WIT COMP1000

OOP Case Study: ArrayList

Dynamically Sized Arrays

- So far, when we wanted to store many values of the same type, we used an array
- However, we have seen that with arrays, we need to know the size ahead of time, and can't adjust later
- The **`ArrayList`** class contains an array, and supports array-like methods, but can grow and shrink as necessary

Creating an ArrayList

```
ArrayList<Type> a = new ArrayList<Type>();
```

- Like an array, when you create an ArrayList, you provide a data type for all elements, via the **`<Type>`**, which *must be a class*
  - ArrayList is an example of a class that can be parameterized by a type, known as a *generic* class
- You must import ArrayList from **`java.util`**

What About Primitives?

- Thankfully Java has built in "wrapper" classes for all primitive types
  - The class has a single field (type=primitive)

Primitive Type	Wrapper Class
boolean	Boolean
char	Character
int	Integer
double	Double

Examples of Creating an ArrayList

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

ArrayList<Double> b = new ArrayList<Double>();

ArrayList<String> c = new ArrayList<String>();

…
```

ArrayList Size vs. Capacity

- Once an ArrayList is initialized, it has two key properties:
  - **Size**: how many elements are in the list
    - Default constructor: 0; via `size()` and `isEmpty()`
  - **Capacity**: the size of the internal array
    - Default constructor: 10; not accessible
- It is useful to think of the ArrayList as internally maintaining a partially filled array

Adding Elements

- Add an element to the end of the list via the **add** method

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

System.out.println(a.size()); // 0
a.add(3);
a.add(1);
a.add(4);
System.out.println(a.size()); // 3
```

Resizing Behavior

- Whenever an element is added such that the new size would exceed the capacity of the underlying array, the ArrayList automatically resizes to accommodate, and copies old data
- Basic idea:

```
newArray = new Type[newSize];
for (int i=0; i<oldSize; i++)
   newArray[i] = listArray[i];
listArray = newArray;
```

Resizing Efficiently

- Copying arrays can become computationally expensive

- If you are about to add many elements, use the **`ensureCapacity(minSize)`** method to have the ArrayList resize to a desired capacity
  - Note: there is also a constructor that can set the initial capacity

Getting/Setting Elements

- To access the value of an existing element, use the **`get(index)`** method
- To change the value of an existing element, use the **`set(index,value)`** method
- For both methods, a **`IndexOutOfBoundsException`** is thrown if… **`(index < 0 || index >= size())`**

Example

```
ArrayList<Character> a = new ArrayList<Character>();

a.add('h');
a.add('i');
a.add('j');
a.set(2, '!');

System.out.printf("%c%c%c%n",
                  a.get(0), a.get(1), a.get(2));
```

Exercise

Write a program that asks the user for a list of integers (no limit!). When the user ends the list (via the EOF signal), output the sequence of numbers they entered in reverse, each number squared.

```
Enter Numbers: -1 3 -10 12

144 100 9 1
```

Answer

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

public class ClassExamples {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        ArrayList<Integer> a = new ArrayList<Integer>();

        System.out.print("Enter Numbers: ");
        while (input.hasNextInt()) {
            a.add(input.nextInt());
        }

        for (int i=a.size()-1; i>=0; i--) {
            System.out.printf("%d ", a.get(i)*a.get(i));
        }
        System.out.println();
    }
}
```

Removing Elements

- You can erase elements from the list via the **`remove`** and **`clear`** methods
  - **`clear()`**
  - **`remove(index)`**
  - **`remove(value)`**
    - Only first occurrence; returns **`true`** if list changed
- Note: removal requires copying all elements that *follow* the removed index(es), and can thus be slow in large lists

Shrinking the List

- The **`trimtoSize()`** method reduces the capacity of the ArrayList to the current size, thereby saving memory

```
list.clear() // size = 0, capacity = ?
list.trimToSize() // capacity = 0
```

Example

```
public static void printList(ArrayList<Character> l) {
    for (Character c : l) {
        System.out.print(c);
    }
    System.out.println();
}

public static void removeAll(ArrayList<Character> l, Character c) {
    while (l.remove(c));
}

public static void main(String[] args) {
    ArrayList<Character> word = new ArrayList<Character>();
    word.add('h');
    word.add('e');
    word.add('l');
    word.add('l');
    word.add('o');
    printList(word); // hello (size=5, capacity=10)
    removeAll(word, 'l');
    word.trimToSize();
    printList(word); // heo (size=3, capacity=3)
}
```

Wrap Up

- The ArrayList class has useful methods to allow you to grow and shrink an array of elements

- There are many aspects of the ArrayList we didn't cover, and many other useful classes
  - Stay tuned for CS2 :)