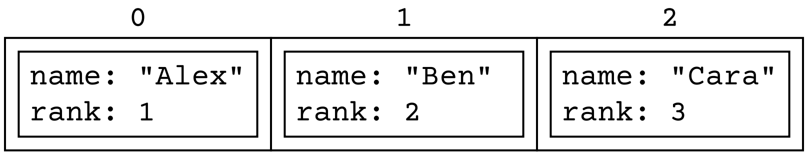
```
/** A match between two competitors */
public class Match
{
  public Match(Competitor one, Competitor two)
  { /* implementation not shown */ }

  /* There may be instance variables, constructors,
     and methods that are not shown. */
}


/** A single round of the tournament */
public class Round
{
  /** The list of competitors participating in this round */
  private ArrayList<Competitor> competitorList;

  /** Initializes competitorList, as described in part (a) */
  public Round(String[] names)
  { /* to be implemented in part (a) */ }

  /**
   * Creates an ArrayList of Match objects for the next round
   * of the tournament, as described in part (b)
   * Preconditions: competitorList contains at least one element.
   *                competitorList is ordered from best to worst rank.
   * Postcondition: competitorList is unchanged.
   */
  public ArrayList<Match> buildMatches()
  { /* to be implemented in part (b) */ }

  /* There may be instance variables, constructors,
     and methods that are not shown. */
}
```

A. Write the constructor for the `Round` class. The constructor should initialize `competitorList` to contain one `Competitor` object for each name in the `String[] names`. The order in which `Competitor` objects appear in `competitorList` should be the same as the order in which they appear in `names`, and the rank of each competitor is based on the competitor's position in `names`. Names are listed in `names` in order from the best-ranked competitor with rank `1` to the worst-ranked competitor with rank *n*, where *n* is the number of elements in `names`.

For example, assume the following code segment is executed.

```
String[] players = {"Alex", "Ben", "Cara"};
Round r = new Round(players);
```

The following shows the contents of `competitorList` in `r` after the constructor has finished executing.



Complete the `Round` constructor.

```
/** Initializes competitorList, as described in part (a) */
public Round(String[] names)
```

B. Write the `Round` method `buildMatches`. This method should return a new `ArrayList<Match>` object by pairing competitors from `competitorList` according to the following rules.

- If the number of competitors in `competitorList` is even, the best-ranked competitor is paired with the worst-ranked competitor, the second-best-ranked competitor is paired with the second-worst-ranked competitor, etc.
- If the number of competitors in `competitorList` is odd, the competitor with the best rank is ignored and the remaining competitors are paired according to the rule for an even number of competitors.

Each pair of competitors is used to create a `Match` object that should be added to the `ArrayList` to return. Competitors may appear in either order in a `Match` object, and matches may appear in any order in the returned `ArrayList`.

The following example shows the contents of `competitorList` in a `Round` object `r1` containing an odd number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r1.buildMatches()`.

`competitorList`:

0	1	2

`name: "Alex"`
`rank: 1`
	`name: "Ben"`
	`rank: 2`
		`name: "Cara"`
		`rank: 3`

The `ArrayList` to be returned contains a single match between Ben and Cara, the second and third-ranked competitors:

0

`First Competitor:`
`name: "Ben"`
`rank: 2`
`Second Competitor:`
`name: "Cara"`
`rank: 3`

The next example shows the contents of `competitorList` in a `Round` object `r2` containing an even number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r2.buildMatches()`.

`competitorList`:

0	1	2	3
name: "Rei" rank: 1	name: "Sam" rank: 2	name: "Vi" rank: 3	name: "Tim" rank: 4

The `ArrayList` to be returned contains two matches: one match between the first- and last-ranked competitors Rei and Tim, and a second match between the second- and third-ranked competitors Sam and Vi:

0	1
First Competitor: name: "Rei" rank: 1	First Competitor: name: "Sam" rank: 2
Second Competitor: name: "Tim" rank: 4	Second Competitor: name: "Vi" rank: 3

Complete the `buildMatches` method.

```
/**
 * Creates an ArrayList of Match objects for the next round
 * of the tournament, as described in part (b)
 * Preconditions: competitorList contains at least one element.
 *                competitorList is ordered from best to worst rank.
 * Postcondition: competitorList is unchanged.
 */
public ArrayList<Match> buildMatches()
```

4. This question involves reasoning about a number puzzle that is represented as a two-dimensional array of integers. Each element of the array initially contains a value between 1 and 9, inclusive. Solving the puzzle involves clearing pairs of array elements by setting them to 0. Two elements can be cleared if their values sum to 10 or if they have the same value. The puzzle is considered solved if all elements of the array are cleared.

You will write the constructor and one method of the `SumOrSameGame` class, which contains the methods that manipulate elements of the puzzle.

```java
public class SumOrSameGame
{
  private int[][] puzzle;

  /**
   * Creates a two-dimensional array and fills it with random integers,
   * as described in part (a)
   * Precondition: numRows > 0; numCols > 0
   */
  public SumOrSameGame(int numRows, int numCols)
  { /* to be implemented in part (a) */ }

  /**
   * Identifies and clears an element of puzzle that can be paired with
   * the element at the given row and column, as described in part (b)
   * Preconditions: row and col are valid row and column indices in puzzle.
   *  The element at the given row and column is between 1 and 9, inclusive.
   */
  public boolean clearPair(int row, int col)
  { /* to be implemented in part (b) */ }

  /* There may be instance variables, constructors,
     and methods that are not shown. */
}
```

Question 3: Array / ArrayList — 9 points

Canonical solution

a. **4 points**

```
public Round(String[] names)
{
   competitorList = new ArrayList<Competitor>();
   for(int i = 0; i < names.length; i++)
   {
      Competitor c = new Competitor(names[i], i + 1);
      competitorList.add(c);
   }
}
```

b. **5 points**

```
public ArrayList<Match> buildMatches()
{
   ArrayList<Match> matches = new ArrayList<Match>();

   if (competitorList.size() % 2 == 0)
   {
      for(int i = 0; i < competitorList.size()/2; i++)
      {
         matches.add(new Match(competitorList.get(i),
              competitorList.get(competitorList.size() - i - 1)));
      }
   }
   else
   {
      for(int i = 1; i < competitorList.size() / 2 + 1; i++)
      {
         matches.add(new Match(competitorList.get(i),
             competitorList.get(competitorList.size() - i)));
      }
   }
   return matches;
}
```

a. `Round`

	Scoring Criteria	Decision Rules	
1	Accesses* all elements of `names` (*no bounds errors*)	Responses **will not** earn the point if they • access `names` incorrectly	**1 point**
2	Initializes and maintains rank associated with each accessed competitor, beginning at 1	Responses **can** still earn the point even if they • maintain a rank variable with initial value 0, as long as it is offset by 1 when accessed • fail to construct a `Competitor` object Responses **will not** earn the point if they • compute rank incorrectly for any competitor	**1 point**
3	Constructs `Competitor` with provided name and computed rank	Responses **can** still earn the point even if they • access names incorrectly • compute rank incorrectly Responses **will not** earn the point if they • omit `new` in the construction of a `Competitor` or fail to use the correct number and type of parameters	**1 point**
4	Adds all constructed competitors to `competitorList` in the correct order (*algorithm*)	Responses **can** still earn the point even if they • fail to initialize `competitorList` *(`ArrayList` creation not assessed in this part)* • omit `new` in the construction of a `Competitor` • fail to access all elements of `names` • add elements with an incorrect or missing rank Responses **will not** earn the point if they • add items without constructing a `Competitor` • access `competitorList` incorrectly • fail to update the instance variable `competitorList` (e.g. by redeclaring as a local variable) • print or return a value instead of or in addition to updating `competitorList`	**1 point**

*An enhanced `for` loop inherently accesses all elements of an `ArrayList`

b. `buildMatches`

	Scoring Criteria	**Decision Rules**	
5	Declares and initializes local `ArrayList` of `Match` objects	Responses **will not** earn the point if they • fail to declare the `ArrayList`, even if they have other local variables declared	**1 point**
6	Initializes and maintains an index used to move from both ends of `competitorList` *(algorithm)*	Responses **can** still earn the point even if they • start at 0 instead of 1 or vice versa • make a bounds error with the "high" index, as long as it is counting down from the end of the list • maintain two separate index variables instead of a single offset or other equivalent strategy, as long as it is used to count up from the start and down from the end • fail to use the indices to construct a `Match`	**1 point**
7	Computes starting low index based on even/odd comparison	Responses **can** still earn the point even if they • compute the index incorrectly, as long as the even/odd cases start at different indices • call the `size` method incorrectly • modifies `competitorList` instead of skipping an index Responses **will not** earn the point if they • determine even and odd number of competitors incorrectly	**1 point**
8	Gets two elements of `competitorList` based on maintained index, constructs a `Match` object from them, and adds it to the local list	Responses **can** still earn the point even if they • omit `new` in the creation of the `Match` object *(use of `new` not assessed for this type)* • use incorrect indices Responses **will not** earn the point if they • access `competitorList` incorrectly • fail to create an object of type `Match` • use incorrect number or type of parameters on any of the method calls	**1 point**

9	Populates the list with the correct number of pairs of competitors, omitting the first competitor in the odd case, without bounds errors *(algorithm)*	Responses **can** still earn the point even if they • fail to return the `ArrayList` (*return not assessed in this question*) • determine even and odd number of competitors incorrectly, as long as the cases are unambiguous • fail to create or use a `Match` object Responses **will not** earn the point if they • fail to guard against even and odd numbers of competitors • include the first competitor for odd numbers of competitors or omit the first for even numbers • include too many matches (e.g. due to continuing past the middle of the list) • add indices instead of competitors • make syntactically incorrect call(s) to `size` or other `ArrayList` methods • permanently modify `competitorList`	**1 point**
	Question-specific penalties		
	None		

Alternate canonical solution:

```
public Round(String[] names)
{
    competitorList = new ArrayList<Competitor>();

    int rank = 1;

    for (String name : names)
    {
       Competitor c = new Competitor(name, rank);
       competitorList.add(c);
       rank++;
    }
}


public ArrayList<Match> buildMatches()
{
   ArrayList<Match> matches = new ArrayList<Match>();

   int low = 0;
   if (competitorList.size() % 2 == 1)
   {
      low = 1;
   }
   int high = competitorList.size() - 1;

   while (low < high)
   {
      Match m = new Match(competitorList.get(low),
                          competitorList.get(high));
      matches.add(m);
      low++;
      high--;
   }
   return matches;

}
```

Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be* ***unambiguously* *inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares* `"int G=99, g=0;"`, *then uses* `"while (G < 10)"` *instead of* `"while (g < 10)"`, *the context does* ***not*** *allow for the reader to assume the use of the lower case variable.*

2025 Digital Decision Rules

- Some non-ASCII characters are not printing correctly in ONE, particularly extended punctuation. Certain kinds of double-quotes may display as â⍰⍰; a character that displays as ï¼⍰ may be a parenthesis, comma, or semicolon (or other punctuation). If the badly displayed character makes sense as one of those, evaluate the response accordingly.
- If there are missing closed-double-quotes or closed-parentheses, assume they are at the end of the line where they opened, immediately before the semicolon or curly bracket (if any).
- Assume an open/left curly bracket `{` immediately after any method header or class header that does not already have one.
- Assume an appropriate amount of closing brackets before any method header to close all open brackets from the previous method or constructor.
- If there are missing curly brackets, clear indentation can "convey intent". Evaluate the response accordingly.
- Inside a method with left-justified code, indentation cannot "convey intent", so missing curly brackets cannot be assumed. Without bracketing or indentation, only the first line of a `while` / `if` / `for` is controlled by the statement; with open curly bracket and no indentation cues, the entire remainder of the method is "inside" the statement.

No Penalty

- `:` instead of `;` and vice versa
- `,` instead of `;` and vice versa