CMSC131

Intro to Generic Data Structures: ArrayList

A Simple Problem?

What if I asked you to write a program that would allow the user to specify how many numbers they wanted to enter, then read them in, and then be able to print them out backwards?

Which of the following terms have you heard?

- 0% A. Stack
- 0% B. Queue
- 0% C. List
- 0% D. Array
- 0% E. Data Structure

General Purpose Data Structures

Imagine creating some sort of "list" class with the ability to add things, remove things, have it grow and shrink as needed, search for things, etc. The type of information being held in the list should not matter much and we should not have to write a new version for each possible type of data it would hold

- Again, we've seen that Java needs everything we declare to have a specific type, and so we will use **Generics** again!

`ArrayList<Type>`

A useful polymorphic data structure provided by Java is a general-purpose, array-based, resizable list called an **`ArrayList`** that can hold any type of object you specify.

- It is a **Generic** structure, and you can specify what **`<Type>`** of references a specific instance of **`ArrayList`** will be allowed to hold.
- This can be better in the long run than a data structure designed to simply allow references to ***any*** type of object to be stored since this array will allow for **compile-time type checking** and make it easier to invoke methods using the objects that were stored in the list.

`ArrayList` Behaviors/Properties

Once created, an **`ArrayList`** object can use instance methods to do things such as add, remove, and search.

It has similarities to the **`StringBuffer`** class in terms of memory management behind the scenes.

- An initial internal list size is allocated, the object keeps track of how much of that space has been used, and when the object runs out of room, the class takes care of allocating a larger sized list, copying things from the smaller list space into that larger list space, and then goes on as before.

`ArrayList` Differences

While similar to the `StringBuffer` class in terms of memory management, there are diffferences.

- Unlike with `StringBuffer`, we can not access the current capacity information.
- We are provided with an instance method `ensureCapacity(#)` that can be used by a programming before they are about to do a large number of additions; it will grow the internal structure to at least that size and copy things over in a single operational unit.

Declaring and creating an `ArrayList`

```
//Declare a reference.
ArrayList<Double> arrName;

//Create a new ArrayList object and
//  store the reference.
arrName = new ArrayList<Double>();
```

Adding references to an `ArrayList`

The **`ArrayList`** has instance methods that allow you to add new items to the end of the list.

```
//Directly with literal values
//(Java will box the literal values)
arrName.add(1.1);
arrName.add(2.3);
arrName.add(37.1);

//Using Double objects we explicitly create
Double d = new Double(3.14);
arrName.add(d);
```

Another way to add an item

Another way that you can add an item to an **`ArrayList`** is to specify a position within the list.

- It will then shift anything at that position or beyond over one additional space to make room for the new value, and then insert the new value.

```
arrName.add(1.1);
arrName.add(2.3);
arrName.add(0, 37.8);
System.out.println(arrName);
```

`ArrayList` adds are "shallow" ones

```
//The following will result in a list
//  with two values stored, but those
//  values will both be references to
//  the same Float object on the heap.
Double d = new Double(3.14);
arrName.add(d);
arrName.add(d);

//Since Double objects are immutable,
//  not an issue.
```

Result of "shallow" adds…

```
ArrayList<Student> myStudents;
myStudents = new ArrayList<Student>();
Student one = new Student("AAA");
  myStudents.add(one);
  myStudents.add(one);
  System.out.println(myStudents);

  one.setUID(123456789);
  System.out.println(myStudents);
//Student is mutable - is the behavior
//what you wanted?
```

Removing references from an `ArrayList`

There are two ways to remove an individual item from an **`ArrayList`**; using a reference to the object in the list that you want to remove or specifying its 0-based index position number in the list.

- While they both would remove an item from the list, they have different return values; removing by providing a reference to the object returns a **`boolean`** value indicating whether that item was in the list but if done by specifying the position whose value to remove it returns a reference to the object that was removed.

Code: removing items

```
ArrayList<Student> arrName;
arrName = new ArrayList<Student>();

Student one = new Student("BBB");
Student two = new Student("CCC");
Student three = new Student("AAA");

arrName.add(one); arrName.add(two); arrName.add(three);
System.out.println(arrName);

arrName.remove(1); System.out.println(arrName);

arrName.remove(three); System.out.println(arrName);
```

Using the `contains` method

Another instance method is **`contains`** which takes a reference to an object and returns a **`boolean`** value indicating whether that object is in the list.

- What do you think will be printed by the following code segment? True or False?

```
Student one = new Student("AAA");
  myStudents.add(one);
Student seek = new Student("AAA");
System.out.println(myStudents.contains(seek));
```

Using the `get` method

Another instance method is **`get`** which takes a 0-based index position and returns a reference to the object at that position.

- Note that this is a reference to the object in the list, not a copy of that object, so any changes we make will impact the contents of the list.

```
Student ref = myStudents.get(3);
ref.setUID(987654321);
//The object referenced from within the
//  list has its UID altered.
```

Type checking: `ArrayList<Float>`

```
ArrayList<Float> arrName;
arrName = new ArrayList<Float>();

//These lines would NOT compile.
arrName.add("hi"); //String not Float
arrName.add(17.6); //Double not Float

//We could declare a Student
Student s = new Student();
//But we could not add it to this list.
arrName.add(s);    //Student not Float
```

Length of an `ArrayList`

Every time a successful add or remove is performed, the logical size of an **`ArrayList`** changes.

- The current logical length can be obtained using the **`size()`** instance method.

The actual length behind the scenes cannot be accessed. We can avoid some inefficiency in how the class grows the list by calling that **`ensureCapacity(#)`** method before a large number of additions are to be done.

Copying an ArrayList

```
//NOTE: This is a shallow copy of the list.
ArrayList<Integer> newArr;
newArr = new ArrayList<Integer>(oldArr);
```

What will Fred's score be in **yourList**?

```
ArrayList<Student> myList = new ArrayList<Student>();
Student myStudent = new
Student("Fred","Fred",123121234);
myList.add(myStudent);
myStudent.setScore(80);
ArrayList<Student> yourList = new
ArrayList<Student>(myList);
myStudent.setScore(50);
```

0% A. 0

0% B. 50

0% C. 80

0% D. Something else.

Response Counter

Iterating through an `ArrayList`

We can iterate through an **`ArrayList`** using a standard **for** loop.

```
int length = arrName.size();
for (int i=0; i<length; i++) {
   //process the object that arrName.get(i)
   //  returns during a given loop iteration
}
```

Another way we can iterate…

However, the **`ArrayList<Type>`** class is a Java **Collection** and this means that we can iterate through each of the individual elements of an **`ArrayList<Type>`** object using the syntax of a "for each" loop:

```
for (Typename iteratedVal : collection) {
   //process the object to which iteratedVal
   //  refers at the moment
}
```

"for each" example:

```
for (Integer val : arrName) {
    System.out.print(val + " ");
}
System.out.println();
```

NOTE: ***You cannot alter the structure of a list while iterating through it using a for-each style loop.***

If you want to perform that type of operation, you would need to create a duplicate of the list and iterate through that one while altering the other.

One way to delete all **even** numbers

```
ArrayList<Integer> shadow;
shadow = new ArrayList<Integer>(actual);
for (Integer val : shadow) {
  if (val%2 == 0) {
     actual.remove(val);
  }
}
```

NOTE: Since the data type of **`val`** is **`Integer`** rather than **`int`**, the correct **`remove`** (the one that takes a reference to the object to remove) is called. The remove call returns a **`boolean`** value but we have no use for it in this context so ignore it.

What will print at the end?

- 0% A. 0,1,2,3,4,5,6,7,8,9
- 0% B. 1,3,5,7,9
- 0% C. It won't compile.
- 0% D. It will crash before it gets to printing.

Response Counter

```
ArrayList<Integer> arrName =
                new ArrayList<Integer>();
for (int i=0; i<10; i++) {
     arrName.add(i);
}
for (Integer val : arrName) {
     if (val%2 == 0) arrName.remove(val);
}
System.out.println(arrName);
```

Making a **deep copy** of an `ArrayList`

```
//Assuming origList has already been created
//   and populated.

ArrayList<StringBuffer> dupList;
dupList = new ArrayList<StringBuffer>();
for (StringBuffer str : origList) {
   dupList.add(new StringBuffer(str));
}
```

Using **Iterator** objects

We can iterate through each of the individual elements of an **`ArrayList<Type>`** object explicitly using an **`Iterator`**:

```
Iterator<Type> valIter = list.iterator();
while (valIter.hasNext()) {
   Type curr = valIter.next();
   //process curr as desired
   //NOTE: ArrayList iterator will allow
   // processing to include remove calls
   // without it causing an exception.
}
```

Sorting a list of items

There are times when we have an unordered list of items but want to sort them based on some field that has a natural ordering to it.

There are a wide variety of algorithms that can be used to sort items and they have their pros and cons. However, what if we didn't want to have to write one and were happy to let the authors of the **`ArrayList`** class provide one for us?

- All we would need to do is tell it how to decide the relative order of two items.

Using the sort method in `ArrayList`

```
ArrayList<Student> arrName;
arrName = new ArrayList<Student>();

        Student one = new Student("BBB");
        Student two = new Student("CCC");
        Student three = new Student("AAA");

        arrName.add(one);
        arrName.add(two);
        arrName.add(three);
        System.out.println(arrName);

        arrName.sort(Student::compareTo);
        System.out.println(arrName);
```