3. Consider the following incomplete class that stores information about a customer, which includes a name and unique ID (a positive integer). To facilitate sorting, customers are ordered alphabetically by name. If two or more customers have the same name, they are further ordered by ID number. A particular customer is "greater than" another customer if that particular customer appears later in the ordering than the other customer.

```
public class Customer
{
  // constructs a Customer with given name and ID number
  public Customer(String name, int idNum)
  {  /* implementation not shown */  }

  // returns the customer's name
  public String getName()
  {  /* implementation not shown */  }

  // returns the customer's id
  public int getID()
  {  /* implementation not shown */  }


  // returns 0 when this customer is equal to other;
  //   a positive integer when this customer is greater than other;
  //   a negative integer when this customer is less than other
  public int compareCustomer(Customer other)
  {  /* to be implemented in part (a) */  }

  // There may be fields, constructors, and methods that are not shown.
}
```

(a) Write the `Customer` method `compareCustomer`, which compares this customer to a given customer, `other`. Customers are ordered alphabetically by name, using the `compareTo` method of the `String` class. If the names of the two customers are the same, then the customers are ordered by ID number. Method `compareCustomer` should return a positive integer if this customer is greater than `other`, a negative integer if this customer is less than `other`, and 0 if they are the same.

For example, suppose we have the following `Customer` objects.

```
Customer c1 = new Customer("Smith", 1001);
Customer c2 = new Customer("Anderson", 1002);
Customer c3 = new Customer("Smith", 1003);
```

The following table shows the result of several calls to `compareCustomer`.

Method Call	Result
`c1.compareCustomer(c1)`	0
`c1.compareCustomer(c2)`	a positive integer
`c1.compareCustomer(c3)`	a negative integer

Complete method `compareCustomer` below.

```
// returns 0 when this customer is equal to other;
//   a positive integer when this customer is greater than other;
//   a negative integer when this customer is less than other
public int compareCustomer(Customer other)
```

GO ON TO THE NEXT PAGE.

(b) A company maintains customer lists where each list is a sorted array of customers stored in ascending order by customer. A customer may appear in more than one list, but will not appear more than once in the same list.

Write method `prefixMerge`, which takes three array parameters. The first two arrays, `list1` and `list2`, represent existing customer lists. It is possible that some customers are in both arrays. The third array, `result`, has been instantiated to a length that is no longer than either of the other two arrays and initially contains `null` values. Method `prefixMerge` uses an algorithm similar to the merge step of a Mergesort to fill the array `result`. Customers are copied into `result` from the beginning of `list1` and `list2`, merging them in ascending order until all positions of `result` have been filled. Customers who appear in both `list1` and `list2` will appear at most once in `result`.

For example, assume that three arrays have been initialized as shown below.

`list1`	Arthur 4920	Burton 3911	Burton 4944	Franz 1692	Horton 9221	Jones 5554	Miller 9360	Nguyen 4339
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`

`list2`	Aaron 1729	Baker 2921	Burton 3911	Dillard 6552	Jones 5554	Miller 9360	Noble 3335
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`

`result`	`null`	`null`	`null`	`null`	`null`	`null`
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`

In this example, the array `result` must contain the following values after the call `prefixMerge(list1, list2, result)`.

`result`	Aaron 1729	Arthur 4920	Baker 2921	Burton 3911	Burton 4944	Dillard 6552
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`

In writing `prefixMerge`, you may assume that `compareCustomer` works as specified, regardless of what you wrote in part (a). Solutions that create any additional data structures holding multiple objects (e.g., arrays, `ArrayLists`, etc.) will not receive full credit.

Complete method `prefixMerge` below.

```
// fills result with customers merged from the
// beginning of list1 and list2;
// result contains no duplicates and is sorted in
// ascending order by customer
// precondition:  result.length > 0;
//                list1.length >= result.length;
//                list1 contains no duplicates;
//                list2.length >= result.length;
//                list2 contains no duplicates;
//                list1 and list2 are sorted in
//                ascending order by customer
// postcondition: list1, list2 are not modified
public static void prefixMerge(Customer[] list1,
                               Customer[] list2,
                               Customer[] result)
```

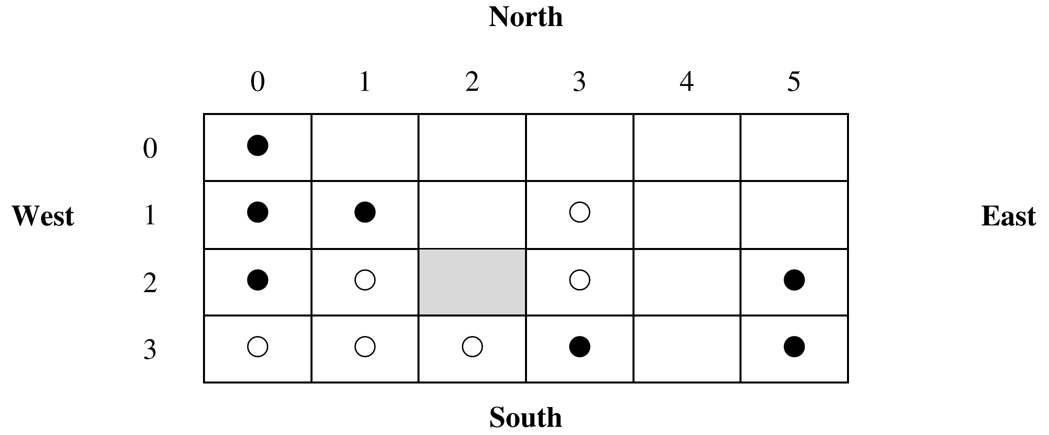
GO ON TO THE NEXT PAGE.

4. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

Consider using the `BoundedEnv` class from the Marine Biology Simulation case study to model a game board. In this implementation of the `Environment` interface, each location has at most **four** neighbors. Those neighbors are determined by the `Environment` method `neighborsOf`.

DropGame is a two-player game that is played on a rectangular board. The players — designated as BLACK and WHITE — alternate, taking turns dropping a colored piece in a column. A dropped piece will fall down the chosen column until it comes to rest in the empty location with the largest row index. If the location for the **newly dropped** piece has **three** neighbors that match its color, the player that dropped this piece wins the game.

The diagram below shows a sample game board on which several moves have been made.



	0	1	2	3	4	5
0	●					
1	●	●		○		
2	●	○	▨	○		●
3	○	○	○	●		●

The following chart shows where a piece dropped in each column would land on this board.

Column	Location for Piece Dropped in the Column
0	No piece can be placed, since the column is full
1	(0, 1)
2	(2, 2)
3	(0, 3)
4	(3, 4)
5	(1, 5)

Note that a WHITE piece dropped in column 2 would land in the shaded cell at location (2, 2) and result in a win for WHITE because the three neighboring locations — (2, 1), (3, 2), and (2, 3) — contain WHITE pieces. This move is the only available winning move on the above game board. Note that a BLACK piece dropped in column 1 would land in location (0, 1) and not result in a win because the neighboring location (0, 2) does not contain a BLACK piece.