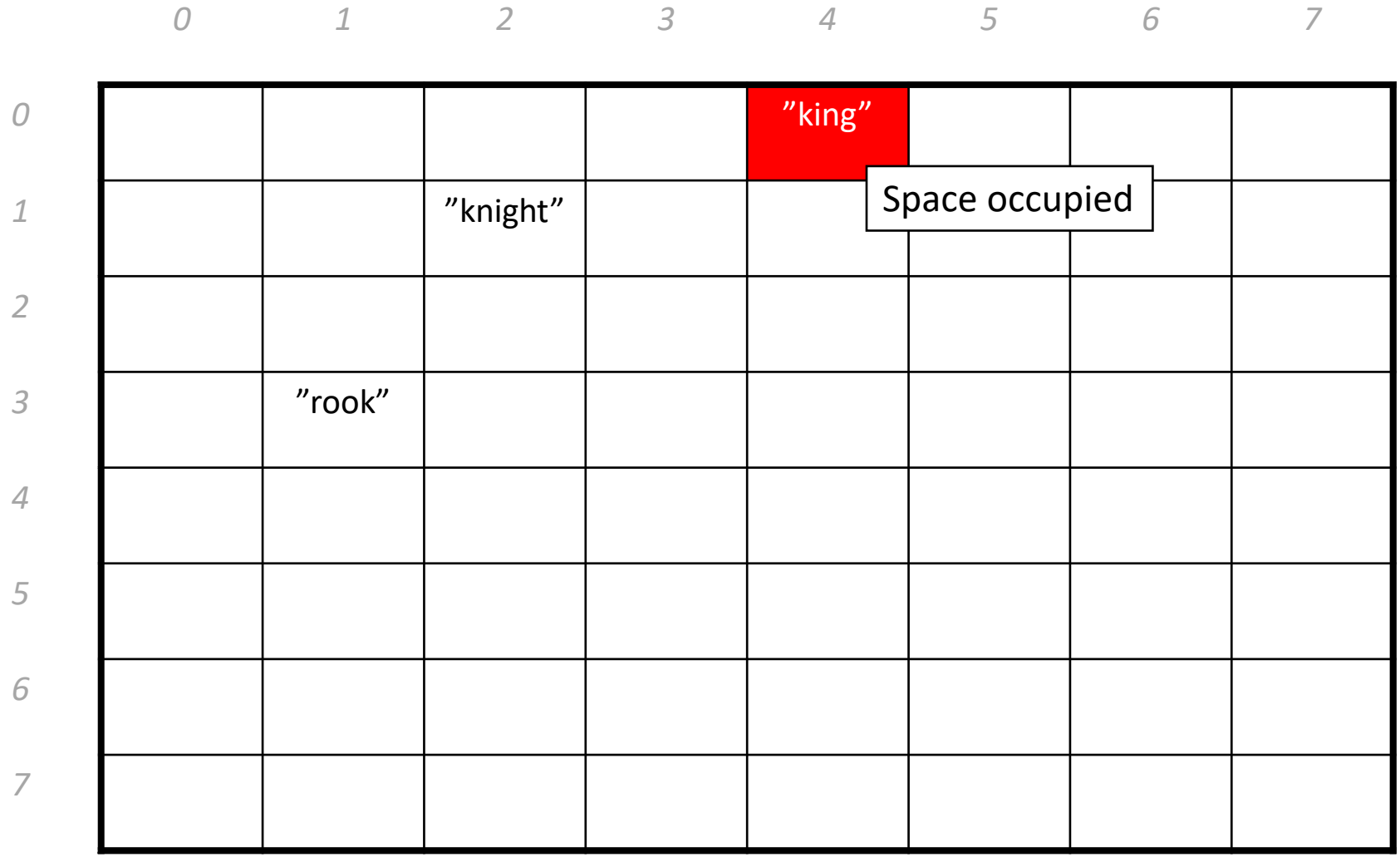
knightCanMove()

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



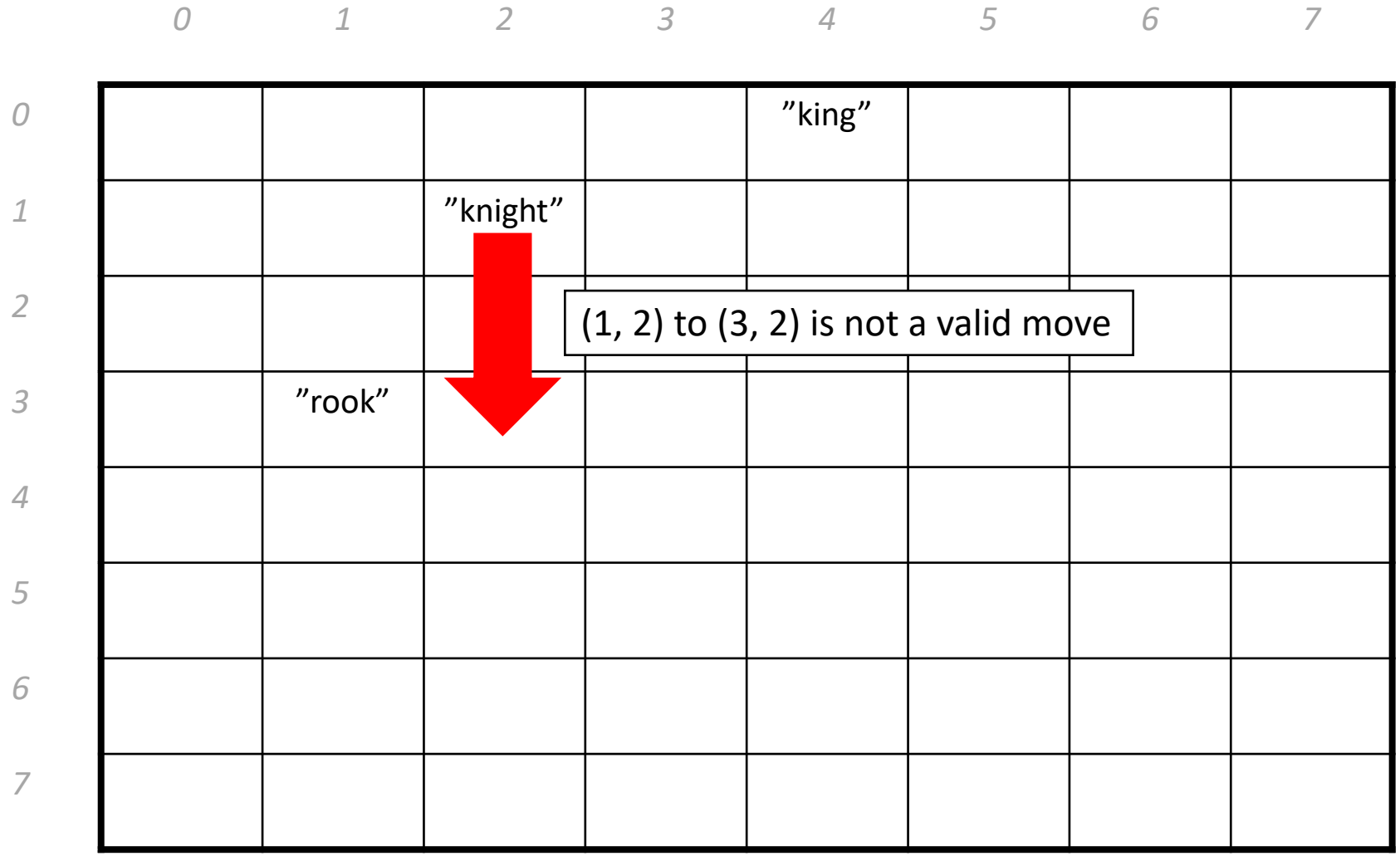
knightCanMove()

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



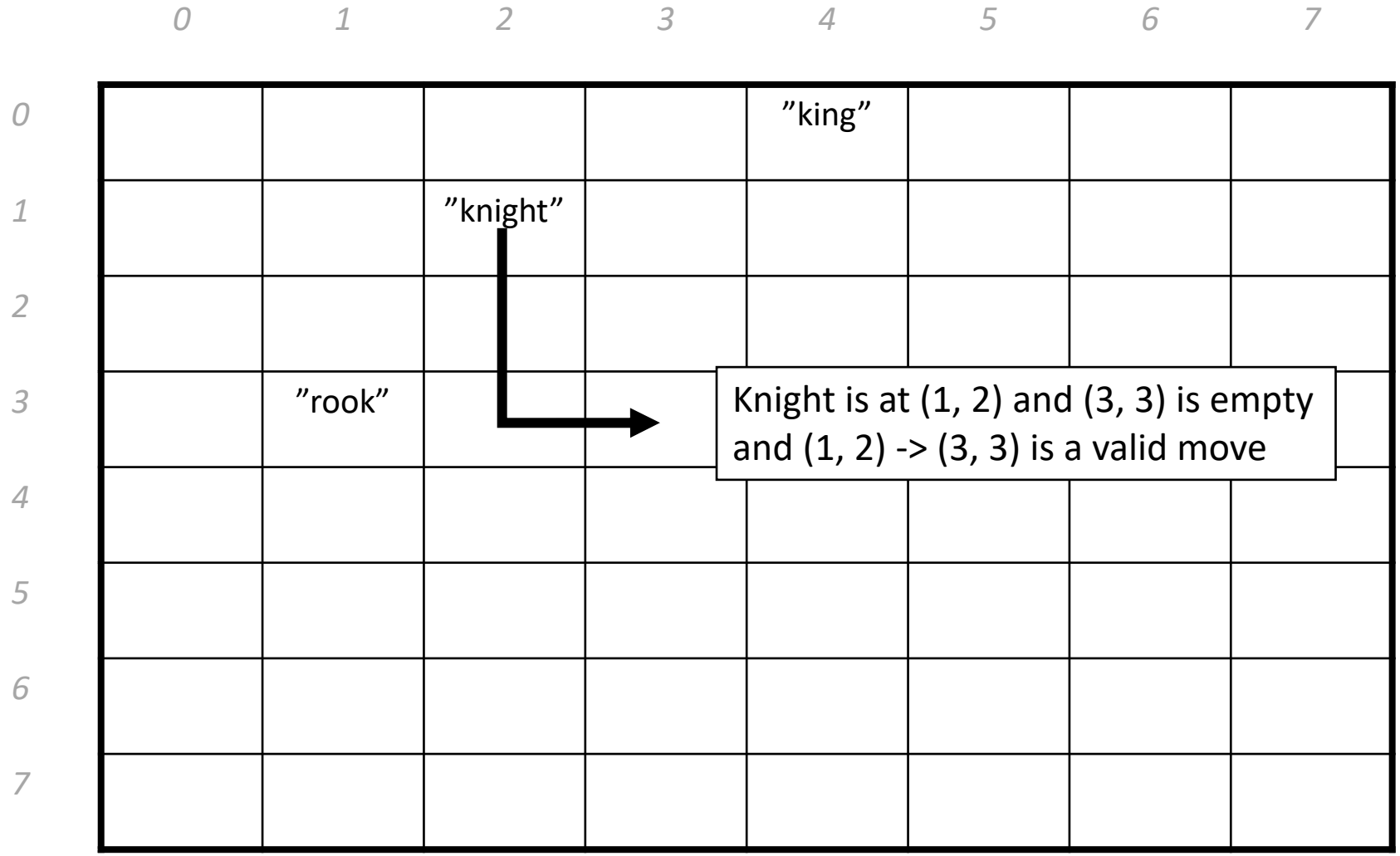
knightCanMove()

`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
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

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);
        }
    }
}
```

Plan for today

- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

Review: HashMaps

- A variable type that represents a collection of **key-value pairs**
- You access values by *key,* and all keys are unique
- Keys and values can be any type of **Object** (use wrapper classes to store primitives)
- Resizable – can add and remove pairs
- Has a variety of methods you can use, including *.containsKey*, *.put*, *.get*, *.keySet*, etc.

HashMap Examples

- **Phone book:** name -> phone number
- **Search engine:** URL -> webpage
- **Dictionary**: word -> definition
- **Bank**: account # -> balance
- **Social Network**: name -> profile
- **Counter**: text -> # occurrences
- And many more…

Review: HashMap Operations

- *m*.put(*key*, *value*); Adds a key/value pair to the map.

```
m.put("Eric", "650-123-4567");
```

  - Replaces any previous value for that key.

- *m*.get(*key*) Returns the value paired with the given key.

```
String phoneNum = m.get("Jenny");  // "867-5309"
```

  - Returns null if the key is not found.

- **m.remove**(*key*); Removes the given key and its paired value.

```
m.remove(“Annie”);
```

  - Has no effect if the key is not in the map.

key	value
"Jenny”	→ "867-5309"
"Mehran"	→ "123-4567"
"Marty"	→ "685-2181"
"Chris”	→ "947-2176"

Review: HashMap Operations

- *m*.containsKey(*key*); Returns true if the key is in the map, false otherwise

- *m*.size(); Returns the number of key/value pairs in the map.

- To iterate over a map:

```
for (KeyType key : map.keySet()) {
    ValueType value = map.get(key);
    // Do something with key and/or value
}
```

What data structure should I use?

- Use an **array** if…
  - Order matters for your information
  - You know how many elements you will store
  - You need the most efficiency
- Use an **ArrayList** if…
  - Order matters for your information
  - You do not know how many elements you will store, or need to resize
  - You need to use ArrayList methods
- Use a **HashMap** if…
  - Order doesn't matter for your information
  - You need to store an *association* between two types of information
  - You do not know how many elements you will store, or need to resize
  - You need to use HashMap methods

Practice: Anagrams

- Write a program to find all **anagrams** of a word the user types.

```
Type a word [Enter to quit]: scared
Anagrams of scared:
cadres cedars sacred scared
```

- Assume you are given the following:
  - A **`dictionary.txt`** file containing words in the dictionary
  - A method **`private String sortLetters(String s)`** method that takes a string and returns the string with its characters alphabetically ordered.

- How can a HashMap help us solve this problem?

Key Idea: Anagrams

- Every word has a *sorted form* where its letters are arranged into alphabetical order.

  `"fare"` → `"aefr"`

  `"fear"` → `"aefr"`

  `"swell"` → `"ellsw"`

  `"wells"` → `"ellsw"`

- Notice that anagrams have the same **sorted form** as each other.

Anagrams Solution

```
public void run() {
        HashMap<String, ArrayList<String>> anagrams =
                createAnagramsMap();
        // prompt user for words and look up anagrams in map
        String word = readLine("Type a word [Enter to quit]: ");
        while (word.length() > 0) {
            String sorted = sortLetters(word.toLowerCase());
            if (anagrams.containsKey(sorted)) {
                println("Anagrams of " + word + ":");
                println(anagrams.get(sorted));
            } else {
                println("No anagrams for " + word + ".");
            }
            word = readLine("Type a word [Enter to quit]: ");
        }
}
```

Anagrams Solution

```
public void run() {
        HashMap<String, ArrayList<String>> anagrams =
                createAnagramsMap();
        // prompt user for words and look up anagrams in map
        String word = readLine("Type a word [Enter to quit]: ");
        while (word.length() > 0) {
            String sorted = sortLetters(word.toLowerCase());
            if (anagrams.containsKey(sorted)) {
                println("Anagrams of " + word + ":");
                println(anagrams.get(sorted));
            } else {
                println("No anagrams for " + word + ".");
            }
            word = readLine("Type a word [Enter to quit]: ");
        }
}
```

Anagrams Solution

```
public void run() {
        HashMap<String, ArrayList<String>> anagrams =
                createAnagramsMap();
        // prompt user for words and look up anagrams in map
        String word = readLine("Type a word [Enter to quit]: ");
        while (word.length() > 0) {
            String sorted = sortLetters(word.toLowerCase());
            if (anagrams.containsKey(sorted)) {
                println("Anagrams of " + word + ":");
                println(anagrams.get(sorted));
            } else {
                println("No anagrams for " + word + ".");
            }
            word = readLine("Type a word [Enter to quit]: ");
        }
}
```

Anagrams Solution

```
public void run() {
        HashMap<String, ArrayList<String>> anagrams =
                createAnagramsMap();
        // prompt user for words and look up anagrams in map
        String word = readLine("Type a word [Enter to quit]: ");
        while (word.length() > 0) {
            String sorted = sortLetters(word.toLowerCase());
            if (anagrams.containsKey(sorted)) {
                println("Anagrams of " + word + ":");
                println(anagrams.get(sorted));
            } else {
                println("No anagrams for " + word + ".");
            }
            word = readLine("Type a word [Enter to quit]: ");
        }
}
```

Anagrams Solution

```
public void run() {
        HashMap<String, ArrayList<String>> anagrams =
                createAnagramsMap();
        // prompt user for words and look up anagrams in map
        String word = readLine("Type a word [Enter to quit]: ");
        while (word.length() > 0) {
            String sorted = sortLetters(word.toLowerCase());
            if (anagrams.containsKey(sorted)) {
                println("Anagrams of " + word + ":");
                println(anagrams.get(sorted));
            } else {
                println("No anagrams for " + word + ".");
            }
            word = readLine("Type a word [Enter to quit]: ");
        }
}
```

Anagrams Solution

```
public void run() {
        HashMap<String, ArrayList<String>> anagrams =
                createAnagramsMap();
        // prompt user for words and look up anagrams in map
        String word = readLine("Type a word [Enter to quit]: ");
        while (word.length() > 0) {
            String sorted = sortLetters(word.toLowerCase());
            if (anagrams.containsKey(sorted)) {
                println("Anagrams of " + word + ":");
                println(anagrams.get(sorted));
            } else {
                println("No anagrams for " + word + ".");
            }
            word = readLine("Type a word [Enter to quit]: ");
        }
}
```

Anagrams Solution, Part 2

```
// Returns a new map from a sorted word to all words created
// from those letters - e.g. “acers” -> {“scare”, “cares”,...}
private HashMap<String, ArrayList<String>> createAnagramsMap() {
        HashMap<String, ArrayList<String>> anagrams =
                new HashMap<>();
        try {
                Scanner scanner =
                        new Scanner(new File("res/dictionary.txt"));
                while (scanner.hasNext()) {
                        String word = scanner.next();
                        String sorted = sortLetters(word);
                        ...
```

Anagrams Solution, Part 2

```
// Returns a new map from a sorted word to all words created
// from those letters - e.g. “acers” -> {“scare”, “cares”,...}
private HashMap<String, ArrayList<String>> createAnagramsMap() {
        HashMap<String, ArrayList<String>> anagrams =
                new HashMap<>();
        try {
                Scanner scanner =
                        new Scanner(new File("res/dictionary.txt"));
                while (scanner.hasNext()) {
                        String word = scanner.next();
                        String sorted = sortLetters(word);
                        ...
```

Anagrams Solution, Part 2

```
// Returns a new map from a sorted word to all words created
// from those letters - e.g. “acers” -> {“scare”, “cares”,...}
private HashMap<String, ArrayList<String>> createAnagramsMap() {
        HashMap<String, ArrayList<String>> anagrams =
                new HashMap<>();
        try {
                Scanner scanner =
                        new Scanner(new File("res/dictionary.txt"));
                while (scanner.hasNext()) {
                        String word = scanner.next();
                        String sorted = sortLetters(word);
                        ...
```

Anagrams Solution, Part 2

```
// Returns a new map from a sorted word to all words created
// from those letters - e.g. “acers” -> {“scare”, “cares”,...}
private HashMap<String, ArrayList<String>> createAnagramsMap() {
        HashMap<String, ArrayList<String>> anagrams =
                new HashMap<>();
        try {
                Scanner scanner =
                        new Scanner(new File("res/dictionary.txt"));
                while (scanner.hasNext()) {
                        String word = scanner.next();
                        String sorted = sortLetters(word);
                        ...
```

Anagrams Solution, Part 2

```
// Returns a new map from a sorted word to all words created
// from those letters - e.g. “acers” -> {“scare”, “cares”,...}
private HashMap<String, ArrayList<String>> createAnagramsMap() {
        HashMap<String, ArrayList<String>> anagrams =
                new HashMap<>();
        try {
                Scanner scanner =
                        new Scanner(new File("res/dictionary.txt"));
                while (scanner.hasNext()) {
                        String word = scanner.next();
                        String sorted = sortLetters(word);
                        ...
```

Anagrams Solution, Part 2

```
                ArrayList<String> words;
                if (anagrams.containsKey(sorted)) {
                        words = anagrams.get(sorted);
                } else {
                        words = new ArrayList<>();
                }

                words.add(word);
                anagrams.put(sorted, words);
        }
        scanner.close();
} catch (IOException ex) {
        println("Error reading file.");
}
return anagrams;
}
```

Anagrams Solution, Part 2

```
                ArrayList<String> words;
                if (anagrams.containsKey(sorted)) {
                        words = anagrams.get(sorted);
                } else {
                        words = new ArrayList<>();
                }

                words.add(word);
                anagrams.put(sorted, words);
        }
        scanner.close();
} catch (IOException ex) {
        println("Error reading file.");
}
return anagrams;
}
```

Anagrams Solution, Part 2

```
                ArrayList<String> words;
                if (anagrams.containsKey(sorted)) {
                        words = anagrams.get(sorted);
                } else {
                        words = new ArrayList<>();
                }

                words.add(word);
                anagrams.put(sorted, words);
        }
        scanner.close();
} catch (IOException ex) {
        println("Error reading file.");
}
return anagrams;
}
```

Anagrams Solution, Part 2

```
                ArrayList<String> words;
                if (anagrams.containsKey(sorted)) {
                        words = anagrams.get(sorted);
                } else {
                        words = new ArrayList<>();
                }

                words.add(word);
                anagrams.put(sorted, words);
        }
        scanner.close();
} catch (IOException ex) {
        println("Error reading file.");
}
return anagrams;
}
```

Anagrams Solution, Part 2

```
                ArrayList<String> words;
                if (anagrams.containsKey(sorted)) {
                        words = anagrams.get(sorted);
                } else {
                        words = new ArrayList<>();
                }

                words.add(word);
                anagrams.put(sorted, words);
        }
        scanner.close();
} catch (IOException ex) {
        println("Error reading file.");
}
return anagrams;
}
```

Recap

- Announcements/Exam logistics
- Learning Goals
- Graphics, Animation, Events
- Arrays
- ArrayLists
- HashMaps

Next time: Final Exam Review 2