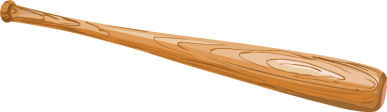
**Batting Average**

A class is a ***type*** (like an int or a double); variables of that type store references to objects, and an object's value is a bundle of variables and methods (rather than just a number like an int).

In this lab, instead of having multiple (parallel) lists of values, we'll bundle some variables and methods together into a convenient class, which we'll use to make objects, and store references to those objects in an array. As always, more information is available on our website.

**Batting Average**

In baseball, a player's ***batting average*** is the percentage of hits the player gets out of the total number of attempts the player had (referred to as "at-bats"). Write a program that will track and calculate a baseball team's batting averages.

1. Make a new class called Player that will be a bundle of related variables for describing a baseball player (name, jersey number, and some basic stats).
   1. Instance variables (every Player object will have these variables)
      * String name the player's name
      * int number the player's jersey number
      * int atBats the player's total number of at-bats
      * int hits the player's total number of hits
   2. Methods
      * public Player(String pName, int pNum) – constructor, initializes name and number to the values of the parameters; atBats and hits should start at 0 (for adding a new player to the team that doesn't have any stats yet).
      * public Player(String pName, int pNum, int atB, int pHit) – constructor used for adding a player to the team that already has stats, e.g. a player moving from JV to varsity or from another school.
      * public double getBattingAverage() – returns this player's batting average in raw form (hits divided by at-bats, where 0 <= avg <= 1). Remember that when two integers are divided, the result of that expression is an integer! Casting will help.

public String getBattingAverageString() – returns a String representation of a player's batting average. A raw average of 0.257 should return a String "257", which is how batting averages are normally displayed in baseball. Round the raw average using the Math.round() method.

Note that when you concatenate an empty String with a number (e.g. "" + 27), the result will be a String. Wise programmers will use what's already been written in this method, rather than re-calculating the raw average.

1. (Riddle) I don't eat food, but I do enjoy a light meal every day. What am I?
2. Create a class called Team, which stores all the players on the team using an array of Player objects.
   1. Instance variables
      * Player[] players – stores all the players (as Player objects) on the team
   2. Methods
      * public Team() – default constructor, initializes the instance variable (initializes players to a new array of 9 Players).
      * public Team(int numPlayers) – parameterized constructor, initializes the instance variable (initializes players to a new array of numPlayers Players).
      * public void printTeamStats() – prints the stats of all the Players on the team, like this (tab width may vary):

<name> #<number> average >>> <batting average>

Normally you would need to check if the player reference at the current index is not null, but we will assume all player references (objects) in the players array have been properly initialized.

1. When finished, create a new class called Runner with a main() method, which contains the following:
   1. Make a new Scanner object that will get input from the text file **"players.txt"**, which contains the names, numbers, and stats for the players on the team. The first line of the file is a single integer p that will indicate the number of players on this team.

/\* The players.txt file should be in the "root directory", the top-level project folder. This is the main folder with the .java files in BlueJ, and the project folder (NOT the ***src*** folder) in Eclipse. Also, don't forget the *throws IOException* after the method headers that read from a file \*/

* 1. Make a variable int p, whose value is taken from the text file. Make a new Team object called liberty, supplying the constructor with p (to instantiate the array to a certain size).
  2. For all players on the team, add new Player objects to the Team's players array. The information on each of the remaining lines in the **"players.txt"** file follows this format:
     + A String, the player's first name (use next() rather than nextLine() to get the next token, rather than the entire line)
     + An integer representing the player's jersey number
     + An integer representing the player's number of at-bats
     + An integer representing the player's number of hits
  3. Call the printTeamStats() method on the liberty object. Your output should look like the following when you run Runner's main() method. If you get a NullPointerException, make sure you've initialized all instance variables (and haven't RE-declared in the constructor\*).

Tom #8 average >>> 349

\*Example of re-declaring:

public class Team {

Player[] players; //DECLARES instance var

public Team() { //don't re-declare!

~~Player[]~~ players = new Player[9];

Mark #4 average >>> 530

Frank #23 average >>> 483

Roger #45 average >>> 203

Alan #12 average >>> 438

Harry #11 average >>> 569

Ron #36 average >>> 549

Tim #1 average >>> 490

Kyle #2 average >>> 444

1. In AP CS, all instance variables should be private (much more info on the private keyword in the powerpoints). Add the private modifier in front of all Player and Team instance variables.
   1. Now that the fields are private, you can't access them "directly" (using dot notation); you'll need to add public "getter" methods, as required, that return instance variables' values. Do not make a getter method for the players array (more information to follow).

/\* A getter method is, as the name suggests, a (public) method that 'gets' the current value of a private field. Getter methods should follow this format:

public <return type> getVariableName()

Example, for the Player class' *name* instance variable:

public String getName() { return name; } Check the website for more info! \*/

* 1. Add methods to return the Player at a particular index from the array, and return the number of players on the team.
  2. Create a public void addPlayer(Player p, int index) method in the Team class. This method will allow users to add a Player object to the players array in a Team, without having to give complete access to the array via a getter method – probably not a wise thing to do if you have been tracking stats all year.
  3. Re-run the main method of the Runner class to ensure your code still works. Fix any mistakes you may have.

**(Advanced) Fix 4 5**

Complete the method public int[] fix45(int[] nums), that will return an array that contains exactly the same numbers as the given array, but rearranged so that every 4 is immediately followed by a 5. Do not move the 4's, but every other number may move. The array contains the same number of 4's and 5's, and every 4 has a number after it that is not a 4. In this version, 5's may appear anywhere in the original array.

fix45(new int[] {5, 4, 9, 4, 9, 5}) >>> {9, 4, 5, 4, 5, 9}  
fix45(new int[] {1, 4, 1, 5}) >>> {1, 4, 5, 1}  
fix45(new int[] {1, 4, 1, 5, 5, 4, 1}) >>> {1, 4, 5, 1, 1, 4, 5}

**(Advanced) Maximum sub-array**

Complete the method public int maxSub(int[] nums), that will return the sum of the largest contiguous (back-to-back) "sub-array" within nums. A sub-array is a contiguous "chunk" of neighboring elements. Hint – as you traverse the array, determine the max of the "current" sum and the current sum plus the element at index i.

maxSub(new int[] {-2, 1, -3, 4, -1, 2, -1, -5, 4})) >>> 5 //from {4, -1, 2}

maxSub(new int[] {-2, 1, -3, 4, -1, 2, 1, -5, 4}) >>> 6 //from {4, -1, 2, 1}

**(Over 9000) Stable marriage**

Given N men and N women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners (which could lead to infidelity and is undesirable in this situation). Consider the following example:

Let there be two men m1 and m2 and two women w1 and w2.

Let m1's list of preferences be {w1, w2}

Let m2's list of preferences be {w1, w2}

Let w1's list of preferences be {m1, m2}

Let w2's list of preferences be {m1, m2}

The matching {{m1, w2}, {w1, m2}} is not stable because m1 and w1 would prefer each other over their assigned partners. The matching {m1, w1} and {m2, w2} is stable because there are no two people of opposite sex that would prefer each other over their assigned partners.

It is always possible to form stable marriages from lists of preferences (a proof is available if you're curious).

**Output:**

1 3

2 2

3 1

4 4

1 4

2 5

3 1

4 3

5 7

6 6

7 2

**Input:**

2

4

1 4 3 1 2

2 2 1 3 4

3 1 3 4 2

4 4 3 1 2

1 3 2 4 1

2 2 3 1 4

3 3 1 2 4

4 3 2 4 1

7

1 3 4 2 1 6 7 5

2 6 4 2 3 5 1 7

3 6 3 5 7 2 4 1

4 1 6 3 2 4 7 5

5 1 6 5 3 4 7 2

6 1 7 3 4 5 6 2

7 5 6 2 4 3 7 1

1 4 5 3 7 2 6 1

2 5 6 4 7 3 2 1

3 1 6 5 4 3 7 2

4 3 5 6 7 2 4 1

5 1 7 6 4 3 5 2

6 6 3 7 5 2 4 1

7 1 7 4 2 6 5 3