COP 3330 – Object Oriented Programming in Java
Two-Dimensional Arrays

Example: Arrays of Test Grades

In our previous lecture, we stored all of the test scores of a class in a one-dimensional array of integers. Now, let's look at a slightly different scenario.

Let's say that different players play a video game. Naturally, different players may play a different number of times. Perhaps we could store each of a single player's scores in an array. To store multiple players' scores, we would need multiple arrays, or an "array of arrays." This is precisely what a two dimensional array is. You first allocate an array of arrays. Then, you are free to allocate each individual array.

For this example, let's say that we'll ask the user to enter in the number of players. Then, for each player, they'll enter their name and the number of times they played the game. We'll store each player's name in an array of Strings. Then, we'll allocate the array to store each of their scores. (So this example will have one 1 dimensional array of Strings and one 2 dimensional array of integers.) Then, we'll read in the scores the user enters.

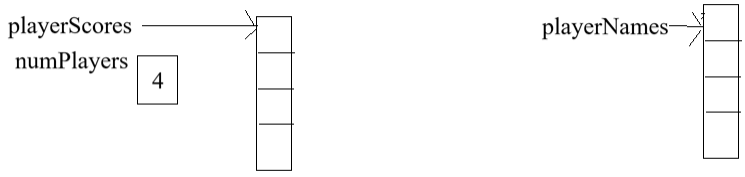
We'll also include our tools from last time, incorporating both the average and standard deviation methods, so that we can present for the user each player's average score and the standard deviation of their scores.

If we know that the number of players is numPlayers, then we can allocate our array of Strings and partially allocate our two dimensional array of scores:

```
int[][] playerScores = new int[numPlayers][];
String[] playerNames = new String[numPlayers];
```

Notice that since the size of each of the numPlayers arrays isn't the same, we can't specify anything for the second set of brackets. Quite literally, playerScores an array of numPlayers entries, each of which is currently a reference of type int[], each set to null. (null means references nothing.)

Here's a picture of what this looks like:



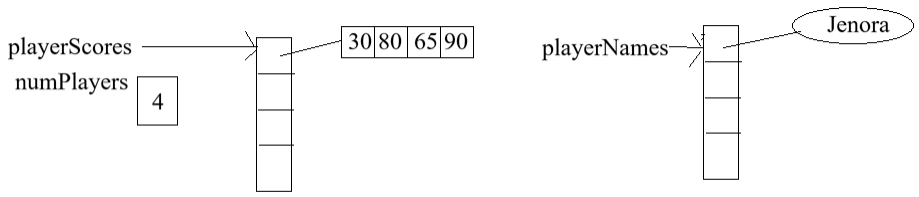
From this point, when we read in the player's names and how many times they've played, we'll allocate the appropriate space. Let's say that the first player's name is Jenora, she's played 4 times with the scores 30, 80, 65 and 90. The syntax to allocate space for playerScores[i], where i is the loop index within which this statement occurs, and numTimes represents the number of times they played the video game is:

```
playerScores[i] = new int[numTimes];
```

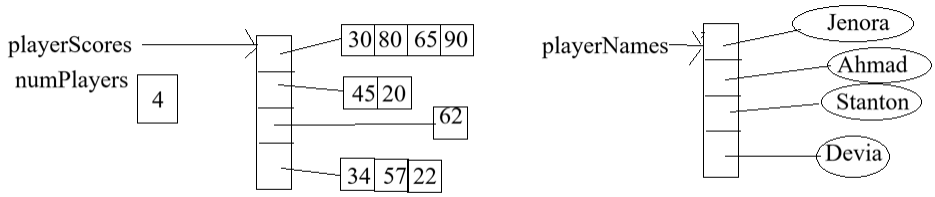
Here is all of the code that reads in this relevant data:

```
for (int i=0; i<numPlayers; i++) {

    System.out.println("Player "+(i+1)+", what is your name?");
    playerNames[i] = stdin.next();

    System.out.println(playerNames[i]+",how many times did you play the game");
    int numTimes = stdin.nextInt();

    playerScores[i] = new int[numTimes];

    System.out.println("Please enter the scores, separated by spaces.");
    for (int j=0; j<numTimes; j++)
        playerScores[i][j] = stdin.nextInt();
}
```

The picture, after finishing the loop with i = 0 and entering the previously mentioned information is:

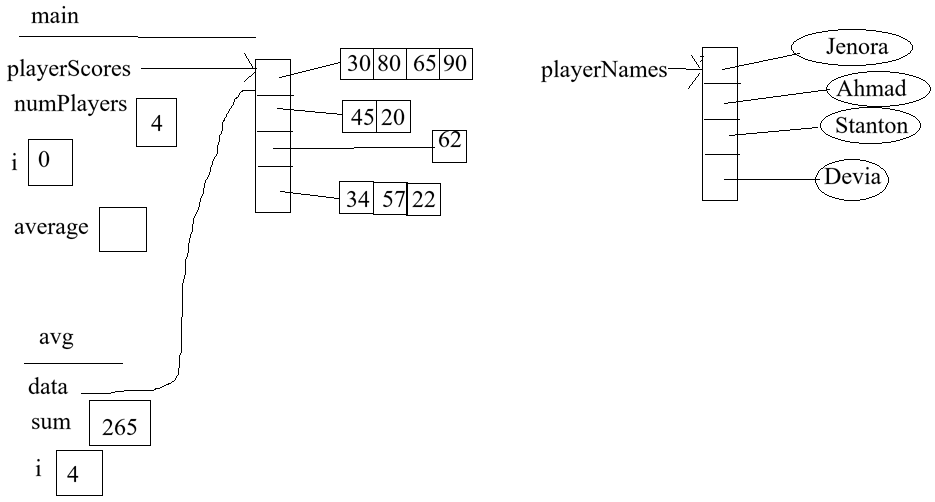


When the loop finishes, each of the String references will be pointing to objects storing the names of the players and each of the references in playerScores will be referencing integer arrays, each of potentially different sizes. To complete the picture, let's put in Ahmad (45, 20), Stanton (62), and Devita (34, 57, 22):



Now, let's take a look at what happens when we pass the array, playerScores[i], where i is the loop index, into the avg (average) method. In this picture on the next page, i = 0 in the main method and the avg function is running right before it returns the answer. The code snippet in question (with a long print in the loop omitted) looks like:

```
for (int i=0; i<numPlayers; i++) {
    double average = avg(playerScores[i]);
    double standardDev = stddev(playerScores[i]);
}
```



Thus, we see it's typical for us to pass in a one-dimensional array (that is one of several one dimensional arrays within a two dimensional array) as a parameter to a method that takes in a one **dimensional array** as the actual parameter to the method that takes in a one dimensional array reference as a formal parameter.

Second Example

Our second example will be a Tic Tac Toe game where two players can play against each other. This will be the non-object oriented version of the game. Later in the semester, after we introduce objects, we'll rewrite this code in an object-oriented design. When we look at both designs side by side, we'll be able to analyze the difference between the object-oriented design philosophy and the sort of standard design without objects.

Our board will be a two dimensional array of characters. Since each array has the same size (we will have three arrays of size three), we can allocate space for the board more succinctly as follows:

```
char[][] board = new char[3][3];
```

The first dimension as previously discussed is the number of arrays. The second dimension in this case will be the length of each of those arrays.

For example, if we want to think of a Connect Four board as seven arrays of length six, where each array is a column, then we would define it as follows:

```
char[][] con4Board = new char[7][6];
```

On the other hand, if we want it to be more natural to print and we want to think of the board as six arrays, each of length seven, where each array is a horizontal level of the board, we would define like this instead:

```
char[][] con4Board = new char[6][7];
```

In our tic-tac-toe implementation, we'll have several problems to solve:

1) Alternating player turns.
2) Reading in player moves.
3) Detecting if a player has won.

The last task is the most difficult and we'll use several methods that take the whole board (2D character array) to help us. The syntax here is what you might expect: the formal parameter is of type char[][] and when we call the method, we'll pass in the actual array in main, board.)

The full code will be posted online and in the notes we'll just highlight a few key points about it.

We use several static variables that belong to the class so that all methods have access to them:

```
final public static int SIZE = 3;
final static private char[] pieces = {'X','O'};
public static Scanner stdin;
```

Two are constants for readability (the keyword final designates a constant that can not change), and the last is because our reading from the user is done in multiple methods.

Our main method is short, it just delegates the work of doing a turn and checking the winner to our two main workhorse methods (of course, those methods do some delegation as well!). Here is the main part of main:

```
while (winner(board, movesmade) == '_') {
    executeMove(board, players, whosemove);
    whosemove = (whosemove+1)%2;
    movesmade++;
}

printboard(board);
char win = winner(board, movesmade);

if (win == 'T')
    System.out.println("It's a cats game, you both tied!");

else {
    System.out.print("Congratulations, " + whosePiece(players, win));
    System.out.println(", you have won the game.");
}
```

In the main game loop, we just loop through executing turns, and just take care of alternating whose move it is.

Then, at the end, we handle getting the winner and displaying our final message.

Let's take a look at the winner method next.

One can win in several ways:

1. Get all the squares in a row.
2. Get all the squares in a column.
3. Get all the squares in one of the two diagonals.

Our winner method will return 'X' if team 'X' wins, 'O' if team O wins, '_' if the game is still going, and 'T' if the game is tied (no more spaces to play and no winner). This method methodically goes through each option, delegating work as needed:

```
public static char winner(char[][] board, int movesmade) {

	char rowW = winRow(board);
	if (rowW != '_') return rowW;

	char colW = winCol(board);
	if (colW != '_') return colW;

	char diagW = winDiagonal(board);
	if (diagW != '_') return diagW;

	if (movesmade == 9)return 'T';

	return '_';
}
```

We can only return an answer from this method if one of my method calls returns a true winner (X or O) of the game. Otherwise, we have to continue checking the other possible ways of winning. This is why none of the if's have matching elses.

Of course, after checking all the ways of winning, we also need to look at the total moves made to see if the game is a cats game. If it isn't then the game continues.

Let's look at the logic of the following methods:

1. winRow
2. winDiagonal
3. winForwardDiag
4. winBackardDiag

The first is stand alone while #3 and #4 on the list are called by #2.

To see if someone won a row, we want to loop through each row.

Then, for each row, we want to see if the first character is taken. If it isn't ('_') then there's no winner here. If it is, then we want to see if the rest of the characters on that row match this first character. If not, there's no winner on the row.

The code is on the next page.

```
private static char winRow(char[][] board) {

    for (int i=0; i<SIZE; i++) {

        if (board[i][0] == '_') continue;

        boolean ok = true;
        for (int j=1; j<SIZE; j++)
            if (board[i][j] != board[i][0])
                ok = false;

        if (ok) return board[i][0];
    }

    return '_';
}
```

Notice the benefit of the continue statement. It allows me to avoid nesting constructs. From there, I just use a boolean flag to keep track of if a future square doesn't match the first square in the row. If there's some square that doesn't match, then our flag just get changed to false. Only if our flag is true do we return. (Again, notice how none of the if's have corresponding else's.) If we get to the very end, this is proof that no team won a row.

Now, let's look at method #2, which does some delegation:

```
private static char winDiagonal(char[][] board) {

    char forward = winForwardDiag(board);
    if (forward != '_') return forward;

    return winBackwardDiag(board);
}
```

It's pretty similar in structure to the main winner method, checking things in order. One difference is that since winBackwardDiag returns either a piece or '_', I don't need a second if, whatever this returns is our answer also.

Finally, let's look at one of the diagonal methods, for the Forward Diagonal (indexes (0,0), (1,1), etc.):

```
private static char winForwardDiag(char[][] board) {

    if (board[0][0] == '_') return '_';

    boolean ok = true;
    for (int i=1; i<SIZE; i++)
        if (board[i][i] != board[0][0])
            ok = false;

    return ok ? board[0][0] : '_';
}
```

The first line is similar to the first line inside the loop of the winRow method. Next, the loop is of similar structure to the inner loop of winRow. The only difference here is that we use the ternary operator (mostly just to illustrate its use) on the return statement as an alternative to how we previously wrote it.

The syntax of expressions with the ternary operator are as follows:

```
<boolean expr> ? expr1 : expr2
```

This is a single expression, not a statement. The value of the expression is the value of expr1, if the boolean expression preceding the question mark is true. The value of the expression is the value of expr2, if the boolean expression preceding the question mark is false. Basically, this operator makes code more concise. For some audiences, this makes the code easier to read, for others, it makes it more difficult to read.