COS126 Princeton University **Programming Exam** Spring 2025

Before the exam. Read this page of instructions before the exam begins.
☞ Do not start the exam (or read the next page) until instructed to do so.

Duration. Once the exam begins, you have **80** minutes to complete it.

Submission. Submit your solutions on TigerFile using the link from the Exams page.
☞ You may submit multiple times (but only the last version will be graded).

Check Submitted Files. You may click the ***Check Submitted Files*** button to receive partial feedback on your submission. We will attempt to provide this feature during the exam, but do not rely upon it.

Grading. Your program will be graded *primarily* on correctness. However, efficiency and clarity will also be considered. You will receive partial credit for a program that implements some of the required functionality. You will receive a substantial penalty for a program that does not compile.

Allowed resources. During the exam you may use **only** the following resources: course textbook, companion booksite, lectures, course website (which includes past exams and solutions), course Ed Discussion and Ed Lessons, your course notes, jshell / jshell-introcs, and your code from the programming assignments and precepts. You **may not use** outside sources such as StackOverflow, Google, Google Docs, ChatGPT, etc.

No collaboration or communication. Collaboration and communication during this exam are prohibited, except with course staff. Staff will be outside the exam room to answer clarification questions.

No electronic devices or software. Software and computational/communication devices (phones, ipads, etc.) are prohibited, except to the extent needed for taking this exam (such as a laptop, browser, and IntelliJ). For example, you must close all unnecessary applications and browser tabs; disable notifications; and turn off your cell phone.

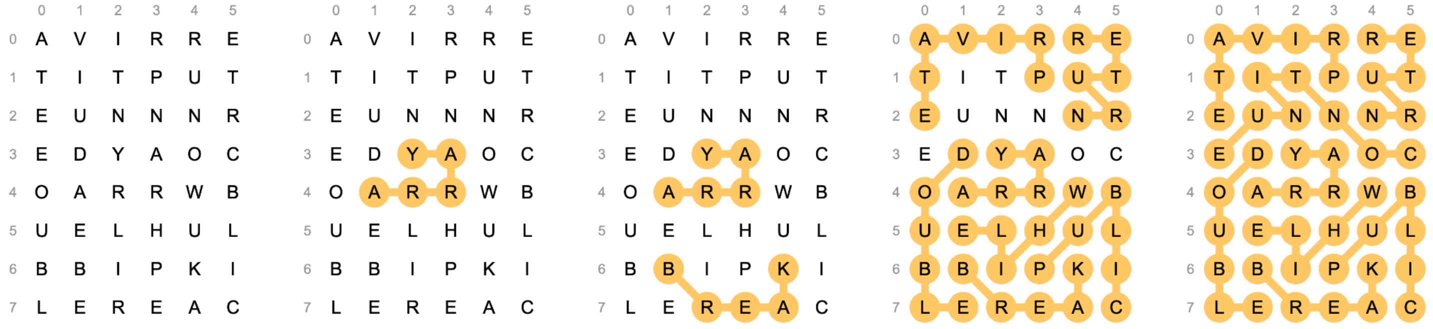
Honor Code pledge. Write and sign the Honor Code pledge by typing the text below in the file **`acknowledgments.txt`**. Submit to TigerFile.

> *I pledge my honor that I have not violated the Honor Code during this examination.*
>
> Electronically sign it by typing **`/s/`** followed by your name.

After the exam. Discussing or communicating the contents of this exam before solutions have been posted is a violation of the Honor Code.

Introduction

Congratulations on your new role as software engineer for *The Prince!* Your task will be to implement the drawing portion of a game called *Tiger Words* for the online edition of the paper. It is a word-search puzzle where players look for words in a grid of letters. The interactive game play is implemented by a different programmer, so your job is simply to draw the board at different stages of play. Below are example outputs of your program, as the player finds words.



Empty Board After 1 Word After 2 Words After 7 Words Completed Puzzle

The first word found by the player is ARRAY, and the second is BREAK. Notice each word in this puzzle traces out a chain of neighboring letters, each adjacent to the previous: left, right, above, below, or diagonal. Each word has three or more letters, and every letter in the puzzle is used exactly one time (only in one word, and only once in that word).

Deliverables: Your main deliverables for this project are three Java files (`WordPath.java`, `Puzzle.java` and `TigerWords.java`) corresponding to the following three sections.

Restrictions: You may use classes defined only in `java.lang` or in our textbook libraries (such as `StdOut` and `StdIn`). You may not use classes defined in `java.util`.

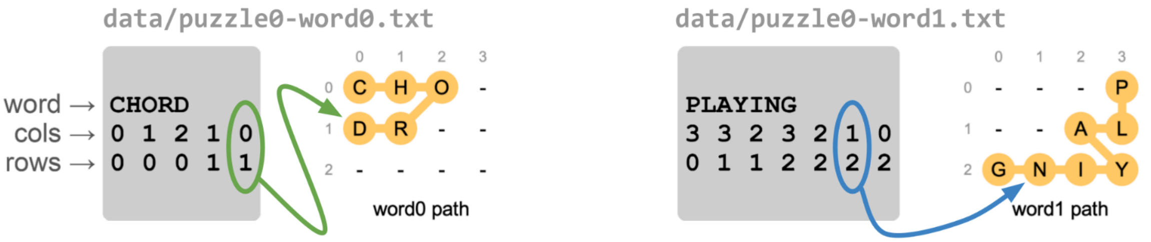
Part 1: WordPath (32/50 points)

Your first task is to implement an immutable data type `WordPath` that stores the locations (column, row) of each letter in a word — just one word! — from the puzzle solution. The API for the WordPath is provided in the `WordPath.java` template file in the project folder. Your `main` method should test every public method in the API.

The **constructor** `WordPath(String word, int[] pathCols, int[] pathRows)` takes the following arguments:

- A string representing the word (of length n).
- An array of n integers that represent the **column** location of each letter in the word.
- An array of n integers that represent the **row** location of each letter in the word.

Here is a visualization of this data (as it appears in a couple input files, in gray) and the corresponding word paths:


data/puzzle0-word0.txt

```
CHORD
0 1 2 1 0
0 0 0 1 1
```

data/puzzle0-word1.txt

```
PLAYING
3 3 2 3 2 1 0
0 1 1 2 2 2 2
```

Letter D: Column 0, Row 1

Letter N: Column 1, Row 2

In all game boards, column and row numbers are annotated in small gray numbers on the top and left of the board. In the first example, the first word is CHORD (five letters, five column values, five row values). The column and row of the final letter (D) is circled in green (column 0, row 1) and the location on the board is noted by the curvy arrow. The second word is PLAYING (with seven letters), and the location of the sixth letter (N) is highlighted in blue.

Because the data type is immutable, you should consider making a *defensive copy* of any appropriate data. This constructor must throw an `IllegalArgumentException` if the string length is less than 3, or if the string length does not match the lengths of the two arrays. (You may assume that none of these arguments is null.)

The next **constructor** `WordPath(String filename)` reads the object's data from an input file, formatted as in the figure above. Assume that the file name and contents are all correct – there is no need to check word or array lengths as in the first constructor. The file data for each of two `WordPath` objects is shown in the two gray boxes above, with corresponding word paths. We provide example data files in the directory `data`, for example: `data/puzzle0-word0.txt`

The `toString()` method returns a string that represents a WordPath object. It should include the word, followed by each letter and its location (column, row) on the path. Here are two example outputs, based on the data in the input files `data/puzzle0-word0.txt` and `data/puzzle0-word1.txt`. Your implementation should use the same formatting (colon, parentheses, commas and spaces) — just as shown below. Where you see space, it is a *single* space character.

```
CHORD: C(0,0) H(1,0) O(2,0) R(1,1) D(0,1)
PLAYING: P(3,0) L(3,1) A(2,1) Y(3,2) I(2,2) N(1,2) G(0,2)
```

Part 2: Puzzle (10/50 points)

Implement a `Puzzle` data type that reads and stores the components (dimensions and word paths) of a *TigerWords* puzzle. The constructor takes one argument — the name of an input file. The file contains:

- An integer specifying the number of **columns** in the puzzle.
- An integer specifying the number of **rows** in the puzzle.
- An integer specifying the number w of words in the solution.
- A list of w filenames, each containing a word path as described in Part 1.

You may assume that the file contents are all correct. Below are the contents of an example, `data/puzzle0.txt`:

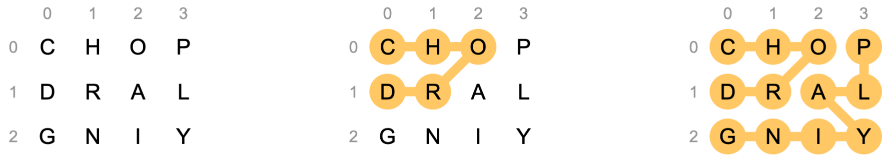
```
4 3 2
data/puzzle0-word0.txt
data/puzzle0-word1.txt
```

This specifies a puzzle that has 4 columns and 3 rows, and the solution has 2 words. The 2 word path files in this puzzle are the same examples described above in Part 1 (CHORD and PLAYING).

As in Part 1, we provide template code to get you started. But here we also provide a `main` method that calls all public methods in the class. A single command-line argument specifies the input data file. After you have implemented all public methods, your program should behave like this:

```
> java-introcs Puzzle data/puzzle0.txt
Puzzle size: 4 cols, 3 rows.
Words in puzzle (2):
CHORD: C(0,0) H(1,0) O(2,0) R(1,1) D(0,1)
PLAYING: P(3,0) L(3,1) A(2,1) Y(3,2) I(2,2) N(1,2) G(0,2)
```

Adding these words in turn to a puzzle would look like this (but you won't actually draw the board until Part 3):



Empty Board

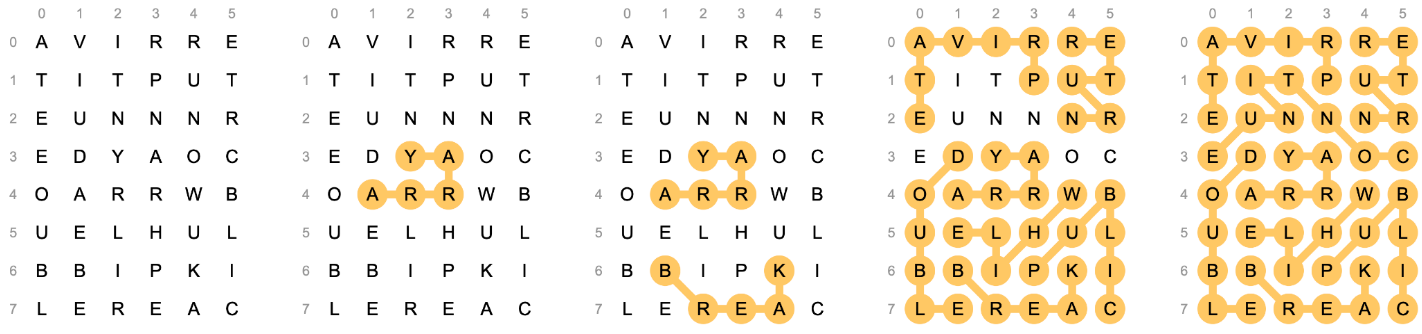
After 1 Word

Completed Puzzle

Here is another example of the behavior of your program, this time with a bigger puzzle:

```
> java-introcs Puzzle data/puzzle1.txt
Puzzle size: 6 cols, 8 rows.
Words in puzzle (8):
ARRAY: A(1,4) R(2,4) R(3,4) A(3,3) Y(2,3)
BREAK: B(1,6) R(2,7) E(3,7) A(4,7) K(4,6)
WHILE: W(4,4) H(3,5) I(2,6) L(2,5) E(1,5)
PUBLIC: P(3,6) U(4,5) B(5,4) L(5,5) I(5,6) C(5,7)
DOUBLE: D(1,3) O(0,4) U(0,5) B(0,6) L(0,7) E(1,7)
RETURN: R(4,0) E(5,0) T(5,1) U(4,1) R(5,2) N(4,2)
PRIVATE: P(3,1) R(3,0) I(2,0) V(1,0) A(0,0) T(0,1) E(0,2)
CONTINUE: C(5,3) O(4,3) N(3,2) T(2,1) I(1,1) N(2,2) U(1,2) E(0,3)
```

The word paths in this example (ARRAY, BREAK, ..., CONTINUE) look like this:



Empty Board

After 1 Word

After 2 Words

After 7 Words

Completed Puzzle

Part 3: TigerWords (8/50 points)

Before you continue: This part of the exam is worth relatively few points. Before working on this section, your solution to Parts 1 and 2 should be complete.

In this section you will actually draw the puzzle board (making pictures like the ones shown in this exam) by implementing the methods in TigerWords.java. To do so, you use a special class called StdBoard for drawing a game board that is provided for you. The only part of the StdBoard API that you need to access is:

```
// Draw a circle on the board at the given location.     [StdBoard API]
public static void drawCircle(int col, int row) {}

// Draw a letter on the board at the given location.
public static void drawLetter(int col, int row, char letter) {}

// Draw a line between a pair of letter locations.
public static void drawLine(int col0, int row0, int col1, int row1) {}
```

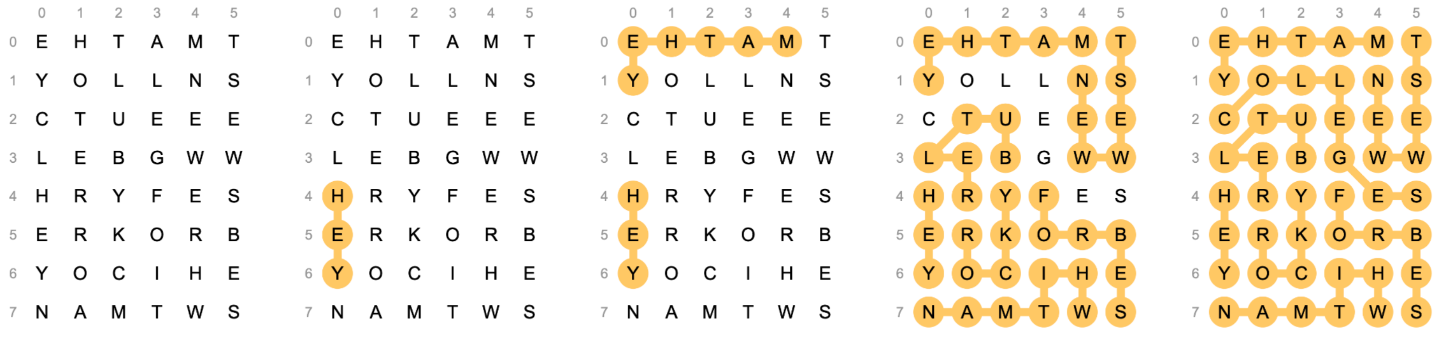
Some hints:

- Your implementation of drawCirclesOnPath should call StdBoard.drawCircle.
- Your implementation of drawLettersOnPath should call StdBoard.drawLetter.
- Your implementation of drawLinesOnPath should call StdBoard.drawLine.
- Each of these StdBoard methods should be called no more often than necessary to draw the figure.

As in Part 2, we provide template code to get you started, including a `main` method that tests the class. As in Part 2, the first command line argument is the input file. A second command line argument specifies how many word paths to highlight with circles and lines. For example, after all your drawing methods are complete, this command:

```
> java-introcs TigerWords data/puzzle2.txt 2
```

will draw the middle figure in the progression below (with caption "After 2 Words") because of the number 2 on the command line:



	0	1	2	3	4	5
0	E	H	T	A	M	T
1	Y	O	L	L	N	S
2	C	T	U	E	E	E
3	L	E	B	G	W	W
4	H	R	Y	F	E	S
5	E	R	K	O	R	B
6	Y	O	C	I	H	E
7	N	A	M	T	W	S

Empty Board

	0	1	2	3	4	5
0	E	H	T	A	M	T
1	Y	O	L	L	N	S
2	C	T	U	E	E	E
3	L	E	B	G	W	W
4	H	R	Y	F	E	S
5	E	R	K	O	R	B
6	Y	O	C	I	H	E
7	N	A	M	T	W	S

After 1 Word

	0	1	2	3	4	5
0	E	H	T	A	M	T
1	Y	O	L	L	N	S
2	C	T	U	E	E	E
3	L	E	B	G	W	W
4	H	R	Y	F	E	S
5	E	R	K	O	R	B
6	Y	O	C	I	H	E
7	N	A	M	T	W	S

After 2 Words

	0	1	2	3	4	5
0	E	H	T	A	M	T
1	Y	O	L	L	N	S
2	C	T	U	E	E	E
3	L	E	B	G	W	W
4	H	R	Y	F	E	S
5	E	R	K	O	R	B
6	Y	O	C	I	H	E
7	N	A	M	T	W	S

After 7 Words

	0	1	2	3	4	5
0	E	H	T	A	M	T
1	Y	O	L	L	N	S
2	C	T	U	E	E	E
3	L	E	B	G	W	W
4	H	R	Y	F	E	S
5	E	R	K	O	R	B
6	Y	O	C	I	H	E
7	N	A	M	T	W	S

Completed Puzzle

The last method to implement for Part 3 is `pathsOverlap`. It checks to see if two word paths contain letters in the same location, e.g., they both have a letter at location (1,2). This is useful for verifying that a puzzle board is legal before drawing it. In the provided template, the method always returns `false`, but you need to change it to return `true` if the paths do overlap. It is called automatically by the provided template code. The example below shows what happens when you run your completed program with a puzzle that has overlapping words:

```
> java-introcs TigerWords data/puzzle3.txt 8
Overlap between two word paths:
BUTLER: B(2,3) U(2,2) T(1,2) L(0,3) E(1,3) R(1,4)
COLLEGES: C(1,2) O(1,1) L(2,1) L(3,1) E(3,2) G(3,3) E(4,4) S(5,4)
```

The program prints that message and quits without drawing anything. It does so because two word paths (BUTLER and COLLEGES) overlap. Notice that the T in BUTLER and the C in COLLEGES are both at location (1,2).

Part 4: For fun (no points, just glory)

If you made it to this point, congratulations! Here we offer an optional extension to the exam worth no points, but rather some fun (and potential glory).

Create a puzzle of your own design. To do so, create a puzzle file called `glory.txt` that names the individual word path files like `glory-word0.txt`, `glory-word1.txt`, etc. Creating a 3x4 puzzle is already challenging, let alone a 6x8 puzzle like the ones shown above. Of course you can use your own program to display your puzzle. When it is complete, display the puzzle and use the "file" menu (in the `StdDraw` window with your puzzle) to save the image as `glory.png`. Submit all your "glory" files (the inputs and the saved image) in TigerFile under "Additional Files". We will share our favorites with students in the class and subsequent semesters.