ArrayLists, Generics

A data structure is a software construct used to organize our data in a particular way. Some common data structures include lists, stacks, queues, and heaps. Dynamic data structures are those data structures that can grow as needed while a program is running. First we will look at a useful class provided by Java called an **ArrayList**. This class uses a feature called **generics or templates** to allow us to create lists of arbitrary data types. Then we will see how to create our own data structures, in particular, how to create dynamically linked lists.

ArrayLists

The ArrayList class provided by Java is essentially a way to create an array that can grow or shrink in size during the execution of a program. If you recall the section on arrays, once we declared an array to be a particular size we were not able to change that size. For example, in our trivia game we declared the array to be of size 5. This means we have a maximum of 5 questions and can never exceed that amount. This is because Java allocates a specific amount of memory to hold exactly the number of items we initially specified. We could get around that problem by creating a new array of the size we like and copy the elements we want into it, but this approach is time consuming. The arraylist class does the dirty work for us to give us the same effect as a dynamic array.

If arraylists do the same thing as arrays but are dynamic, why not use them all the time? We could (and some people do) but there are two good reasons to use arrays over arraylists:

1. Arrays are more efficient than arraylists
2. The elements in an arraylist must be objects (which contributes to #1 for simple items)
3. We can't use the convenient [] notation on arraylists

If we wanted an arraylist of int's, then instead we would have to make an arraylist of class Integer, which is a wrapper class for objects of type int (or we could make our own class).

Here is how we use an arraylist. To access the arraylist code we can import the class via:

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

To create a variable of type ArrayList that is a list of type DataType use the following:

```
ArrayList<DataType> varName = new ArrayList<DataType>();
```

In newer versions of Java you can omit the data type on the right hand side, resulting in the slightly shorter:

```
ArrayList<DataType> varName = new ArrayList<>();
```

This uses a feature called type inference.
If you know how many items the arraylist will likely need, execution can be a bit faster by specifying an initial capacity. Note however that we can still change the size later during runtime if we want to. The example below initializes the arraylist with 50 elements:

```
ArrayList<Integer> a = new ArrayList<>(50);
```

Here are methods to manipulate data in an arraylist:

public boolean add (BaseType newElement)
Adds a new object to the end of the arraylist.

Recall that all classes are derived from class Object, therefore all classes are supported as parameters for this method.

public void add(int index, BaseType newElement)
Inserts the newElement at position index and pushes everything else after this object down by one in the arraylist.

public void set(int index, BaseType newElement)
Sets an existing element at position index to newElement.
Index starts at 0.

An element at this index must previously exist (i.e. can't use to add to a new position)

public BaseType get(int index)
Returns the object at position index
Index starts at 0.

public BaseType remove(int index)
Deletes the element at position index and moves everything else after this object up by one in the arraylist. Returns the object removed.

public int indexOf(Object target)
Returns the index of the first element equal to target or -1 if not found. This method invokes the equals() method of the object, so if your object does not implement and override equals() inherited from class object, most likely this method will not work properly!

public void remove(BaseType element)
Removes element from the arraylist by first searching for it using the equals method. Requires that equals be overridden.

public int size()
Returns the number of elements placed in the arraylist

Here is a basic example:

```
import java.util.ArrayList;

public class Test
{
	public static void main(String[] args)
	{
		ArrayList<String> stringList = new ArrayList<>();

		stringList.add("foo");
		stringList.add("bar");
		stringList.add("zot");
		stringList.add("bah");

		System.out.println("Size of List: " + stringList.size());
		System.out.println("Contents:");
		for (int i = 0; i < stringList.size(); i++)
		{
			System.out.println("At index " + i + " value=" +
				stringList.get(i));
		}

		stringList.remove("bar");
		System.out.println("Contents after remove bar:");
		for (int i = 0; i < stringList.size(); i++)
		{
			System.out.println("At index " + i + " value=" +
				stringList.get(i));
		}

		stringList.remove(1);
		System.out.println("Contents after remove 1:");
		for (int i = 0; i < stringList.size(); i++)
		{
			System.out.println("At index " + i + " value=" +
				stringList.get(i));
		}

		stringList.add(1, "meh");
		System.out.println("Contents after add 1:");
		for (int i = 0; i < stringList.size(); i++)
		{
			System.out.println("At index " + i + " value=" +
				stringList.get(i));
		}

		stringList.set(1, "bar");
		System.out.println("Contents after set 1:");
		for (int i = 0; i < stringList.size(); i++)
```

```
            {
                  System.out.println("At index " + i + " value=" +
                        stringList.get(i));
            }

            System.out.println("Index of bar:");
            System.out.println(stringList.indexOf("bar"));

            System.out.println("Index of zzz:");
            System.out.println(stringList.indexOf("zzz"));
      }
}
```

Here is another simple example. Let's say we would like to make an arraylist of integers. To drive home the point that the arraylist only works with objects, we'll make our own Integer class (but we could have used the built-in Integer class too).

```
import java.util.ArrayList;

// Simple integer class with just two constructors.
// A more robust version would make the int value private with
// accessor methods instead

public class MyIntClass
{
 public int value;

 public MyIntClass()
 {
      value = 0;
 }
 public MyIntClass(int i)
 {
      value = i;
 }
}

// This class illustrates common uses of the arraylist
public class ArrayL
{
      public static void main(String[] args)
      {
            ArrayList<MyIntClass> v = new ArrayList<>();
            int i;

            // First add 4 numbers to the arraylist
            for (i=0; i<4; i++)
            {
                  v.add(new MyIntClass(i));
            }
            // Print out size of the arraylist, should be 4
            System.out.println("Size: " + v.size() + "\n");

            // Print arraylist
            System.out.println("Original arraylist:");
            PrintArraylist(v);
```

```
            // Remove the second element
            v.remove(2);
            // Print out the elements again
            System.out.println("After remove:");
            PrintArraylist(v);

            // Insert the number 100 at position 1
            v.add(1, new MyIntClass(100));
            // Print out the elements again
            System.out.println("After insert:");
            PrintArraylist(v);
      }

      // This method prints out the elements in the arraylist
      public static void PrintArraylist(ArrayList<MyIntClass> v)
      {
            MyIntClass temp;
            int i;

            for (i=0; i<v.size(); i++)
            {
                  temp =  v.get(i);
                  System.out.println(temp.m_value);
            }
            System.out.println();
      }
}
```

Let's try using the indexOf method:

```
            // Index of -1
            System.out.println("Position of 1");
            System.out.println(v.indexOf(new MyIntClass(1)));
```

Adding this gives an output of:

```
      Position of 1
      -1
```

This is not right, the list has a node with the value of 1 in position 2. We are getting -1 back because the `indexOf` method invokes the `equals` method for the BaseType object to determine which item in the arraylist matches the target. However, we didn't define one ourselves, so our program is really using the `equals` method defined for `Object` (which is inherited automatically for any object we make). However, this definition of equals only checks to see if the addresses of the objects are identical, which they are not in this case. What we really need to do is see if the integer values are the same by defining our own `equals` method. Add to the `MyIntClass` object:

```
public boolean equals(Object otherIntObject)
 {
       MyIntClass otherInt = (MyIntClass) otherIntObject;
       if (this.m_value == otherInt.m_value)
            return true;
```

```
        return false;
}
```

If we run the program now, it will output:

```
Position of 1
2
```

A few comments of note: First, we had to define `equals` with an input parameter of type `Object`. This is because this is how the method is defined in the `Object` class which we need to override (and is invoked by the `indexOf` method). This means we have to typecast the object back to our class of `MyIntClass`. Once this is done we can compare the ints.

For Each Loop

Iterating through an arraylist is such a common occurrence that Java includes a special type of loop specifically for this purpose. Called the for-each loop, it allows you to easily iterate through every item in the arraylist. The syntax looks like this:

for (BaseType varName : ArrayListVariable)
{
    Statement with varName
}

Here is a modified version of the PrintArrayList method from the previous example, but using the for each loop:

```
// This method prints out the elements in the arraylist
public static void PrintArraylist(ArrayList<MyIntClass> v)
{
      for (MyIntClass intObj : v)
      {
            System.out.println(intObj.m_value);
      }
      System.out.println();
}
```

Future courses will cover data structures called Collections (e.g. sets, maps, hash tables); you can iterate through these collections using the same for-each loop. In the next section we touch on one of the map classes.

Introduction to the Map class

Java has an inheritance hierarchy of collection classes that ranges from lists, to sets to maps. A map is a form of an associative array, which essentially allows you to implement a small database. It lets you associate a VALUE with a KEY in an efficient manner when you have many keys. For example, we might have a map from STUDENT ID to STUDENT RECORD. Our map could store all students and efficiently access a student record based on the student ID. This is a one-way association, it requires brute

force iteration if you start with the values and try to find the keys that map to it. Here we give an example of the HashMap class. It uses a structure called a hash table to efficiently map from a key to a value. Here is a demo:

```
import java.util.HashMap;
import java.util.Scanner;
public class HashMapDemo
{
	public static void main(String[] args)
	{
		// First create a hashmap with an initial size of 10 and
		// the default load factor
		HashMap<Integer,String> employees = new
			HashMap<>(10);

		// Add several employees objects to the map using
		// their id as the key
		employees.put(10, "Joe");
		employees.put(49, "Andy");
		employees.put(91, "Greg");
		employees.put(70, "Kiki");
		employees.put(99, "Antoinette");
		System.out.print("Added Joe, Andy, Greg, Kiki, ");
		System.out.println("and Antoinette to the map.");

		// Output everything in the map
		System.out.println("The map contains:");
		for (Integer key : employees.keySet())
			System.out.println(key + " : " + employees.get(key));
		System.out.println();

		// Ask the user to type a name.  If found in the map,
		// print it out.
		Scanner keyboard = new Scanner(System.in);
		int id;
		do
		{
			System.out.print
				("\nEnter an id to look up in the map. ");
			System.out.println("Enter -1 to quit.");
			id = keyboard.nextInt();
			if (employees.containsKey(id))
			{
				String e = employees.get(id);
				System.out.println("ID found: " +
				 e.toString());
			}
			else if (id != -1)
				System.out.println("ID not found.");
		} while (id != -1);
	}
}
```

Here are the key methods:

put(key, value)

```
get(key)              ; returns value
containsKey(key)
containsValue(value)  ; slow
keySet()              ; collection of keys
values()              ; collection of values
isEmpty()
remove(key)
```

If you intend to use your own class as the parameterized type K in a HashMap<K,V> then your class must override the following methods:

```
public int hashCode();
public boolean equals(Object obj);
```

These methods are required for indexing and checking for uniqueness of the key. We already discussed equals; the hashCode method is discussed in the Data Structures and Algorithms course (essentially it should ideally return a unique integer for each object).

Short Introduction to Generics

The ArrayList and HashMap classes used something new. It took as input a data type, in our case a MyIntClass definition, and used that to create the list. This type of class is called a **generic class** or a **parameterized class**. We can define our own generic classes if we wish.

We won't go into much detail here, but will just give a short example. Let's say we would like to make a class that can hold a Set of data. We can define a Set class that uses an ArrayList to store its data internally. In our add method we ensure there are no duplicate entries in the arraylist since most sets typically have no duplicates:

```
import java.util.ArrayList;

class MyGenericSet<T>         // Add <T> here to define a generic class
{
      private ArrayList<T> data;

      public MyGenericSet()
      {
            data = new ArrayList<T>();
      }

      public void add(T item)       // Use T for the generic data type
      {
            if (!data.contains(item))
                  data.add(item);
      }

      public boolean contains(T item)
      {
            return (data.contains(item));
      }

      public void remove(T item)
      {
            data.remove(item);
      }
}
```

Here is a main method to test it, using sets of integers and strings:

```
      public static void main(String[] args)
      {
            MyGenericSet<String> set1 = new MyGenericSet<String>();
            set1.add("java");
            set1.add("c++");
            set1.add("php");
            System.out.println(set1.contains("java"));
            System.out.println(set1.contains("english"));

            MyGenericSet<Integer> set2 = new MyGenericSet<Integer>();
            set2.add(3);
            set2.add(55);
            System.out.println(set2.contains(20));
            System.out.println(set2.contains(55));
      }
```