Lecture - ArrayList

ArrayList

`ArrayList` is a class in the standard Java libraries

Unlike arrays, which have a fixed length once they have been created, an `ArrayList` is an object that can **grow** and **shrink** while your program is running

In general, an `ArrayList` serves the same purpose as an array, except that an `ArrayList` can change length while the program is running.

The class `ArrayList` is implemented using an array as a private instance variable. When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array, like in the class we wrote in the previous module.

Why not always use an ArrayList instead of an array?

Don't know the size of array.

- An `ArrayList` is less efficient than an array
- It does not have the convenient square bracket notation
- The base type of an `ArrayList` must be a class type or interface type (or other reference type): it cannot be a primitive type. This last point is less of a problem now that Java provides automatic boxing and unboxing of primitives.

Using the ArrayList class

In order to make use of the `ArrayList` class, it must first be imported from the package `java.util`

An `ArrayList` is created and named in the same way as object of any class, except that you specify the base type in angle brackets ("less than" and "greater than") as follows:

```
ArrayList<Double> aList = new ArrayList<>();
```

Compare with array:

```
double[] score = new double[5];
```

An initial capacity can be specified when creating an ArrayList as well

– The following code creates an `ArrayList` that stores objects of the base type String with an initial capacity of 20 items

```
ArrayList<String> list = new ArrayList<String>(20);
```

Specifying an initial capacity does not limit the size to which an `ArrayList` can eventually grow
Note again that the base type of an `ArrayList` is specified as a type parameter.

The `add` method is used to set an element for the first time in an `ArrayList`

```
list.add("something");
```

This assigns to the first unassigned element of the underlying array.

The method name `add` is overloaded

- There is also a two argument version that allows an item to be inserted at any currently used index position or at the first unused position.
- This is *insertion*: the elements at or after the specified position are moved up to make room for the new value. The old value is not overwritten as it would be when assigning to an array.
- To overwrite the element at idx, use

```
list.set(idx, "something");
String s = list.get(idx);
```

The size method is used to find out how many indices already have elements in the ArrayList

```
int howMany = list.size();
```

Java uses three methods for reporting sizes of objects.
For arrays: member variable int length
For strings: method int length ()
For containers: method int size ()

Predict what the following code will do before running it.

```
import java.util.ArrayList;

class Main {
    public static void main (String[] args) {
        ArrayList<String> list = new ArrayList<String> (20);
        list.add("one");
        list.add("two");
        list.add(2, "three");
        list.add(0, "zero");
        list.set(3, "Three");

        for (String item : list)
            System.out.println(item);
        System.out.println(list.size());
    }
}
```

```
String list2[] = new String[20]; // this is an equivalent to above;
list.add("one");
list2[0] = "one";  // equivalent
list.add("two");
list2[1] = "two";  //equivalent
list.add(2, "three");
list2[2] = "three"; // equivalent to above
list.add(0, "zero");
for(int i = 0; i < list2.length; i++){
    if(list2[i+1] != null){
        list2[i+1] = list2[i];
    }else{
        break;
    }
```

Summary

`add(item)` : append item to the end of the list

`add(idx, item)` : insert item at location idx, and move up all elements after idx to the next higher position.

`set(idx, item)` : overwrite the data at idx, which must already be present in the list.

`int size()` : return the number of elements in the list -- not the number pre-allocated in the constructor.

for-each loop

As we saw previously, it is possible to loop over all elements of a container, just like over elements of an array:

```
ArrayList<String> foo = new ArrayList<String> ();
foo.add("one");
foo.add("two");
for (String s : foo)             // for-each loop
    System.out.println(s);
```

Methods in the class ArrayList

The tools for manipulating arrays consist only of the square brackets and the instance variable `length`.

`ArrayLists`, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays.

Constructors

```
public ArrayList<BaseType>(int initialCapacity)
public ArrayList<BaseType>()
```

Creates an empty `ArrayList` with the specified `BaseType`. If `initialCapacity` is omitted, 10 is used.

Array-like methods

```
public BaseType set (int index, BaseType newElement)
```

Sets the element at the specified index to newElement. Returns the element previously at that position, but the method is often used as if it were a void method. (In java, return values can be ignored.) If you draw an analogy between the `ArrayList` and an array `a`, this function is analogous to setting `a[index]` to the value `newElement`. We need `0 <= index <= ArrayList.size()`. Throws an `IndexOutOfBoundsException` if the index is not in this range.

public BaseType get (int index)

Returns the element at the specified `index`. This statement is analogous to returning `a[index]` for an array `a`. We need `0 <= index < ArrayList.size()`. (Note that the second is `<` not `<=`.) Throws an `IndexOutOfBoundsException` if the index is not in this range.

Inserting elements

```
public void add (int index, BaseType newElement)
```

Inserts `newElement` as an element of the calling `ArrayList` at the specified index. Each element in the `ArrayList` with an index greater than or equal to `index` is shifted upward to have an index that is one greater than the value it had previously. The `index` must be a value greater than or equal to 0 and less than ***or equal to*** the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range. Note that you can use this method to add an element after the last element. The capacity of the `ArrayList` is increased if that is required.

```
public boolean add(BaseType newElement)
```

Equivalent to `v.add(v.size(), newElement)`.

Adds `newElement` to the end of the calling `ArrayList` and increases the `ArrayList`'s size by one. The capacity of the `ArrayList` is increased if that is required. Returns true if the add was successful.

Deleting elements

```
public BaseType remove (int index)
```

Deletes and returns the element at the specified `index`. Each element of the ArrayList with an index greater than `index` is decreased to have an index that is one less than the value it had previously. The `index` must be a value greater than or equal to 0, and less than the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range. Often used as if it were a `void` method.

```
protected void removeRange (int fromIndex, int toIndex)
```

Deletes all the elements with indices *i* such that `fromIndex` <= i < `toIndex`. Elements with indices greater than or equal to `toIndex` are decreased appropriately.

```
public boolean remove (Object theElement)
```

(Treat this as if `Object` were the base type of the `ArrayList`. This will become clear next lecture.)

Removes the first occurrence of `theElement` from the calling `ArrayList`. If `theElement` is found in the `ArrayList`, then each element in the `ArrayList` with an index greater than the removed element's index is decreased to have an index that is one less than the value it had previously. Returns `true` if `theElement` was found (and removed). Returns `false` if `theElement` was not found in the calling `ArrayList`.

```
public void clear ()
```

Removes all elements from the calling `ArrayList` and sets the `ArrayList`'s size to 0. This does not reduce the `ArrayList`'s capacity (i.e., the amount of memory allocated to it). See `trimToSize()` below for that.

Searching

(Treat this as if `Object` were the base type of the ArrayList. This will become clear next lecture.)

```
public boolean contains (Object target)
```

Returns `true` if the calling `ArrayList` contains target; otherwise, returns `false`. Uses the method

`equals` of the object `target` to test for equality with any element in the calling `ArrayList`.

```
public int indexOf (Object target)
```

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns -1 if `target` is not found.

```
public int lastIndexOf (Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns -1 if `target` is not found.

Finding the how many elements there are

```
public int size()
```

Returns the number of elements in the calling `ArrayList`.

```
public boolean isEmpty()
```

Equivalent to `size() == 0`. Using `isEmpty` makes the intention clearer to the reader. For some other container classes, it may also be faster, and so using it makes it easier to refactor code to use another container.

Resizing the underlying array

```
public void ensureCapacity (int newCapacity)
```

Increases the capacity of the calling `ArrayList`, if necessary, in order to ensure that the `ArrayList` can hold at least `newCapacity` elements. Using `ensureCapacity` can sometimes increase efficiency (as it can avoid increasing the capacity multiple times and elements are added sequentially), but it is not needed for any other reason.

```
public void trimToSize()
```

An `ArrayList` automatically increases its capacity when needed. However, the capacity may increase beyond what a program requires. In addition, although an `ArrayList` grows automatically when needed, it does not shrink automatically.

If an `ArrayList` has a large amount of excess capacity, an invocation of the method `trimToSize` will shrink the capacity of the `ArrayList` down to the size needed.

Copies

```
public Object[] toArray ()
```

Returns an array containing all the elements in the list. Preserves the order of the elements.

```
public Type[] toArray( Type[] a)
```

Returns an array containing all the elements in the list. Preserves the order of the elements. `Type` can be any class type. If the list will fit in `a`, the elements are copied to `a` and `a` is returned. Any elements of `a` not needed for list elements are set to `null`. If the list will not fit in `a`, a new array is created and returned.

(It is explained in Setion 14.2 of the text book that the correct Java syntax for this method heading is `public <Type> Type[] toArray(Type[] a)` but you can treat it as if it is the simpler heading.)

```
public Object clone()
```

Returns a shallow copy of the calling ArrayList. Warning: The clone is not an independent copy. Subsequnt changes to the clone may affect the calling object and vice versa. (See Chapter 5 of the text book for a discussion of shallow copy.)

```
public boolean equals (Object other)
```

If `other` is another `ArrayList` (of any base type), then equals returns `true` if and only if both `ArrayLists` are of the same size and contain the same list of elements in the same order. (In fact, if other is any kind of list, then `equals` returns `true` if and only if both the `ArrayList` and other are of the same size and contain the same list of elements in the same order. Lists are discussed in Chapter 16 of the text book.) Does not require the capacities to be equal.

```java
public class Person{

    public int id;
    public String name;

    public Person(int id, String name){
        this.id = id;
        this.name = name;
    }
}
```

```java
import java.util.ArrayList;

public class ArrayListPerson{

    public static void main(String[] args){

        ArrayList<Person> pList = new ArrayList<>();
        pList.add(new Person(1, "Me"));
        pList.add(new Person(2, "You"));

        for(Person p : pList){
            System.out.println(p.name);
        }
```

```java
ArrayList<Person> pList = new ArrayList<>();
Person person1 = new Person(1, "Me");
Person p2 = new Person(2, "You");

pList.add(person1);
pList.add(p2);

for(Person p : pList){
    System.out.println(p.name);
}

pList.remove(p2);
```

another example

remove p2 directly.

```java
}

public static Person searchByName(String name, ArrayList<Person> pList){
    Person p = null;
    for(Person p: pList){
        if(p.name.equalsIgnoreCase(n))
    }
```

Example: To do list

Write a to-do list object that will read lines from standard input (ending when a blank line is entered).

Enter these into an ArrayList, in the order they are typed.

Then print them all out, in order.

Extra challenge:

After the user has entered the list, allow them to remove items.

Place the output line within a while loop that continues while the ArrayList is not empty.

After outputting the list, ask the user to enter an entry to remove.

Remove it and continue the next iteration of the loop.

Example: Score keeper

Fill in the blanks here to use an ArrayList of doubles to calculate basic statistics of some sports scores.

(Cricket scores can have two numbers: <runs> for <wickets>. If you are *super* keen, you can use an ArrayList<ArrayList<double> > to keep track of both. I'm not that keen, so the solution doesn't do this.)