5.1 Array-based lists

Array-based lists

An ***array-based list*** is a list ADT implemented using an array. An array-based list supports the common list ADT operations, such as append, prepend, insert after, remove, and search.

In many programming languages, arrays have a fixed size. An array-based list implementation will dynamically allocate the array as needed as the number of elements changes. Initially, the array-based list implementation allocates a fixed size array and uses a length variable to keep track of how many array elements are in use. The list starts with a default allocation size, greater than or equal to 1. A default size of 1 to 10 is common.

Given a new element, the ***append*** operation for an array-based list of length X inserts the new element at the end of the list, or at index X.

PARTICIPATION ACTIVITY 5.1.1: Appending to array-based lists.

Animation captions:

1. An array of length 4 is initialized for an empty list. Variables store the allocation size of 4 and list length of 0.
2. Appending 45 uses the first entry in the array, and the length is incremented to 1.
3. Appending 84, 12, and 78 uses the remaining space in the array.

PARTICIPATION ACTIVITY 5.1.2: Array-based lists.

1) The length of an array-based list equals the list's array allocation size.
   - ○ True
   - ○ False

2) 42 is appended to an array-based list with allocationSize = 8 and length = 4. Appending assigns the array at index _____ with 42.

3) An array-based list can have a default allocation size of 0.

- True
- False

Resize operation

An array-based list must be resized if an item is added when the allocation size equals the list length. A new array is allocated with a length greater than the existing array. Allocating the new array with twice the current length is a common approach. The existing array elements are then copied to the new array, which becomes the list's storage array.

Because all existing elements must be copied from 1 array to another, the resize operation has a runtime complexity of O(N).

PARTICIPATION ACTIVITY 5.1.3: Array-based list resize operation.

Animation content:

undefined

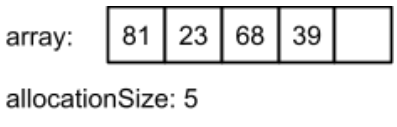
Animation captions:

1. The allocation size and length of the list are both 4. An append operation cannot add to the existing array.
2. To resize, a new array is allocated of size 8, and the existing elements are copied to the new array. The new array replaces the list's array.
3. 51 can now be appended to the array.

PARTICIPATION ACTIVITY 5.1.4: Array-based list resize operation.

Assume the following operations are executed on the list shown below:

ArrayListAppend(list, 98)
ArrayListAppend(list, 42)
ArrayListAppend(list, 63)



length : 4

1) Which operation causes ArrayListResize to be called?

- ArrayListAppend(list, 98)
- ArrayListAppend(list, 42)
- ArrayListAppend(list, 63)

2) What is the list's length after 63 is appended?

- 5
- 7
- 10

3) What is the list's allocation size after 63 is appended?

- 5
- 7
- 10

Prepend and insert after operations

The ***Prepend*** operation for an array-based list inserts a new item at the start of the list. First, if the allocation size equals the list length, the array is resized. Then all existing array elements are moved up by 1 position, and the new item is inserted at the list start, or index 0. Because all existing array elements must be moved up by 1, the prepend operation has a runtime complexity of O(N).

The ***InsertAfter*** operation for an array-based list inserts a new item after a specified index. Ex: If the contents of `numbersList` is: 5, 8, 2, `ArrayListInsertAfter(numbersList, 1, 7)` produces: 5, 8, 7, 2. First, if the allocation size equals the list length, the array is resized. Next, all elements in the array residing after the specified index are moved up by 1 position. Then, the new item is inserted at index (specified index + 1) in the list's array. The InsertAfter operation has a best case runtime complexity of O(1) and a worst case runtime complexity of O(N).

InsertAt operation.

Array-based lists often support the InsertAt operation, which inserts an item at a specified index. Inserting an item at a desired index X can be achieved by using InsertAfter to insert after index X - 1.

PARTICIPATION ACTIVITY 5.1.5: Array-based list prepend and insert after operations.

Animation content:

Step 1: Data members for a list are shown: array, allocationSize, and length.
"array:" label is followed by 8 boxes for the array's data. Entries at indices 0 to 4 are: 45, 84, 12, 78, 51. Entries at indices 5, 6, and 7 are empty.
The other two labels are "allocationSize: 8" and "length: 5".
ArrayListPrepend(list, 91) begins execution. The first if statement's condition is false. Then the for loop executes, moving items up in the array, yielding: 45, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty.

Step 2: Item 91 is assigned to index 0 in the array, yielding: 91, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty. Then length is incremented to 6.

Step 3: ArrayListInsertAfter(list, 2, 36) executes. The first if statement's condition is false, since 6 ≠ 8. The for loop moves items at indices 3, 4, and 5 up one, yielding: 91, 45, 84, 12, 12, 78, 51. Then 36 is assigned to index 3, yielding: 91, 45, 84, 36, 12, 78, 51. The array box for index 7 remains empty. Lastly, length is incremented to 7.

Animation captions:

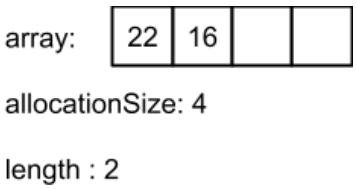
1. To prepend 91, every array element is first moved up one index.

2. Item 91 is assigned to index 0 and length is incremented to 6.
3. Inserting item 36 after index 2 requires elements at indices 3 and higher to be moved up 1. Item 36 is inserted at index 3.

PARTICIPATION ACTIVITY 5.1.6: Array-based list prepend and insert after operations.

Assume the following operations are executed on the list shown below:

ArrayListPrepend(list, 76)
ArrayListInsertAfter(list, 1, 38)
ArrayListInsertAfter(list, 3, 91)



1) Which operation causes ArrayListResize to be called?
   - ArrayListPrepend(list, 76)
   - ArrayListInsertAfter(list, 1, 38)
   - ArrayListInsertAfter(list, 3, 91)

2) What is the list's allocation size after all operations have completed?
   - 5
   - 8
   - 10

3) What are the list's contents after all operations have completed?
   - 22, 16, 76, 38, 91
   - 76, 38, 22, 91, 16
   - 76, 22, 38, 16, 91

Search and removal operations

Given a key, the ***search*** operation returns the index for the first element whose data matches that key, or -1 if not found.

Given the index of an item in an array-based list, the ***remove-at*** operation removes the item at that index. When removing an item at index X, each item after index X is moved down by 1 position.

Both the search and remove operations have a worst case runtime complexity of O(N).

PARTICIPATION ACTIVITY 5.1.7: Array-based list search and remove-at operations.

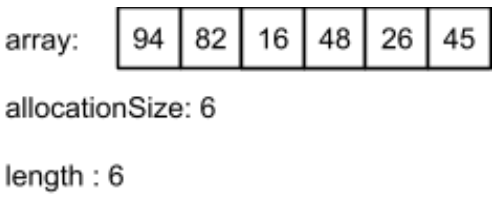
Animation content:

`undefined`

Animation captions:

1. The search for 84 compares against 3 elements before returning 2.
2. Removing the element at index 1 causes all elements after index 1 to be moved down to a lower index.
3. Decreasing the length by 1 effectively removes the last 51.
4. The search for 84 now returns 1.

PARTICIPATION ACTIVITY 5.1.8: Search and remove-at operations.



1) What is the return value from ArrayListSearch(list, 33)?

Check Show answer

2) When searching for 48, how many elements in the list will be compared with 48?

Check Show answer

3) ArrayListRemoveAt(list, 3) causes how many items to be moved down by 1 index?

Check Show answer

4) ArrayListRemoveAt(list, 5) causes how many items to be moved down by 1 index?

Check Show answer

PARTICIPATION ACTIVITY 5.1.9: Search and remove-at operations.

1) Removing at index 0 yields the best case runtime for remove-at.

- True
- False

2) Searching for a key that is not in the list yields the worst case runtime for search.

- True
- False

3) Neither search nor remove-at will resize the list's array.

- True
- False

CHALLENGE ACTIVITY 5.1.1: Array-based lists.

422352.2723990.qx3zqy7

Start

numList: | 92 | 34 | |

allocationSize: 3
length: 2

If an item is added when the allocation size equals the array length, a new array with twice current length is allocated.

Determine the length and allocation size of numList after each operation.

Operation	Length	Allocation size
ArrayListAppend(numList, 23)	Ex: 1	Ex:1
ArrayListAppend(numList, 33)		
ArrayListAppend(numList, 32)		
ArrayListAppend(numList, 95)		
ArrayListAppend(numList, 95)		

1 2 3 4

Check Next

5.2 ArrayList

ArrayList introduction

Sometimes a programmer wishes to maintain a list of items, like a grocery list, or a course roster. An ***ArrayList*** is an ordered list of reference type items that comes with Java. Each item in an ArrayList is known as an ***element***. The statement `import java.util.ArrayList;` enables use of an ArrayList.

The declaration `ArrayList<Integer> vals = new ArrayList<Integer>()` creates reference variable vals that refers to a new ArrayList object consisting of Integer objects. The ArrayList list size can grow to contain the desired elements. ArrayList does not support primitive types like int, but rather reference types like Integer. A common error among beginners is to declare an ArrayList of a primitive type like int, as in `ArrayList<int> myVals`, yielding a compilation error: "unexpected type, found : int, required: reference."

PARTICIPATION ACTIVITY 5.2.1: An ArrayList allows a programmer to maintain a list of items.

Animation content:

undefined

Animation captions:

1. valsList is a reference variable that refers to an ArrayList object consisting of Integer objects.
2. Java automatically creates an Integer object from the integer literal passed to the add() method. The add() method then adds the Integer object to the end of the ArrayList.
3. The get() method returns the element at the specified list location.
4. The set() method replaces the element at the specified list position with the new Integer object. Again, Java automatically converts the integer literal 119 to an Integer object with that value.

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Table 5.2.1: Common ArrayList methods.

add()	`add(element)`
	Create space for and add the element at the end of the list.
		`// List originally empty`
		`valsList.add(31); // List now: 31`
		`valsList.add(41); // List now: 31 41`

get()	`get(index)`
	Returns the element at the specified list location known as the ***index***. Indices start at 0.
		`// List originally: 31 41 59. Assume x is an int.`
		`x = valsList.get(0);  // Assigns 31 to x`
		`x = valsList.get(1);  // Assigns 41`
		`x = valsList.get(2);  // Assigns 59`
		`x = valsList.get(3);  // Error: No such element`

set()	`set(index, element)` Replaces the element at the specified position in this list with the specified element.	`// List originally: 31 41 59` `valsList.set(1, 119);  // List now 31 119 59`
size()	`size()` Returns the number of list elements.	`// List originally: 31 41 59. Assume x is an int.` `x = valsList.size();  // Assigns x with 3`

Accessing ArrayList elements

The ArrayList's get() method returns the element at the specified list location, and can be used to lookup the Nth item in a list. The program below allows a user to print the name of the Nth most popular operating system. The program accesses the Nth most popular operating system using `operatingSystems.get(nthOS - 1);`. Note that the index is nthOS - 1 rather than just nthOS because an ArrayList's indices start at 0, so the 1st operating system is at index 0, the 2nd at index 1, etc.

An ArrayList's index must be an integer type. The index cannot be a floating-point type, even if the value is 0.0, 1.0, etc.

Figure 5.2.1: ArrayList's ith element can be directly accessed using .get(i): Most popular OS program.

```
import java.util.ArrayList;
import java.util.Scanner;

public class MostPopularOS {
   public static void main(String [] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<String> operatingSystems = new ArrayList<String>();
      int nthOS;      // User input, Nth most popular OS

      // Source: StatCounter.com, 2018
      operatingSystems.add("Windows 10");
      operatingSystems.add("Windows 7");
      operatingSystems.add("Mac OS X");
      operatingSystems.add("Windows 8");
      operatingSystems.add("Windows XP");
      operatingSystems.add("Linux");
      operatingSystems.add("Chrome OS");
      operatingSystems.add("Other");

      System.out.println("Enter N (1-8): ");
      nthOS = scnr.nextInt();

      if ((nthOS >= 1) && (nthOS <= 8)) {
         System.out.print("The " + getNumberSuffix(nthOS) + " most
popular OS is ");
         System.out.println(operatingSystems.get(nthOS - 1));
      }
   }

   private static String getNumberSuffix(int number) {
      String[] firstThree = { "st", "nd", "rd" };
      if (number >= 1 && number <= 3) {
         return number + firstThree[number - 1];
      }
      return number + "th";
   }
}
```

```
Enter N (1-8):
1
The 1st most popular OS is Windows 10
...
Enter N (1-8):
3
The 3rd most popular OS is Mac OS X
...
Enter N (1-8):
6
The 6th most popular OS is Linux
```

ArrayList initialization

While a technique exists to initialize an ArrayList's elements with specific values in the object creation, the syntax is relatively complex. Thus, this material does not describe such initialization here.

Iterating through ArrayLists

The program below allows a user to enter 8 numbers, then prints the average of those 8 numbers. The first loop uses the add() method to add each user-specified number to the ArrayList userNums. After adding the numbers to userNums, the size() method can be used to determine the number of elements in userNums. Thus, size() is used in the second for loop to calculate the sum, and in the statement that computes the average.

With an ArrayList and loops, the program could easily be changed to support say 100 numbers; the code would be the same, and only the value of NUM_ELEMENTS would be changed to 100.

Figure 5.2.2: ArrayLists with loops.

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListAverage {
   public static void main(String [] args) {
      final int NUM_ELEMENTS = 8;
      Scanner scnr = new Scanner(System.in);
      ArrayList<Double> userNums = new
ArrayList<Double>(); // User numbers
      Double sumVal;
      Double averageVal;
      int i;

      // Get user numbers and add to userNums
      System.out.println("Enter " + NUM_ELEMENTS + "
numbers...");
      for (i = 0; i < NUM_ELEMENTS; ++i) {
         System.out.print("Number " + (i + 1) + ": ");
         userNums.add(scnr.nextDouble());
      }

      // Determine average value
      sumVal = 0.0;
      for (i = 0; i < userNums.size(); ++i) {
         sumVal = sumVal + userNums.get(i); // Calculate
sum of all numbers
      }
      averageVal = sumVal / userNums.size(); // Calculate
average

      System.out.println("Average: " + averageVal);
   }
}
```

```
Enter 8
numbers...
Number 1: 1.2
Number 2: 3.3
Number 3: 5.5
Number 4: 2.4
Number 5: 3.14
Number 6: 3.0
Number 7: 5.3
Number 8: 3.1
Average: 3.3675
```


PARTICIPATION ACTIVITY 5.2.2: ArrayList declaration, initialization, and use.

1) In a single statement, declare and initialize a reference variable for an ArrayList named frameScores that stores items of type Integer.

Check Show answer

2) Assign the Integer element at index 8 of ArrayList frameScores to a variable currFrame.

Check Show answer

3) Assign the value 10 to element at index 2 of ArrayList frameScores.

Check Show answer

4) Expand the size of ArrayList frameScores by appending an element with an integer value of 9.

Check Show answer

Java Collections Framework

*An ArrayList is one of several **Collections** supported by Java for keeping groups of items. Other collections include LinkedList, Set, Queue, Map, and many more. A programmer selects the collection whose features best suit the desired task. For example, an ArrayList can efficiently access elements at any valid index but inserts are expensive, whereas a LinkedList supports efficient inserts but access requires iterating through elements. So a program that will do many accesses and few inserts might use an ArrayList.*

CHALLENGE ACTIVITY 5.2.1: Enter the output for the ArrayList.

422352.2723990.qx3zqy7

Exploring further:

- Collections from Oracle's Java tutorial.

Start

```
import java.util.ArrayList;

public class IntegersList {
   public static void main(Strin
      ArrayList<Integer> userVal
      int i;

      userVals.add(1);
      userVals.add(6);
      userVals.add(7);

      for (i = 0; i < userVals.s
         System.out.println(user
      }
   }
}
```


1 2

Check Next

5.3 ArrayList ADT

List interface and ArrayList ADT

The ***Java Collection Framework*** (or JCF) defines interfaces and classes for common ADTs known as collections in Java. A ***Collection*** represents a generic group of objects known as elements. Java supports several different Collections, including List, Queue, Map, and others. Refer to Introduction to Collection Interfaces and Java Collections Framework overview from Oracle's Java documentation for detailed information on each Collection type. Each Collection type is an interface that declares the methods accessible to programmers.

The ***List*** interface is one of the most commonly used Collection types as it represents an ordered group of elements -- i.e., a sequence. Both an ArrayList and LinkedList are ADTs implementing the List interface. Although both ArrayList and LinkedList implement a List, a programmer should select the implementation that is appropriate for the intended task. For example, an ArrayList offers faster positional access -- e.g., `myArrayList.get(2)` -- while a LinkedList offers faster element insertion and removal.

The ArrayList type is an ADT implemented as a class (actually as a generic class that supports

different types such as ArrayList<Integer> or ArrayList<String>, although generics are discussed elsewhere).

For the commonly-used public member methods below, assume an ArrayList declared as:

```
ArrayList<T> arrayList = new ArrayList<T>();
```

where T represents the ArrayList's type, such as:

```
ArrayList<Integer> teamNums = new ArrayList<Integer>();
```

Assume ArrayList teamNums has existing Integer elements of 5, 9, 23, 11, 14.

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Table 5.3.1: ArrayList ADT methods.

get()	`T get(int index)` Returns element at specified index.	`x = teamNums.get(3); // Assig element 3's value 11 to x`

set()	`T set(int index, T newElement)`
	
	Replaces element at specified index with newElement. Returns element previously at specified index.
		`teamNums.set(0, 25); // A 25 to element 0`
		
		`x = teamNums.set(3, 88); // A 88 to element 3.`
		`// A previous element's`
		`// v 11 to x.`

size()	`int size()`
	
	Returns the number of elements in the ArrayList.
		`if (teamNums.size() > 0) { // is 5 so condition is true`
		`...`
		`}`

isEmpty()	`boolean isEmpty()`
	
	Returns true if the ArrayList does not contain any elements. Otherwise, returns false.
		`if (teamNums.isEmpty()) { // 5 so condition is false`
		`...`
		`}`

clear()	`void clear()`
	
	Removes all elements from the ArrayList.
		`teamNums.clear();`
		`// ArrayList now has no eleme`
		`System.out.println(teamNums.s`
		`// Prints 0`

add()	`boolean add(T newElement)`
	
	Adds newElement to the end of the ArrayList. ArrayList's size is increased by one.
	
	`void add(int index, T newElement)`
	
	Adds newElement to the ArrayList at the specified index. Elements at that specified index and higher are shifted over to make room. ArrayList's size is increased by one.
		`// Assume ArrayList is empty`
		`teamNums.add(77);`
		`// ArrayList is: 77`
		`teamNums.add(88);`
		`// ArrayList is: 77, 88`
		`System.out.println(teamNums.s`
		`// Prints 2`
		`teamNums.add(0, 23);`
		`// ArrayList is: 23, 77, 88`
		`teamNums.add(2, 34);`
		`// ArrayList is: 23, 77, 34,`
		`System.out.println(teamNums.s`
		`// Prints 4`

	`boolean remove(T existingElement)`	

remove()	Removes the first occurrence of an element which refers to the same object as existingElement. Elements from higher positions are shifted back to fill gap. ArrayList size is decreased by one. Return true if specified element was found and removed.
	
	`T remove(int index)`
	
	Removes element at specified index. Elements from higher positions are shifted back to fill gap. ArrayList size is decreased by one. Returns reference to element removed from ArrayList.
		`// Assume ArrayList is: 23, 7`
		`88`
		`teamNums.remove(1);`
		`// ArrayList is: 23, 34, 88`
		`System.out.println(teamNums.s`
		`// Prints 3`

Basic ArrayList methods

Use of get(), set(), size(), isEmpty(), and clear() should be straightforward.

PARTICIPATION ACTIVITY 5.3.1: ArrayList functions get(), size(), isEmpty(), and clear().

Given the following code declaring and initializing an ArrayList:

```
ArrayList<Integer> itemList = new ArrayList<Integer>();

itemList.add(0);
itemList.add(0);
itemList.add(0);
itemList.add(0);
itemList.add(99);
itemList.add(98);
itemList.add(97);
itemList.add(96);
```

1) itemList().size returns 8.

- ○ True
- ○ False

2) itemList.size(8) returns 8.

True

3) itemList.size() returns 8.

False
True

False

4) itemList.get(8) returns 96.

True

False

5) itemList.isEmpty() removes all elements.

True

False

6) After itemList.clear(), itemList.get(0) is an invalid access.

True

False

ArrayList's add() and remove() methods

Both add() methods are useful for appending new items at certain locations in an ArrayList. Similarly, the remove() method enables a programmer to remove certain elements. Resizing of the ArrayList is handled automatically by these methods.

PARTICIPATION ACTIVITY 5.3.2: ArrayList add() and remove() methods.

Animation content:

undefined

Animation captions:

1. Java automatically creates an Integer object from the integer literal passed to the add() method. The add() method then adds the element to the end of the ArrayList and increases ArrayList's size by 1.
2. The remove() method removes the element at the specified index. Elements from higher positions are moved back to fill the gap and the ArrayList size is decreased by 1.
3. The add() method can be also be used to add a new element at a specified index. Elements

at that index and higher are shifted over to make room. The ArrayList's size is increased by 1.

4. The size() method returns the number of elements currently in the ArrayList and is commonly used in a for loop to iterate through each element. The get() method returns the element at the specified index.

One can deduce that the ArrayList class has a private field that stores the current size. In fact, the ArrayList class has several private fields. However, to use an ArrayList, a programmer only needs to know the public abstraction of the ArrayList.

Example: List of players' jersey numbers

The program below assists a soccer coach in scouting players, allowing the coach to enter the jersey number of players, enter the jersey number of players the coach wants to cut, and printing a list of those numbers when requested.

Figure 5.3.1: Using ArrayList member methods: A player jersey numbers program.

```
import java.util.ArrayList;
import java.util.Scanner;

public class PlayerManager {
   // Adds playerNum to end of ArrayList
   public static void addPlayer (ArrayList<Integer> players, int
playerNum) {
      players.add(playerNum);
   }

   // Deletes playerNum from ArrayList
   public static void deletePlayer (ArrayList<Integer> players, int
playerNum) {
      int i;
      boolean found;

      // Search for playerNum in ArrayList
      found = false;
      i = 0;

      while ( (!found) && (i < players.size()) ) {
         if (players.get(i).equals(playerNum)) {
            players.remove(i); // Remove
            found = true;
         }

         ++i;
      }
   }

   // Prints player numbers currently in ArrayList
   public static void printPlayers(ArrayList<Integer> players) {
      int i;

      for (i = 0; i < players.size(); ++i) {
         System.out.println(" " + players.get(i));
      }
   }

   // Maintains ArrayList of player numbers
   public static void main(String [] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<Integer> players = new ArrayList<Integer>();
      String userInput;
      int playerNum;

      userInput = "-";

      System.out.println("Commands: 'a' add, 'd' delete,");
      System.out.println("'p' print, 'q' quit: ");

      while (!userInput.equals("q")) {
```

```
        while (!userInput.equals("q")) {
            System.out.print("Command: ");
            userInput = scnr.next();

            if (userInput.equals("a")) {
                System.out.print(" Player number: ");
                playerNum = scnr.nextInt();

                addPlayer(players, playerNum);
            }
            if (userInput.equals("d")) {
                System.out.print(" Player number: ");
                playerNum = scnr.nextInt();

                deletePlayer(players, playerNum);
            }
            else if (userInput.equals("p")) {
                printPlayers(players);
            }
        }

    }
}
```



```
Commands: 'a' add, 'd' delete,
'p' print, 'q' quit:
Command: p
Command: a
 Player number: 27
Command: a
 Player number: 44
Command: a
 Player number: 9
Command: p
 27
 44
 9
Command: d
 Player number: 9
Command: p
 27
 44
Command: q
```

The line highlighted in the addPlayer() method illustrates use of the add() member method. Note from the sample input/output that the items are stored in the ArrayList in the order they were added. The program's deletePlayer() method uses a common while loop form for finding an item in an ArrayList. The loop body checks if the current item is a match; if so, the item is deleted using the remove() method, and the variable found is set to true. The loop expression exits the loop if found is true, since no further search is necessary. A while loop is used rather than a for loop because the number of iterations is not known beforehand.

Note that the programmer did not specify an initial ArrayList size in main(), meaning the size is 0. Note from the output that the items are stored in the ArrayList in the order they were added.

PARTICIPATION ACTIVITY 5.3.3: ArrayList's add() method.

Given: `ArrayList<Integer> itemsList = new ArrayList<Integer>();`
If appropriate, type: Error
Answer the questions in order; each may modify the ArrayList.

1) What is the initial ArrayList's size?

Check Show answer

2) After itemsList.set(0, 99), what is the ArrayList's size?

Check Show answer

3) After itemsList.add(99), what is the ArrayList's size?

Check Show answer

4) After itemsList.add(77), what are the ArrayList's contents? Type element values in order separated by one space as in: 44 66

Check Show answer

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

5) After itemsList.add(44), what is the ArrayList's size?

Check Show answer

6) What does itemsList.get(itemsList.size()) return?

Check Show answer

Inserting elements in sorted order

The overloaded add() methods are especially useful for maintaining a list in sorted order.

PARTICIPATION ACTIVITY 5.3.4: Intuitive depiction of how to add items to an ArrayList while maintaining items in ascending order.

Animation content:

undefined

Animation captions:

1. The first number is added to the ArrayList.
2. 44 is greater than 27 so it is added to the end of the ArrayList.
3. 9 is less than 27. 44 and 27 are moved down so 9 can be added to front of the ArrayList.
4. The rest of the numbers are added in the appropriate spots.

zyDE 5.3.1: Insert in sorted order.

Run the program and observe the output to be: 55 4 50 19. Modify the addPlayer func
insert each number in sorted order. The new program should output: 4 19 50 55

Load default ter


```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  public class PlayerManager {
5     // Adds playerNum to end of ArrayList
6     public static void addPlayer (ArrayList<Integer> players, int playerNum) {
7        int i;
8        boolean foundHigher;
9
10       // Look for first item greater than playerNum
11       foundHigher = false;
12       i = 0;
13
14       while ( (!foundHigher) && (i < players.size()) ) {
15          if (players.get(i) > playerNum) {
16             // FIXME: insert playerNum at element i
17             foundHigher = true;
```

Run

PARTICIPATION ACTIVITY 5.3.5: The add() and remove() functions.

Given: `ArrayList<Integer> itemList = new ArrayList<Integer>();`
Assume itemList currently contains: 33 77 44.
Answer questions in order, as each may modify the vector.

1) itemList.get(1) returns 77.

- ○ True
- ○ False

2) itemList.add(1, 55) changes itemList to:
33 55 77 44.

- True
- False

3) itemList.add(0, 99) inserts 99 at the front of the list.

- True
- False

4) Assuming itemList is 99 33 55 77 44, then itemList.remove(55) results in: 99 33 77 44

- True
- False

5) To maintain a list in ascending sorted order, a given new item should be inserted at the position of the first element that is greater than the item.

- True
- False

6) To maintain a list in descending sorted order, a given new item should be inserted at the position of the first element that is equal to the item.

- True
- False

Exploring further:

- Oracle's Java String class specification
- Oracle's Java ArrayList class specification
- Oracle's Java LinkedList class specification
- Introduction to Collection Interfaces from Oracle's Java tutorials
- Introduction to List Implementations from Oracle's Java tutorials

CHALLENGE ACTIVITY 5.3.1: Enter the output of the ArrayList ADT functions.

422352.2723990.qx3zqy7

Start

Type the program's output

```
import java.util.ArrayList;
import java.util.Scanner;

public class IntegerManager {
   public static void printSize(ArrayList<Integer> numsList) {
      System.out.println(numsList.size() + " items");
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int currVal;
      ArrayList<Integer> intList = new ArrayList<Integer>();

      printSize(intList);

      currVal = scnr.nextInt();
      while (currVal >= 0) {
         intList.add(currVal);
         currVal = scnr.nextInt();
      }

      printSize(intList);

      intList.clear();

      printSize(intList);
   }
}
```


Input

1 2 3 4 5 6 -1

Output

1 2

Check Next

CHALLENGE ACTIVITY 5.3.2: Modifying ArrayList using add() and remove().

Modify the existing ArrayLists's contents, by erasing the second element, then inserting 100 and 102 in the shown locations. Use ArrayList's remove() and add() only. Sample output of below program with input 101 200 103:

```
100 101 102 103
```


422352.2723990.qx3zqy7

```
1 import java.util.ArrayList;
```

Run

View your last submission

5.4 Classes and ArrayLists

ArrayList of objects: A reviews program

A programmer commonly uses classes and ArrayLists together. The program below creates a Review class (reviews might be for a restaurant, movie, etc.), then manages an ArrayList of Review objects.

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 5.4.1: Classes and ArrayLists: A reviews program.

Review.java

```
public class Review {
   private int rating = -1;
   private String comment = "NoComment";

   public void setRatingAndComment(int revRating,
String revComment) {
      rating = revRating;
      comment = revComment;
   }
   public int getRating() { return rating; }
   public String getComment() { return comment; }
}
```


ReviewSystem.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class ReviewSystem {

   public static void main(String [] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<Review> reviewList = new
ArrayList<Review>();
      Review currReview;
      int currRating;
      String currComment;
      int i;

      System.out.println("Type rating + comments. To
end: -1");
      currRating = scnr.nextInt();
      while (currRating >= 0) {
         currReview = new Review();
         currComment = scnr.nextLine(); // Gets rest
of line
         currReview.setRatingAndComment(currRating,
currComment);
         reviewList.add(currReview);
         currRating = scnr.nextInt();
      }

      // Output all comments for given rating
      System.out.println();
      System.out.println("Type rating. To end: -1");
      currRating = scnr.nextInt();
      while (currRating != -1) {
         for (i = 0; i < reviewList.size(); ++i) {
            currReview = reviewList.get(i);
            if (currRating ==
currReview.getRating()) {
```

```
Type rating +
comments. To end:
-1
5 Great place!
5 Loved the food.
2 Pretty bad
service.
4 New owners are
nice.
2 Yuk!!!
4 What a gem.
-1

Type rating. To
end: -1
5
 Great place!
 Loved the food.
1
4
 New owners are
nice.
 What a gem.
-1
```

```
                System.out.println(currReview.getComment());
                }
            }
            currRating = scnr.nextInt();
        }
    }
}
```


PARTICIPATION ACTIVITY 5.4.1: Reviews program.

Consider the reviews program above.

1) How many member methods does the Review class have?

Check Show answer

2) After `currReview = new Review();`, what is the initial rating?

Check Show answer

3) As rating and comment pairs are read from input, what method adds the pairs to ArrayList reviewList? Type the name only, like: append.

Check Show answer

4) How many comments are output for reviews having a rating of 5?

Check Show answer

A class with an ArrayList: The Reviews class

A class can also involve ArrayLists. The program below redoes the example above, creating a Reviews class for managing an ArrayList of Review objects.

The Reviews class has methods for reading reviews and printing comments. The resulting main() is clearer than above.

The Reviews class has a getter method that returns the average rating. The method computes the average rather than reading a private field, but the class user need not know how the method is implemented.

Figure 5.4.2: Improved reviews program with a Reviews class.

Review.java

```
public class Review {
   private int rating = -1;
   private String comment = "NoComment";

   public void setRatingAndComment(int revRating,
String revComment) {
      rating = revRating;
      comment = revComment;
   }
   public int getRating() { return rating; }
   public String getComment() { return comment; }
}
```


Reviews.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class Reviews {
   private ArrayList<Review> reviewList = new
ArrayList<Review>();

   public void inputReviews(Scanner scnr) {
      Review currReview;
      int currRating;
      String currComment;

      currRating = scnr.nextInt();
      while (currRating >= 0) {
         currReview = new Review();
         currComment = scnr.nextLine(); // Gets rest
of line
         currReview.setRatingAndComment(currRating,
currComment);
         reviewList.add(currReview);
         currRating = scnr.nextInt();
      }
   }

   public void printCommentsForRating(int currRating)
{
      Review currReview;
      int i;

      for (i = 0; i < reviewList.size(); ++i) {
         currReview = reviewList.get(i);
         if (currRating == currReview.getRating()) {

System.out.println(currReview.getComment());
         }
      }
```

```
    }

    public int getAverageRating() {
        int ratingsSum;
        int i;

        ratingsSum = 0;
        for (i = 0; i < reviewList.size(); ++i) {
            ratingsSum += reviewList.get(i).getRating();
        }
        return (ratingsSum / reviewList.size());
    }
}
```


ReviewSystem.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class ReviewSystem {

    public static void main(String [] args) {
        Scanner scnr = new Scanner(System.in);
        Reviews allReviews = new Reviews();
        String currName;
        int currRating;

        System.out.println("Type rating + comments. To
end: -1");
        allReviews.inputReviews(scnr);

        System.out.println("\nAverage rating: ");

System.out.println(allReviews.getAverageRating());

        // Output all comments for given rating
        System.out.println("\nType rating. To end: -1");
        currRating = scnr.nextInt();
        while (currRating != -1) {

allReviews.printCommentsForRating(currRating);
            currRating = scnr.nextInt();
        }
    }
}
```

```
Type rating +
comments. To
end: -1
5 Great place!
5 Loved the
food.
2 Pretty bad
service.
4 New owners
are nice.
2 Yuk!!!
4 What a gem.
-1

Average rating:
3

Type rating. To
end: -1
5
 Great place!
 Loved the
food.
1
4
 New owners are
nice.
 What a gem.
-1
```


PARTICIPATION ACTIVITY 5.4.2: Reviews program.

Consider the reviews program above.

1) The first class is named Review. What

is the second class named?

- Reviews
- reviewList
- allReviews

2) How many private fields does the Reviews class have?

- 0
- 1
- 2

3) Which method reads all reviews from input?

- getReviews()
- inputReviews()

4) What does printCommentsForRating() do?

- Prints reviews sorted by rating level.
- Print all reviews above a rating level.
- Print all reviews having a particular rating level.

5) Does main() declare an ArrayList?

- Yes
- No

Using Reviews in the Restaurant class

Programmers commonly use classes within classes. The program below uses a Restaurant class that contains a Reviews class so reviews can be associated with a specific restaurant.

Figure 5.4.3: Improved reviews program with a Restaurant class.

Restaurant.java

```
import java.util.Scanner;

// Review and Reviews classes omitted from the
figure

public class Restaurant {
   private String name;
   private Reviews reviews = new Reviews();

   public void setName(String restaurantName) {
      name = restaurantName;
   }

   public void readAllReviews(Scanner scnr) {
      System.out.println("Type ratings +
comments. To end: -1");
      reviews.inputReviews(scnr);
   }

   public void printCommentsByRating() {
      int i;

      System.out.println("Comments for each
rating level: ");
      for (i = 1; i <= 5; ++i) {
         System.out.println(i + ":");
         reviews.printCommentsForRating(i);
      }
   }
}
```

RestaurantReviews.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class RestaurantReviews {

   public static void main (String [] args) {
      Scanner scnr = new Scanner(System.in);
      Restaurant ourPlace = new Restaurant();
      String currName;

      System.out.println("Type restaurant name:
");
      currName = scnr.nextLine();
      ourPlace.setName(currName);
      System.out.println();

      ourPlace.readAllReviews(scnr);
      System.out.println();
```

```
Type restaurant name:
Maria's Healthy Food

Type ratings +
comments. To end: -1
5 Great place!
5 Loved the food.
2 Pretty bad service.
4 New owners are nice.
2 Yuk!!!
4 What a gem.
-1

Comments for each
rating level:
1:
2:
 Pretty bad service.
 Yuk!!!
3:
4:
```

```
        ourPlace.printCommentsByRating();
    }
}
```

```
 New owners are nice.
 What a gem.
5:
 Great place!
 Loved the food.
```

PARTICIPATION ACTIVITY 5.4.3: Restaurant program with reviews.

Consider the restaurant program above.

1) How many private fields does the Restaurant class have?
   - 0
   - 1
   - 2

2) Which Restaurant method reads all reviews?
   - getReviews()
   - inputReviews()
   - readAllReviews()

3) What does printCommentsByRating() do?
   - Prints comments sorted by rating level.
   - Print all reviews having a particular rating level.

4) Does main() declare a Reviews object?
   - Yes
   - No

CHALLENGE ACTIVITY 5.4.1: Enter the output of classes and ArrayLists.

422352.2723990.qx3zqy7

Start

Type the program's output

CallProduct.java | Product.java


```
import java.util.ArrayList;
import java.util.Scanner;

public class CallProduct {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<Product> productList = new ArrayList<Product>();
      int currPrice;
      String currName;
      int i;
      Product resultProduct;

      currPrice = scnr.nextInt();
      while (currPrice >= 0) {
         resultProduct = new Product();
         currName = scnr.next();
         resultProduct.setPriceAndName(currPrice, currName);
         productList.add(resultProduct);
         currPrice = scnr.nextInt();
      }

      resultProduct = productList.get(0);

      for (i = 0; i < productList.size(); ++i) {
         if (productList.get(i).getPrice() > resultProduct.getPrice()) {
            resultProduct = productList.get(i);
         }
      }

      System.out.println("$" + resultProduct.getPrice() + " " + resultProduct.getName());
   }
}
```

1 | 2

Check Next

5.5 Comparable Interface: Sorting an ArrayList

Sorting the elements of an ArrayList into ascending or descending order is a common programming task. Java's ***Collections*** class provides static methods that operate on various types of lists such as an ArrayList. The sort() method sorts collections into ascending order provided that the elements within the collection implement the Comparable interface (i.e., the elements are also

of the type Comparable). For example, each of the primitive wrapper classes (e.g., Integer, Double, etc.) implements the ***Comparable*** interface, which declares the compareTo() method. Classes implementing the Comparable interface must define a custom implementation of the compareTo() method. A programmer may use sort() to sort an ArrayList in which the elements implement the Comparable interface (e.g., Integer). The programmer must import java.util.Collections to use the sort() method. The following example demonstrates the use of sort() to sort an ArrayList of Integer objects.

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 5.5.1: Collections' sort() method operates on lists of Integer objects.

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Collections;

public class ArraySorter {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      final int NUM_ELEMENTS = 5;                                  // Number
of items in array
      ArrayList<Integer> userInts = new ArrayList<Integer>(); // Array of
user defined values
      int i;                                                       // Loop
index

      // Prompt user for input, add values to array
      System.out.println("Enter " + NUM_ELEMENTS + " numbers...");
      for (i = 1; i <= NUM_ELEMENTS; ++i) {
         System.out.print(i + ": ");
         userInts.add(scnr.nextInt());
      }

      // Sort ArrayList of Comparable elements
      Collections.sort(userInts);

      // Print sorted array
      System.out.print("\nSorted numbers: ");
      for (i = 0; i < NUM_ELEMENTS; ++i) {
         System.out.print(userInts.get(i) + " ");
      }
      System.out.println("");
   }
}
```

```
Enter 5 numbers...
1: -10
2: 99
3: 31
4: 5
5: 31

Sorted numbers: -10 5 31 31 99
```


The Collections' sort() method calls the compareTo() method on each object within the ArrayList to determine the order and produce a sorted list.

The sort() method can also be used to sort an ArrayList containing elements of a user-defined class type. The only requirement, however, is that the user-defined class must also implement the Comparable interface and override the compareTo() method, which should return a number that

determines the ordering of the two objects being compared as shown below.

compareTo(otherComparable) compares a Comparable object to otherComparable, returning a number indicating if the Comparable object is less than, equal to, or greater than otherComparable. The method compareTo() will return 0 if the two Comparable objects are equal. Otherwise, compareTo() returns a negative number if the Comparable object is less than otherComparable, or a positive number if the Comparable object is greater than otherComparable.

The following program allows a user to add new employees to an ArrayList and print employee information in sorted order. The EmployeeData class implements `Comparable<EmployeeData>` and overrides the compareTo() method in order to enable the use of the Collections class's sort() method.

Figure 5.5.2: Sorting an ArrayList of employee records.

EmployeeData.java:

```
public class EmployeeData implements Comparable<EmployeeData> {
   private String firstName; // First Name
   private String lastName;  // Last Name
   private Integer emplID;   // Employee ID
   private Integer deptNum;  // Department Number

   EmployeeData(String firstName, String lastName, Integer emplID,
Integer deptNum) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.emplID = emplID;
      this.deptNum = deptNum;
   }

   @Override
   public int compareTo(EmployeeData otherEmpl) {
      String fullName;           // Full name, this employee
      String otherFullName;      // Full name, comparison employee
      int comparisonVal;         // Outcome of comparison

      // Compare based on department number first
      comparisonVal = deptNum.compareTo(otherEmpl.deptNum);

      // If in same organization, use name
      if (comparisonVal == 0) {
         fullName = lastName + firstName;
         otherFullName = otherEmpl.lastName + otherEmpl.firstName;
         comparisonVal = fullName.compareTo(otherFullName);
      }

      return comparisonVal;
   }

   @Override
   public String toString() {
      return lastName + " " + firstName +
             "       \tID: " + emplID +
             "\t\tDept. #: " + deptNum;
   }
}
```


EmployeeRecords.java:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Collections;

public class EmployeeRecords {

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

```
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<EmployeeData> emplList = new ArrayList<EmployeeData>();
// Stores all employee data
      EmployeeData emplData;
// Stores info for one employee
      String userCommand;
// User defined add/print/quit command
      String emplFirstName;
// User defined employee first name
      String emplLastName;
// User defined employee last name
      Integer emplID;
// User defined employee ID
      Integer deptNum;
// User defined employee Dept
      int i;
// Loop counter

      do {
         // Prompt user for input
         System.out.println("Enter command ('a' to add new employee,
'p' to print all employees, 'q' to quit): ");
         userCommand = scnr.next();

         // Add new employee entry
         if (userCommand.equals("a")) {
            System.out.print("First Name: ");
            emplFirstName = scnr.next();
            System.out.print("Last Name: ");
            emplLastName = scnr.next();
            System.out.print("ID: ");
            emplID = scnr.nextInt();
            System.out.print("Department Number: ");
            deptNum = scnr.nextInt();
            emplData = new EmployeeData(emplFirstName, emplLastName,
emplID, deptNum);
            emplList.add(emplData);
         }
         // Print all entries
         else if (userCommand.equals("p")) {

            // Sort employees by department number first
            // and name second
            Collections.sort(emplList);

            System.out.println("");
            System.out.println("Employees: ");
            // Access employee records
            for (i = 0; i < emplList.size(); ++i) {
               System.out.println(emplList.get(i).toString());
            }
            System.out.println("");
         }
      } while (!userCommand.equals("q"));
   }
}
```

```
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: Michael
Last Name: Faraday
ID: 124
Department Number: 1
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: Ada
Last Name: Lovelace
ID: 203
Department Number: 2
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: James
Last Name: Maxwell
ID: 123
Department Number: 1
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: Alan
Last Name: Turing
ID: 201
Department Number: 2
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
p

Employees:
Faraday Michael          ID: 124           Dept. #: 1
Maxwell James            ID: 123           Dept. #: 1
Lovelace Ada             ID: 203           Dept. #: 2
Turing Alan              ID: 201           Dept. #: 2

Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
q
```


Interface implementation is a concept similar to class inheritance. The ***implements*** keyword tells the compiler that a class implements, instead of extends, a particular interface (e.g., Comparable<EmployeeData>). Like with inheritance, an Employee object is of type Comparable<EmployeeData> as well as EmployeeData. However, an interface differs from a typical super class in that interfaces cannot be instantiated and the methods declared by an interface must be overridden and defined by the implementing class. In this example, the built-in Comparable interface declares the compareTo() method, which EmployeeData must override. Failing to override compareTo() results in the following compiler error: "EmployeeData is not abstract and does not override abstract method compareTo(EmployeeData) in java.lang.Comparable".

The ArrayList of EmployeeData elements is sorted via the sort() method, as in `Collections.sort(emplList);`. The sort() method invokes each element's compareTo() method in order to determine the ordering and sort the ArrayList. EmployeeData's compareTo() method performs a comparison between two EmployeeData objects, prioritizing department number over an employee's name. Thus, an employee hired within a numerically smaller department number will precede another employee with a numerically larger department number,

and vice versa. If two employees are located in the same department, they are compared lexicographically based on their names. The end result is that employees are sorted according to department number, and employees in the same department are sorted in alphabetical order according to their names.

zyDE 5.5.1: Sort Employee elements using employee IDs.

Modify EmployeeData's compareTo() method so that elements are sorted based on t employees' department number (deptNum) and ID (emplID). Specifically, employee's first be sorted in ascending order according to department number first, and those er within the same department should be sorted in ascending order according to the en ID.

Current file: **EmployeeData.java** Load default ter

```
1
2  public class EmployeeData implements Comparable<EmployeeData> {
3     private String firstName; // First Name
4     private String lastName;  // Last Name
5     private Integer emplID;   // Employee ID
6     private Integer deptNum;  // Department Number
7
8     EmployeeData(String firstName, String lastName, Integer emplID, Integer dep
9        this.firstName = firstName;
10       this.lastName = lastName;
11       this.emplID = emplID;
12       this.deptNum = deptNum;
13    }
14
15    @Override
16    public int compareTo(EmployeeData otherEmpl) {
17       String fullName;           // Full name, this employee
```

```
a Michael Faraday 124 1
a Ada Lovelace 203 2
a James Maxwell 123 1
```

Run

Classes that already inherit from a base class can also be defined to implement an interface. For example, the above EmployeeData class could have been defined so that it extends a Person class and implements the Comparable interface, as in

```
public class EmployeeData extends Person implements Comparable<EmployeeD
```

Finally, note that Comparable's compareTo() method is meant to work with any class. Thus, a programmer must append the class name in angle brackets to "Comparable", as in `Comparable<EmployeeData>`, in order to tell the compiler that the compareTo() method requires an argument of the indicated class type. Generic methods, classes, and interfaces are discussed in more detail elsewhere.

PARTICIPATION ACTIVITY 5.5.1: Sorting elements in an ArrayList.

1) The following statement sorts an ArrayList called prevEmployees. Assume prevEmployees is an appropriately initialized ArrayList of EmployeeData elements.
```
sort(prevEmployees);
```
   - ○ True
   - ○ False

2) An interface contains method declarations, as opposed to method definitions.
   - ○ True
   - ○ False

3) An interface cannot be instantiated.
   - ○ True
   - ○ False

4) The EmployeeData class, as defined above, is not required to override the compareTo() method declared by the Comparable interface.
   - ○ True
   - ○ False

5) A class may not simultaneously "extend" a class and "implement" an interface.
   - ○ True
   - ○ False

CHALLENGE ACTIVITY 5.5.1: Enter the output for sorting an ArrayList.

422352.2723990.qx3zqy7

Start

Type the program's output


```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Collections;

public class ArraySorter {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      final int NUM_ELEMENTS = 5;
      ArrayList<Double> userElements = new ArrayList<Double>();
      int i;

      for (i = 0; i < NUM_ELEMENTS; ++i) {
         userElements.add(scnr.nextDouble());
      }

      Collections.sort(userElements);

      for (i = 0; i < NUM_ELEMENTS; ++i) {
         System.out.println(userElements.get(i));
      }
   }
}
```

Input

```
2.59
3.82
6.73
1.2
5.27
```

Output

1 2

Check Next

Exploring further:

- Introduction to interfaces from Oracle's Java tutorials
- Introduction to object ordering from Oracle's Java tutorials
- Oracle's Java Comparable class specification

5.6 Generic methods

Multiple methods may be nearly identical, differing only in their data types, as below.

Figure 5.6.1: Methods may have identical behavior, differing only in data types.

```
// Find the minimum of three **ints**
public static Integer tripleMinInt(Integer item1, Integer item2, Integer
item3) {
   Integer minVal;

   minVal = item1;

   if (item2.compareTo(minVal) < 0) {
      minVal = item2;
   }
   if (item3.compareTo(minVal) < 0) {
      minVal = item3;
   }
   return minVal;
}


// Find the minimum of three **chars**
public static Character tripleMinChar(Character item1, Character item2,
Character item3) {
   Character minVal;

   minVal = item1;

   if (item2.compareTo(minVal) < 0) {
      minVal = item2;
   }
   if (item3.compareTo(minVal) < 0) {
      minVal = item3;
   }
   return minVal;
}
```


Writing and maintaining redundant methods that only differ by data type can be time-consuming and error-prone. The language supports a better approach.

A ***generic method*** is a method definition having a special type parameter that may be used in place of types in the method.

Figure 5.6.2: A generic method enables a method to handle various class types.

```
public class ItemMinimum {
   public static <TheType extends Comparable<TheType>>
   TheType tripleMin(TheType item1, TheType item2, TheType item3) {
      TheType minVal = item1; // Holds min item value, init to first item

      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }

   public static void main(String[] args) {
      Integer num1 = 55;    // Test case 1, item1
      Integer num2 = 99;    // Test case 1, item2
      Integer num3 = 66;    // Test case 1, item3

      Character let1 = 'a'; // Test case 2, item1
      Character let2 = 'z'; // Test case 2, item2
      Character let3 = 'm'; // Test case 2, item3

      String str1 = "zzz";  // Test case 3, item1
      String str2 = "aaa";  // Test case 3, item2
      String str3 = "mmm";  // Test case 3, item3

      // Try tripleMin method with Integers
      System.out.println("Items: " + num1 + " " + num2 + " " + num3);
      System.out.println("Min: " + tripleMin(num1, num2, num3) + "\n");

      // Try tripleMin method with Characters
      System.out.println("Items: " + let1 + " " + let2 + " " + let3);
      System.out.println("Min: " + tripleMin(let1, let2, let3) + "\n");

      // Try tripleMin method with Strings
      System.out.println("Items: " + str1 + " " + str2 + " " + str3);
      System.out.println("Min: " + tripleMin(str1, str2, str3) + "\n");
   }
}
```



```
run:
Items: 55 99 66
Min: 55

Items: a z m
Min: a

Items: zzz aaa mmm
Min: aaa
```

The method return type is preceded by `<TheType extends Comparable<TheType>>`, where TheType can be any identifier. That type is known as a ***type parameter*** and can be used throughout the method for any parameter types, return types, or local variable types. The identifier is known as a template parameter, and may be various reference types or even another template parameter.

A type parameter may be associated with a ***type bound*** to specify the class types for which a type parameter is valid. Type bounds are specified using the extends keyword and appear after the corresponding type parameter. For example, the code `<TheType extends Comparable<TheType>>` specifies that TheType is bounded by the type bound `Comparable<TheType>`. Thus, TheType may only represent types that implement the Comparable interface. If the type bound is a class type (e.g., the Number class), the type parameter may only represent types that are of the type specified by the type bound or any derived classes.

Type bounds are also necessary to enable access to the class members of the class specified by the type bound (e.g., compareTo()) via a variable of a generic type (e.g., item1, item2, item3, and min). By bounding TheType to the Comparable interface, the programmer is able to invoke the Comparable interface's compareTo() method with the generic types, as in `item2.compareTo(min);`. Attempting to invoke a class member via a generic type without specifying the appropriate type bound results in a compiler error.

Importantly, type arguments cannot be primitive types such as int, char, and double. Instead, the type arguments must be reference types. If primitive types are desired, a programmer should use the corresponding primitive wrapper classes (e.g., Integer, Character, Double, etc.), discussed elsewhere.

PARTICIPATION ACTIVITY 5.6.1: Generic methods.

1) Fill in the blank.

```
public static <MyType extends
Comparable<MyType>>
______ GetMax3 (MyType i,
MyType j, MyType k) {
   ...
};
```

- TheType
- Integer
- MyType

2) Fill in the blank.

```
public static <______ extends
Comparable<______>>
T TripleMedian(T item1, T
item2, T item3) {
   ...
}
```

- Integer
- TheType
- T
- Not possible; T is not a valid type.

3) For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, k) but those arguments are of type Character?

- The compiler generates an error message because only Integer and Double are supported.
- During runtime, the Character values are forced to be Integer values.
- The compiler creates a method with Character types and calls that method.

4) For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, k) but those arguments are String objects?

- The method will compare the Strings.
- The compiler generates an error, because only numerical types can be passed.

5) For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, z), where i and j are Integers, but z is a String?

Programmers optionally may explicitly specify the generic type as a special argument, as in `ItemMinimum.<Integer>tripleMin(num1, num2, num3);`.

A generic method may have multiple parameters:

Construct 5.6.1: Method definition with multiple generics.


```
modifiers <Type1 extends BoundType1, Type2 extends
BoundType2>
ReturnType methodName(parameters) {
   ...
}
```

Note that the modifiers represent a space delimited list of valid modifiers like `public` and `static`.

zyDE 5.6.1: Generic methods.

This program currently fails to compile because the parameters cannot be automatic converted to Double in the statement `tripleSum = item1 + item2 + item3;` Because TheType is bound to the class Number, the Number class' doubleValue() me be called to get the value of the parameters as a double value. Modify tripleAvg() met use the doubleValue() method to convert each of the parameters to a double value b adding them.

Load default template...

Run

```
1
2  public class ItemMinimum {
3
4     public static <TheType extends Number>
5     Double tripleAvg(TheType item1, TheType
6        Double tripleSum;
7
8        tripleSum = item1 + item2 + item3;
9
10       return tripleSum / 3.0;
11    }
12
13    public static void main(String[] args) {
14       Integer intVal1 = 55;
15       Integer intVal2 = 99;
16       Integer intVal3 = 66;
17
```

CHALLENGE ACTIVITY 5.6.1: Generic methods.

422352.2723990.qx3zqy7

Start

Type the program's output

```
public class ItemChoice {

   public static <T extends Comparable<T>>
   T chooseItem(T item1, T item2, T item3) {
      T chosenItem = item1;

      if (item2.compareTo(chosenItem) < 0) {
         chosenItem = item2;
      }
      if (item3.compareTo(chosenItem) < 0) {
         chosenItem = item3;
      }
      return chosenItem;
   }

   public static void main(String[] args) {
      Integer i1 = 6;
      Integer i2 = 5;
```


Exploring further:

- Introduction to generics from Oracle's Java tutorials
- Introduction to bounded type parameters from Oracle's Java tutorials

5.7 Class generics

Multiple classes may be nearly identical, differing only in their data types. The following shows a class managing three Integer numbers, and a nearly identical class managing three Short numbers.

Figure 5.7.1: Classes may be nearly identical, differing only in data type.

```
public class TripleInt {
   private Integer item1; // Data value 1
   private Integer item2; // Data value 2
   private Integer item3; // Data value 3

   public TripleInt(Integer i1, Integer i2, Integer i3) {
      item1 = i1;
      item2 = i2;
      item3 = i3;
   }

   // Print all data member values
   public void printAll() {
      System.out.println("(" + item1 + "," + item2 + "," + item3 + ")");
   }

   // Return min data member value
   public Integer minItem() {
      Integer minVal;         // Holds min item value, init to first item

      minVal = item1;

      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }
}

public class TripleShort {
   private Short item1; // Data value 1
   private Short item2; // Data value 2
   private Short item3; // Data value 3

   public TripleShort(Short i1, Short i2, Short i3) {
      item1 = i1;
      item2 = i2;
      item3 = i3;
   }

   // Print all data member values
   public void printAll() {
      System.out.println("(" + item1 + "," + item2 + "," + item3 + ")");
   }

   // Return min data member value
   public Short minItem() {
      Short minVal;         // Holds min item value, init to first item

      minVal = item1;
```

```
        if (item2.compareTo(minVal) < 0) {
           minVal = item2;
        }
        if (item3.compareTo(minVal) < 0) {
           minVal = item3;
        }
        return minVal;
    }
}
```


Writing and maintaining redundant classes that only differ by data type can be time-consuming and error-prone. The language supports a better approach.

A ***generic class*** is a class definition having a special type parameter that may be used in place of types in the class. A variable declared of that ***generic*** class type must indicate a specific type.

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 5.7.2: A generic class enables one class to handle various data types.

TripleItem.java:

```
public class TripleItem <TheType extends Comparable<TheType>> {
   private TheType item1; // Data value 1
   private TheType item2; // Data value 2
   private TheType item3; // Data value 3

   public TripleItem(TheType i1, TheType i2, TheType i3) {
      item1 = i1;
      item2 = i2;
      item3 = i3;
   }

   // Print all data member values
   public void printAll() {
      System.out.println("(" + item1 + "," + item2 + "," + item3 +
")");
   }

   // Return min data member value
   public TheType minItem() {
      TheType minVal;        // Holds min item value, init to first
item

      minVal = item1;

      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }
}
```


TripleItemManager.java:

```
public class TripleItemManager {
   public static void main(String[] args) {

      // TripleItem class with Integers
      TripleItem<Integer> triInts = new TripleItem<Integer>(9999, 5555,
6666);

      // TripleItem class with Shorts
      TripleItem<Short> triShorts = new TripleItem<Short>((short)99,
(short)55, (short)66);

      // Try methods from TripleItem
```

```
        triInts.printAll();
        System.out.println("Min: " + triInts.minItem() + "\n");

        triShorts.printAll();
        System.out.println("Min: " + triShorts.minItem());
    }
}
```



```
(9999,5555,6666)
Min: 5555

(99,55,66)
Min: 55
```

The class name is succeeded by `<TheType ... >`, where TheType can be any identifier. That type is known as a ***type parameter*** and can be used throughout the class, such as for parameter types, method return types, or field types. An object of this class can be instantiated by appending after the class name a specific type in angle brackets, such as `TripleItem<Short> triShorts = new TripleItem<Short>((short)99, (short)55`

Each type parameter can be associated with type bounds to specify the data types a programmer is allowed to use for the type arguments. As with generic methods, type bounds (discussed elsewhere) also allow a programmer to utilize the class members specified by the bounding type with variables of a generic type (e.g., item1, item2, item3, and min). Thus, above, TripleItem is a generic class whose instances expect type arguments that implement the Comparable<TheType> interface. By bounding the generic class's type parameter to the Comparable interface, a programmer can invoke the Comparable interface's compareTo() method with the generic types, as in `item2.compareTo(min)`.

PARTICIPATION ACTIVITY 5.7.1: Generic classes.

1) A class has been defined using the type GenType throughout, where GenType is intended to be chosen by the programmer when declaring and initializing a variable of this class. The code that should immediately follow the class's name in the class definition is `<GenType>`

- ○ True
- ○ False

2) A key advantage of generic classes is relieving the programmer from having to write redundant code that differs only by type.
   - True
   - False

3) For a generic class with type parameters defined as `public class Vehicle <T> { … }`, an appropriate instantiation of that class would be `Vehicle<T> v1 = new Vehicle<T>();`.
   - True
   - False

A generic class may have multiple type parameters, separated by commas. Additionally, each type parameter may have type bounds.

Construct 5.7.1: Generic class template with multiple parameters.

```
public class ClassName <Type1 extends BoundType1, Type2 extends
BoundType2> {
   ...
}
```

Importantly, type arguments cannot be primitive types such as int, char, and double. Instead, the type arguments must be reference types. If primitive types are desired, a programmer should use the corresponding primitive wrapper classes (e.g., Integer, Char, Double, etc.), discussed elsewhere.

Note that Java's ArrayList class is a generic class, which is why a variable declared as an ArrayList indicates the type in angle brackets, as in `ArrayList<Integer> nums = new ArrayList<Integer>();`.

zyDE 5.7.1: Class generics.

The following program uses a generic class ItemCount to count the number of times same word is read from the user input. Modify the program to:

- Complete the incrementIfDuplicate() method and update the main() method wi DuplicateCounter class to use the incrementIfDuplicate() method.
- Modify the program to count the number of times a specific integer value is rea the user input. Be sure to use the Integer class.

Current file: **DuplicateCounter.java** Load default ter

```
1
2  import java.util.Scanner;
3
4  public class DuplicateCounter {
5     public static void main(String[] args) {
6        Scanner scnr = new Scanner(System.in);
7        ItemCount<String> wordCounter = new ItemCount<String>();
8        String inputWord;
9
10       wordCounter.setItem("that");
11
12       System.out.println("Enter words (END at end):");
13
14       // Read first word
15       inputWord = scnr.next();
16
17       // Keep reading until word read equals <end>
```

```
that that is is not that that is not
END
```

Run

CHALLENGE ACTIVITY 5.7.1: Enter the output of class generics.

422352.2723990.qx3zqy7

Start

Type the program's output

PairManager.java | Pair.java

```
public class PairManager {
   public static void main(String[] args) {
      Pair<Integer> twoInts = new Pair<Integer>(8, 46);
      Pair<Double> twoDbls = new Pair<Double>(27.8, 5.5);
      Pair<Character> twoChars = new Pair<Character>('i', 'n');

      System.out.println(twoInts.chooseItem());
      System.out.println(twoDbls.chooseItem());
      System.out.println(twoChars.chooseItem());
   }
}
```


1 | 2

Check Next

Exploring further:

- Introduction to generics from Oracle's Java tutorials

5.8 LAB: What order? (generic methods)

Define a generic method called checkOrder() that checks if four items are in ascending, neither, or descending order. The method should return -1 if the items are in ascending order, 0 if the items are unordered, and 1 if the items are in descending order.

The program reads four items from input and outputs if the items are ordered. The items can be different types, including integers, Strings, characters, or doubles.

Ex. If the input is:

```
bat hat mat sat
63.2 96.5 100.1 123.5
```


the output is:

```
Order: -1
Order: -1
```

422352.2723990.qx3zqy7

LAB ACTIVITY | 5.8.1: LAB: What order? (generic methods) | 0 / 10

WhatOrder.java Load default template...

```
1  import java.util.Scanner;
2
3  public class WhatOrder {
4     // TODO: Define a generic method called checkOrder() that
5     //       takes in four variables of generic type as arguments.
6     //       The return type of the method is integer
7
8
9        // Check the order of the input: return -1 for ascending,
10       // 0 for neither, 1 for descending
11
12
13
14    public static void main(String[] args) {
15       Scanner scnr = new Scanner(System.in);
16
17       // Check order of four strings
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **WhatOrder.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

5.9 LAB: Zip code and population (generic types)

Define a class `StatePair` with two generic types (`Type1` and `Type2`), a constructor, mutators, accessors, and a printInfo() method. Three ArrayLists have been pre-filled with StatePair data in main():

- `ArrayList<StatePair<Integer, String>> zipCodeState`: Contains ZIP code/state abbreviation pairs
- `ArrayList<StatePair<String, String>> abbrevState`: Contains state abbreviation/state name pairs
- `ArrayList<StatePair<String, Integer>> statePopulation`: Contains state name/population pairs

Complete main() to use an input ZIP code to retrieve the correct state abbreviation from the ArrayList zipCodeState. Then use the state abbreviation to retrieve the state name from the ArrayList abbrevState. Lastly, use the state name to retrieve the correct state name/population pair from the ArrayList statePopulation and output the pair.

Ex: If the input is:

```
21044
```

the output is:

```
Maryland: 6079602
```

422352.2723990.qx3zqy7

LAB ACTIVITY 5.9.1: LAB: Zip code and population (generic types) 0 / 10

Current file: StatePopulations.java Load default template...


```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.IOException;
4  import java.util.ArrayList;
5
6  public class StatePopulations {
7
8     public static ArrayList<StatePair<Integer, String>> fillArray1(ArrayList<StatePair<Integ
9                                                                     Scanner inFS) {
10       StatePair<Integer, String> pair;
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **StatePopulations.java** (Your program)

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

5.10 Lab 8 - ArrayStringList

Module 4: Lab 8 - ArrayStringList

Making Your Very Own ArrayList

This lab includes the following .java files:

L7/
└── ArrayStringList.java

└── Main.java*

**Main.java is used for testing and cannot be modified.*

Here is the starter jar with ArrayStringList.java if you would like to code in a different environment: L7.jar.

Fair warning: you may find this lab slightly more difficult than most. Read this writeup in its entirety, and please don't be afraid to ask your TAs for help if you get stuck!

We all know and love the standard Java ArrayList. It's a wonderful alternative to regular Java arrays, the ones that look like this:


```
String[] myStrings = {"hello", "world"};
```

This is great and all, but it's very *static*. Sure, we can change the existing elements, but what if we want to add a third one? Or take one out? There's no easy way to do either of these things, and so working with regular arrays is a bit of a chore.

The ArrayList is an implementation of a **List**. A List is more powerful than an array.

You can add elements to the List with `.add()`, you can take them out with `.remove()`, you can even see if a certain element is in the list with a single call to `.contains()`. In fact, the list of everything a List can do is quite extensive, ranging from iteration to replacement to filtering.

An ArrayList is a class that supports all of these awesome List features. But how does it do it? In this lab, you'll learn more about how the ArrayList works **internally** by creating a simpler one of your own!

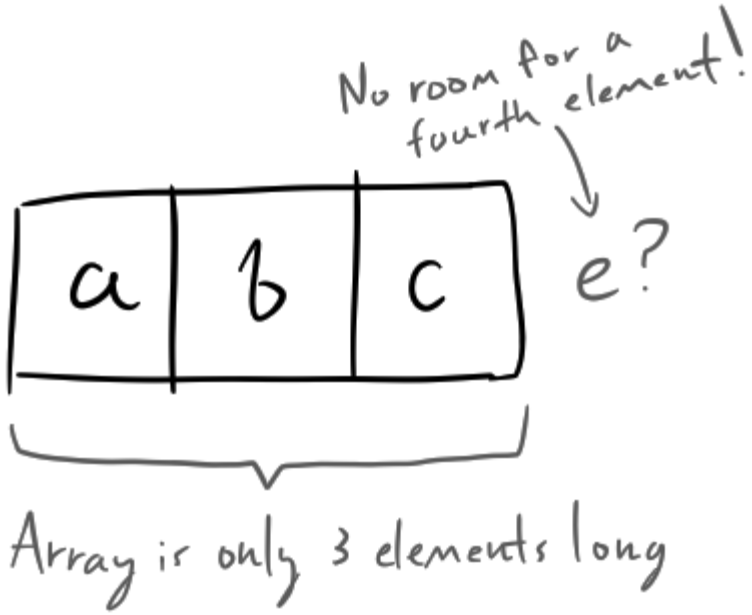
How the ArrayList Works

As the name would suggest, an ArrayList works by internally storing a regular **array** of values. When a method like `.add()` or `.remove()` is called on an ArrayList, it manipulates this internal array. If something like a `.get()` is called on an ArrayList, it gets the value from the array. Indices in the List can map directly to indices in the array; for instance, if you want to get the item at index 4 in the List, that's the same thing as index 4 in the internal array. If you want to add an item to the end of the list, just add it to the first free index in the array. Seems simple enough!

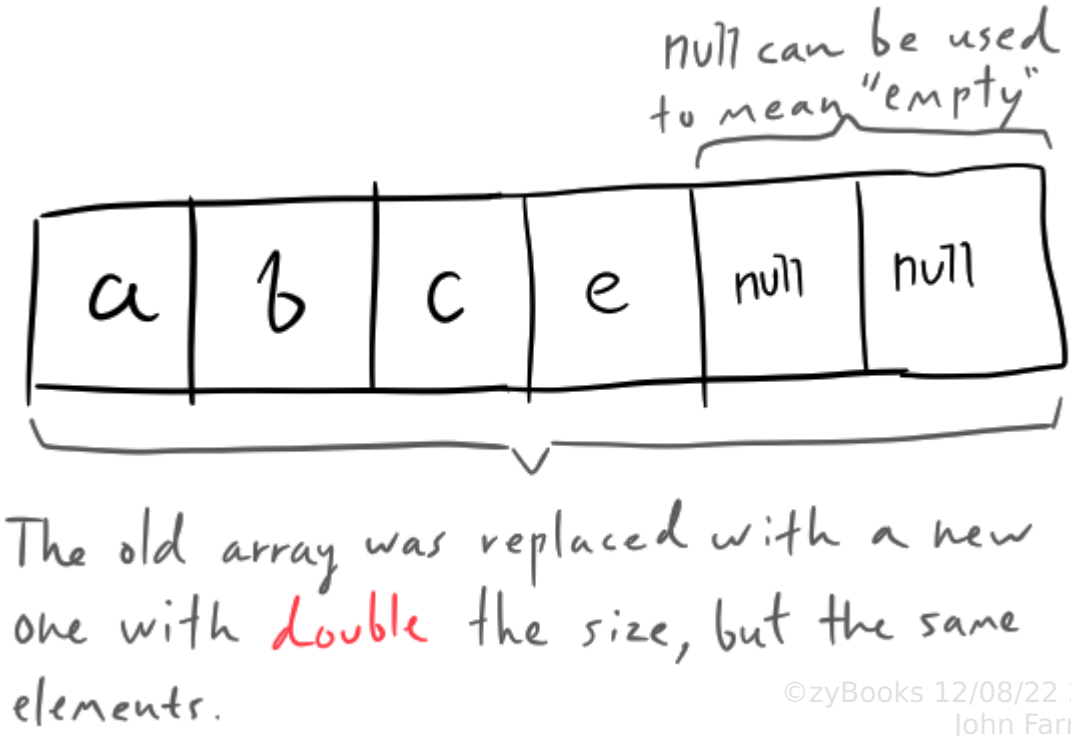
But there are several issues to using an array to implement a List like this. One of the features of a List is that adding to it should *always* work. A List is never full, but arrays *do* get full eventually. They can only store so many elements. How do we use an array to make a List, then?

The solution is simple in concept: *resize the array if it gets too full.*

Imagine this is our internal array. It has a size of three, and each of those slots is already full.



We are asked to add a fourth element, *e*, to the List. This should be possible, because a List is never too full. However, our internal array *is* full. Our solution is to **double** the size of the array, while keeping the existing elements intact.



We now have space for the fourth element, and so we added it in. Hooray!

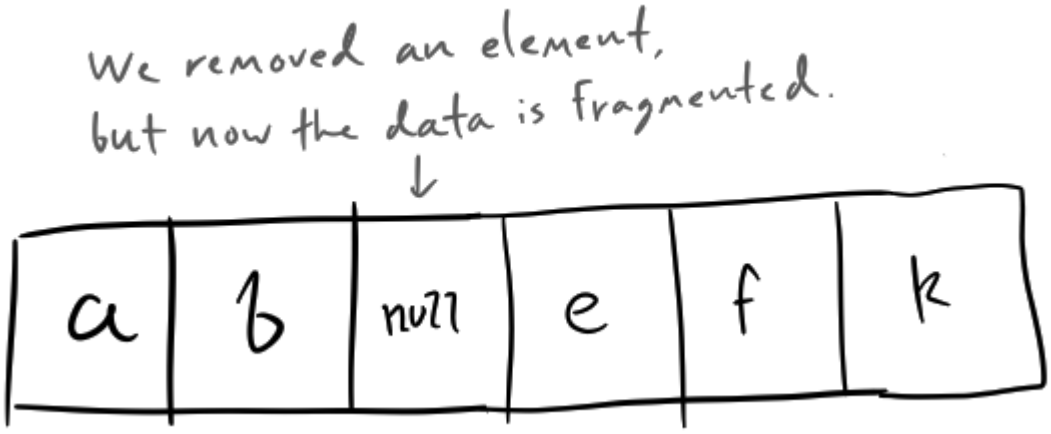
This is how you should implement your ArrayList with a resizing array. Every time `.add()` is called, see if the element will fit in the array. If it will, put it in the next available slot. If it will not, resize the array by creating a new one with double the size, copying all of old elements over, then putting the new element in the next available slot.

Shuffling things around

Another challenge of implementing a List with an array is data contiguity.

In a List, we expect data to be **contiguous**, at least with regards to their indices. This means there should be no "gaps" in the data; if there is an element at index 3 and one at index 5, there must be one at index 4.

The problem is that this is **not** enforced in arrays. If we remove an element in the middle of the array, we now have a hole in the array. This is bad, and the array doesn't fix it for us.



When you remove elements, or add new elements into the middle of an array, you will have to slide the data around so it remains contiguous. Your challenge is to figure out how to keep the data in your ArrayList contiguous.

About the lab code

The lab code is fairly self-explanatory: it's a single class, the ArrayStringList, and you will be completing the following methods:

- resizeData
- ArrayStringList
- add (String)
- add (index, String)
- get
-remove
- size
- contains

If you implement everything correctly, you should have a fully functional List that contains strings, implemented with an internal array!

You may have noticed that The ArrayStringList doesn't implement the entire List interface. That's

because, if you look at the List interface, it's expansive to say the least. We're not going to have you implement *all* of these methods, only a subset of them. So in reality we're not making a List, but we're making something pretty close.

Submission

In Submit mode, select "Submit for grading" when you are ready to turn in your assignment. If you successfully implement everything, running your code will give you this output:


```
If you're reading this, everything worked!
```

422352.2723990.qx3zqy7

LAB ACTIVITY | 5.10.1: Lab 8 - ArrayStringList | 0 / 7

Downloadable files

`ArrayStringList.java` and `Main.java` **Download**

Current file: **ArrayStringList.java** Load default template...

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  public class ArrayStringList {
5
6      /* This field is really important!
7       * This is the internal array of data you're going to use to implement the
8       * ArrayStringList. This is what actually STORES the Strings in your list.
9       * More information about how this should be used is in the lab writeup.
10      * Read it first!
11      */
12     private String[] data;
13
14     /* Storing the amount of valid Strings that are in the array turns out to be
15      * fairly useful. This variable is for that.
16      */
17     private int size;
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → ArrayStringList.java (Your program)

Program output displayed here

Coding trail of your work What is this?

```
9/20 T---- min:13
```

5.11 Lab 8 - GenericArrayList

Module 4: Lab 8 - GenericArrayList

How Generic!

This lab requires you have a working ArrayStringList from the last lab. Finish that lab before you tackle this one.

Download the lab materials here:

L8.jar.

The following .java files are included in this lab:

L8
├── GenericArrayList.java **(Main class in zyBooks)**
├── GenericArrayListPt2.java
├── Point.java (Read-only on zyBooks)
├── Point3D.java (Read-only on zyBooks)

In Lab 7, we made a simple ArrayList that only works with Strings. But what if we want to store something else, like Integers, or arrays, or our own custom classes? The solution is generics!

In this lab, you will be giving your old ArrayStringList a *type parameter*, so that anyone who makes an object of that class can choose what type they want to store. This is what Java's standard

ArrayList does - whenever you type something like ArrayList<String>, you are basically asking Java to create a version of ArrayList that only works for Strings. The ArrayList class, then, becomes a sort of template for other, more specific classes.

Using the code

You'll notice two unfamiliar classes in the jar file we provide; Point and Point3D. These create a simple inheritance hierarchy. You'll also notice both implement the Comparable<Point> interface, which is to say they can both be *compared* to a Point to get some ordering.

You won't be writing code in either of these classes, and the implementations of their methods isn't very important. The important thing to understand for now is how they relate to one another.

You will be working in the GenericArrayList and GenericArrayListPt2 classes for this lab. Follow the instructions below.

GenericArrayList

For the **first part** of this lab, copy your working ArrayStringList code into the GenericArrayList class. Then, modify the class so that it can store *any* type someone asks for, instead of only Strings. You shouldn't have to change any of the actual logic in your class to accomplish this, only type declarations (i.e. the types of parameters, return types, etc.)

Note:

In doing so, you may end up needing to write something like this (where T is a generic type):

```
T[] newData = new T[capacity];
```

...and you will find this causes a compiler error. This is because Java dislikes creating new objects of a generic type. In order to get around this error, you can write the line like this instead:

```
T[] newData = (T[]) new Object[capacity];
```

This creates an array of regular Objects which are then cast to the generic type. It works, and it doesn't create an error in the Java compiler. How amazing!

You will likely still get warnings depending on how you implement this, however. See question #2 below. You will want to know what these warnings mean.

GenericArrayListPt2

For the **second part** of the lab, modify your GenericArrayList so that it can store any type that is comparable to a Point. Remember the Point and Point3D classes? Both of those implement the Comparable<Point> interface, so they both can compared to a Point. In fact, they are the *only* classes that can be compared to a Point, so after modifying your GenericArrayList, it should only be able to contain these two classes.

In both parts, test your classes by following the directions in the comments. They will ask you to uncomment some code and look for a specific result. *(Note: only the main in GenericArrayList will run in zyBooks.)*

Questions to think about:

1. Why can't you write something like the following in GenericArrayListPt2?
   `GenericArrayList<Float> floatList = new GenericArrayList<Float>(2);`
2. Why might there be unchecked and raw type warnings when you run your code? What do these warnings tell you and why is it important to pay attention to them?

Submission

Submit your code under Submit Mode here in zyBooks.

422352.2723990.qx3zqy7

LAB ACTIVITY | 5.11.1: Lab 8 - GenericArrayList | 0 / 7

Downloadable files

`GenericArrayList.java` , `Point.java` , `Point3D.java` , and `GenericArrayListPt2.java`

Download

Current file: GenericArrayList.java

```
1 Loading latest submission...
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → GenericArrayList.java (Your program) -

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

5.12 LAB: Pairs (generic classes)

This section's content is not available for print.

5.13 LAB: Min, max, median (generic methods)

This section's content is not available for print.

5.14 LAB: Students (generic class)

This section's content is not available for print.

©zyBooks 12/08/22 21:50 1361995
John Farrell
COLOSTATECS165WakefieldFall2022