CS Lab 101

**Multidimensional Loops and Arrays**

# Preamble:

Consider the following code:

**import** javax.sound.midi.MidiSystem;

**import** javax.sound.midi.Synthesizer;

**import** javax.sound.midi.MidiChannel;

**public** **class** APSynthesizerTest {

**public** **static** **void** main( String[] args ) {

**int** channel = 0; // 0 is a piano, 9 is percussion, the rest are other instruments

**int** volume = 80; // between 0 and 127

**try** {

Synthesizer synth = MidiSystem.*getSynthesizer*();

synth.open();

MidiChannel[] channels = synth.getChannels();

// --------------------------------------

// Play a few notes.

// The two arguments to the noteOn() method are:

// "MIDI note number" (pitch of the note),

// and "velocity" (i.e., volume, or intensity).

// Each of these arguments is between 0 and 127.

channels[channel].noteOn( 60, volume ); // C note

Thread.*sleep*( 200 ); // in milliseconds

channels[channel].noteOff( 60 );

channels[channel].noteOn( 62, volume ); // D note

Thread.*sleep*( 200 );

channels[channel].noteOff( 62 );

channels[channel].noteOn( 64, volume ); // E note

Thread.*sleep*( 200 );

channels[channel].noteOff( 64 );

Thread.*sleep*( 500 );

// --------------------------------------

// Play a C major chord.

channels[channel].noteOn( 60, volume ); // C

channels[channel].noteOn( 64, volume ); // E

channels[channel].noteOn( 67, volume ); // G

Thread.*sleep*( 3000 );

channels[channel].allNotesOff();

Thread.*sleep*( 500 );

synth.close();

}

**catch** (Exception e) {

e.printStackTrace();

}

}

}

# Part 1:

**Filename**: APSynthesizer.java

You will be given an int[][] that represents a song in the following fashion:

Each int[] is a musical event, which can consist of one or more notes either playing or stopping.

Each int[][] is a series of numbers representing the actions in the event. The series begins with an int representing the time that the computer must wait before beginning the next event. After this, multiple actions may take place. Each action is placed into groups of 4 as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 to play, 0 to stop | Channel | Note | Volume |

So, if the numbers were 250 1 0 60 120 1 1 64 100, it would play note 60 on channel 0 at volume 120 and note 64 on channel 1 at volume 100. It would then sleep for 250 milliseconds.

There may be any number of groups of 4 (+1) in a set. Thus, if there are 5 numbers, that would represent one note being played or stopped during an event. If there were 13 numbers, there would be 3 notes played or stopped, etc.

During a “stop” event, the Volume int is ignored.

Your goal is to, given a properly formatted array, create the method public void **play** (int[][] song). Here is some code with a song that you can copy/paste to test out your code:

**public** **class** APSynthesizerRunner {

**public** **static** **void** main(String[] args){

// Tiny Wipe Out arrangement by Ben Isecke, 11/23/14

**int**[][] wipeOut = {{400,1,9,0,100},{190,1,1,40,100},{10,0,1,40,100},{190,1,1,43,100},{10,0,1,43,100},{190,1,1,44,100},{10,0,1,44,100},{190,1,9,37,100,1,1,45,100},{10,0,1,45,100},{190,1,1,45,100},{10,0,1,45,100},{190,1,9,37,100,1,9,28,100,1,1,45,100},{10,0,1,45,100},{190,1,1,43,100},{10,0,1,43,100},{190,1,9,37,100,1,1,40,100},{10,0,1,40,100},{190,1,9,37,100,1,1,40,100},{10,0,1,40,100},{190,1,9,37,100,1,9,28,100,1,1,43,100},{10,0,1,43,100},{190,1,1,44,100},{10,0,1,44,100},{200,1,9,37,100,1,1,45,100},{190,1,9,37,100},{10,0,1,45,100},{200,1,9,37,100,1,9,28,100},{400,1,9,37,100},{200,1,9,37,100},{400,1,9,37,100,1,9,28,100}};

APSynthesizer mySynth = **new** APSynthesizer();

mySynth.play(wipeOut);

}

}

# Part 2:

**Filename**: ASCIIText.java

The goal here is to take a font from the user and convert regular text to ASCII text, similar to what we did in the Input and Output lab. However, there are a few key differences:

1. Many more characters will be accepted
2. The input from the user will be of arbitrary length
3. A font file will be specified during the run and will be read in at that point.

A sample font file has been provided with this lab. Any other fonts will conform to the same spacing specifications, and will contain the same characters in the same order.

The some number of lines of the file (each of which begin with “//”) are reserved for comments. The very next line after the comments contains a single digit number followed by a space, followed by the character list, in order. (This character list will be identical in every font.) That first digit specifies the uniform height of all of the characters in this font. There will always be a blank line between each letter in the font file. Be aware, also, that the last character in the character list is a space.

Here is a sample run (although these fonts are from the internet and do not follow the spec exactly, this example is only so you can get a sense of the program flow):

What font would you like: graffiti.txt

Text to print: Hi there!

\_\_\_ \_\_\_ .\_\_ \_\_ .\_\_ .\_.

/ | \|\_\_| \_/ |\_| |\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_| |

/ ~ \ | \ \_\_\ | \\_/ \_\_ \\_ \_\_ \\_/ \_\_ \ |

\ Y / | | | | Y \ \_\_\_/| | \/\ \_\_\_/\|

\\_\_\_|\_ /|\_\_| |\_\_| |\_\_\_| /\\_\_\_ >\_\_| \\_\_\_ >\_

\/ \/ \/ \/\/

Here is another sample run:

What font would you like: graceful.txt

Text to print: "To be or not to be"

\_ \_ \_\_\_\_ \_\_ \_\_\_\_ \_\_\_\_ \_\_ \_\_\_\_ \_\_ \_ \_\_ \_\_\_\_ \_\_\_\_ \_\_ \_\_\_\_ \_\_\_\_ \_ \_

(/(/(\_ \_)/ \ ( \_ \( \_\_) / \( \_ \ ( ( \ / \(\_ \_) (\_ \_)/ \ ( \_ \( \_\_)(/(/

)( ( O ) ) \_ ( ) \_) ( O )) / / /( O ) )( )( ( O ) ) \_ ( ) \_)

(\_\_) \\_\_/ (\_\_\_\_/(\_\_\_\_) \\_\_/(\_\_\\_) \\_)\_\_) \\_\_/ (\_\_) (\_\_) \\_\_/ (\_\_\_\_/(\_\_\_\_)

# Part 3:

**Filename**: CalendarPrinter.java

*“Thirty days have September,*

*April, June, and November;*

*February has 28 alone,*

*All the rest have 31;*

*Except leap year, that's the time,*

*When February's days are 29.”*

– Mother Goose

For this problem, you will be creating two methods: printMonth and printYear.

For printMonth(int year, int month, int janFirst, boolean leapYear), the int year represents the year in question, month will be a number from 1-12 representing the month (January-December). janFirst will be a number from 0-6 (Sunday-Saturday) representing the day of the week of January 1st, and leapYear will be true if the year is a leap year.

printMonth(2016, 2, 5, true) should print:

February 2016

Su Mo Tu We Th Fr Sa

1 2 3 4 5 6

7 8 9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29

printYear(int year, int janFirst, boolean leapYear) is similar. However, it will indicate special holidays with certain characters:

|  |  |  |
| --- | --- | --- |
| President’s Day | \*p | Third Monday in Feb |
| Veteran’s Day | \*v | November 11 |
| Thanksgiving | \*t | 4th Thursday in November |
| Memorial Day | \*m | Last Monday in May |
| Christmas | \*c | December 25 |
| New Years | \*y | January 1st |
| 4th of July | \*4 | July 4th |
| Labor Day | \*L | First Monday in September |
| Tax Day | \*x | April 15th (16th if Sunday, 17th if Saturday) |

So, printYear(2013, 2, false) should print:

**2013**

**January February March**

**Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa**

**\*y 2 3 4 5 1 2 1 2**

**6 7 8 9 10 11 12 3 4 5 6 7 8 9 3 4 5 6 7 8 9**

**13 14 15 16 17 18 19 10 11 12 13 14 15 16 10 11 12 13 14 15 16**

**20 21 22 23 24 25 26 17 \*p 19 20 21 22 23 17 18 19 20 21 22 23**

**27 28 29 30 31 24 25 26 27 28 24 25 26 27 28 29 30**

**31**

**April May June**

**Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa**

**1 2 3 4 5 6 1 2 3 4 1**

**7 8 9 10 11 12 13 5 6 7 8 9 10 11 2 3 4 5 6 7 8**

**14 \*x 16 17 18 19 20 12 13 14 15 16 17 18 9 10 11 12 13 14 15**

**21 22 23 24 25 26 27 19 20 21 22 23 24 25 16 17 18 19 20 21 22**

**28 29 30 26 \*m 28 29 30 31 23 24 25 26 27 28 29**

**30**

**July August September**

**Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa**

**1 2 3 \*4 5 6 1 2 3 1 \*L 3 4 5 6 7**

**7 8 9 10 11 12 13 4 5 6 7 8 9 10 8 9 10 11 12 13 14**

**14 15 16 17 18 19 20 11 12 13 14 15 16 17 15 16 17 18 19 20 21**

**21 22 23 24 25 26 27 18 19 20 21 22 23 24 22 23 24 25 26 27 28**

**28 29 30 31 25 26 27 28 29 30 31 29 30**

**October November December**

**Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa**

**1 2 3 4 5 1 2 1 2 3 4 5 6 7**

**6 7 8 9 10 11 12 3 4 5 6 7 8 9 8 9 10 11 12 13 14**

**13 14 15 16 17 18 19 10 \*v 12 13 14 15 16 15 16 17 18 19 20 21**

**20 21 22 23 24 25 26 17 18 19 20 21 22 23 22 23 24 \*c 26 27 28**

**27 28 29 30 31 24 25 26 27 \*t 29 30 29 30 31**

Please note that the red is just for your visual clarification, your code does NOT need to do this (though there will be extra credit if you do).

# Part 4:

**Filename**: Life.java

The Game of Life is played on an infinite two-dimensional grid of square cells. Since it’s hard to have infinite objects living in a computer, we’ll settle for an 200 (wide) x 60 (high) grid. It is not really a game, but rather a simulation that progresses with time which is considered to be discrete. At time = 0, the user inputs an initial configuration of empty and “live” cells. At each successive generation t = 1, t = 2, t = 3, etc. the configuration changes, some live cells dying, and some empty cells coming to life, the changes governed by how many occupied neighbors a given cell has. Each cell has eight neighbors, namely the cells NW, N, NE, E, SE, S, SW and W of the given cell. The state of each cell is noted and the number of neighbors is calculated at a given time t, and then the cell’s state at time t + 1 is computed according to the following rules:

1. If the cell is unoccupied and has exactly 3 occupied neighboring cells, the state changes to occupied, otherwise it remains unoccupied
2. If the cell is occupied and has fewer than 2 or more than 3 occupied neighbors, the state changes to empty, otherwise it remains occupied.

It is important to note that all changes from time = t to time = t + 1 are considered to occur simultaneously, so that, for instance, if a cell is going to die at the next time step, it still counts as an occupied cell when computing any of its neighbors’ state changes.

Write a program to run the simulation of the Game of Life! Specifically, prompt the user to enter a “lifeform” as well as an option to run the simulation

1. Forever
2. For some number of generations that the user specifies (e.g., 100)
3. Stopping after each generation and waiting for the user to press Enter

The program should print out each generation in the smallest bounding box possible (because 200 x 200 is rather hard to follow). To avoid “falling off the edge” the boundary of the box should be clamped to empty cells.

Input will be performed by keyboard. A sample run is provided on the next page. Notice that in order to center the initial configuration, you will need to determine the number of lines entered and the maximum width of a given line

**FOR EXTRA CREDIT**: make the g (go forever) option stop by itself when it reaches a stable condition.

Such as: (Note that this is actually called a “glider”, and would not end in an infinite field.)

\*

\*

\*\*\*

**EXTRA EXTRA CREDIT**: Make the g option stop at a repeating period of length 2.

Such as:

\*\*\*

**EXTRA EXTRA EXTRA CREDIT**: Make the g option stop at any repeating period.

Such as:

\*

\* \*

\* \*

\*\* \*\*

\* \*

\* \* \* \* \*

\* \* \* \*

\*\* \* \* \*\*

\* \* \* \* \*

\* \*

\*\* \*\*

\* \*

\* \*

\*

Enter starting arrangement of spaces, asterisks and returns, followed by a line with the letter "Q" to start the simulator:

\*

\*\*

\*\*

\*

Q

--------------------------------------------------------

\*

\*\*

\*\*

\*

Input g for "go forever", an integer n for n generations, or enter for the next step: