AKA. THE "NEW SCHOOL" APPROACH

STUDENT OUTLINE

Lesson 21 – Array Advances

INTRODUCTION:

As stated before, it is very common for a program to manipulate data that is kept in a list. You have already seen how this is done using arrays and ArrayList, but here we'll examine how the Java language itself uses inheritance to some advantages. A further discussion of typecasting for ArrayList will follow and address the use of *Generics* and *autoboxing*. Finally we'll discuss a new type of loop known as the for-each loop.

The key topics for this lesson are:

A. `List` interface
B. Generics and autoboxing
C. `for-each` loops

VOCABULARY:

interface
`List`
generics
raw type
autoboxing
`for-each` loop

DISCUSSION:

A. `List` interface

1. Look back to the API for the Java `ArrayList` class:

https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Notice this class implements many interfaces, but focus on `List` interface here:

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

Recall, an interface is a way to ensure that a group of classes MUST implement some methods. Since methods are actions, think of an interface as a protocol. Each class need not implement the methods in the same way (although they may), but the methods MUST exist in any class implementing the interface. In this case, `get()`, `set()`, and `add()` are all methods which any class implementing `List` MUST have a definition for. Like ArrayList, you must `import java.util.*;`

2. The thing to realize is that `ArrayList` uses an array internally to represent the list in question. But an array is not the only data structure we could use to store the data we have!! Any Java classes using another data structure to store the data as a list (maybe a stack or a vector) should also implement this interface. Then, no matter the internal representation, we can access that data in the same way we would if it were an `ArrayList`!! We can interact with the object in ways we're already familiar with.

3. This idea assists programmers to employ good decomposition and encapsulation in a project. To demonstrate, let's say you and your friend Arman are working on a project. You both decide that a list of integer data is required. Arman is writing the class which stores the data and provides some other methods needed. You could write some code that looks like this:

```
public Class Decider
{
    int value;
    Integer hold;
    List given;
    ArmanObj stuff = new ArmanObj();

    given = stuff.returnList();
    hold = (Integer)given.get(0);
    value = hold.intValue();

    …
}
```

Arman MAY have used an `ArrayList`, but he may NOT have!! As long as the return type of the `returnList` method implements the `List` interface it doesn't matter!!! We've left the implementation of the data storage completely up to Arman but designed this code in a way which we can still use it!

4. It is a powerful tool to use an interface as the type of an identifier, as above. It allows the identifier to reference any object created from a class which implements that interface. This allows the exact implementation of an idea to be left to elsewhere in the code and fully uses the ideas of inheritance and abstraction.

B. Generics and autoboxing

1. As you've undoubtedly noticed, typecasting when using an `ArrayList` is cumbersome at best. Early adopters of Java noticed this quickly as well and the language has evolved to make this work easier! The *Generic* concept allows us to create data structures that "know" what data types they hold.

2. Here is the way we've dealt with `ArrayList` thus far.

THE "OLD SCHOOL" APPROACH

```
List list = new ArrayList();
list.add(new String("ugh"));
String s = (String) list.get(0);
```

This approach is technically known as instantiating the `ArrayList` with a **raw type**. The data type stored is not known at the time of compilation and is treated as an `Object`. Therefore, we MUST typecast.

REQUIRED

3. Here's the same idea using generics:

NO TYPECASTING NEEDED

```
List<String> list = new ArrayList<String>();
list.add(new String("Wow!!"));
String s = list.get(0);    // no cast required
```

As you can see, when we "get" an element from this `ArrayList` we don't need to typecast it! Our identifier has been "told" what type to expect when working with the `ArrayList` and the code to handle that is inserted by the compiler at compilation for us automatically. Magic!!

4. The angle brackets(< >) hold the specific data type to be stored in the `ArrayList` above

PREFERRED VERSION, MORE EXPLICIT

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

So we are telling the compiler that our list will hold `String` elements. We could instead write this:

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

where inside the angle brackets are empty (and is known as the diamond). While technically correct in syntax, it's a better idea as a learning programmer to be explicit and use the first line given. Be sure to include the data type at BOTH sets of angle brackets.

5. Another concept that works along with generics is *autoboxing*. It essentially does away with the need to create wrapper objects when working with generic-ed (sic ??) classes. With autoboxing, we don't need to create an object when adding to our ArrayList.

```
List<String> list = new ArrayList<String>();
list.add("Yeeessss!!"); // no new required
String s = list.get(0);
```

As long as the compiler can infer the correct data type, it will add in the appropriate wrapper class codes at compilation. This autoboxing concept also applies to the use of any primitive data types.

AUTOBOXING

```
List<Integer> mylist = new ArrayList<Integer>();
for(int i = 1; i <= 42; i++)
   mylist.add(i);
int s = mylist.get(0);
```

In the above fragment the last line is technically autounboxing.

6. In the above examples we are simply using generics in pre-made classes. It's very powerful! Even further, we can incorporate this concept into classes of our own making. The idea of designing with generics is NOT part of the AP Subset and will be left for a later part of this course. But, with a little thought it should be clear how powerful an idea this is!!

AKA for-every

C. for-each loop

1. Lists and arrays are often associated with for loops. Many times when processing data in a list a full traversal of the list is required to process each element identically. Assume the `myList ArrayList` from number 5 above is being passed to the following method:

OUR KNOWN APPROACH

```
public int sumEven(List<Integer> listy)
{
    int sum = 0;
    for (int j = 0; j < listy.size(); j++ )
    {
          int i = listy.get(j);      //autounboxing
          if (i % 2 == 0)
               sum += i;
    }
    return sum;
}
```

2. But, there is another type of loop! One we can use when we want to visit each and every element in a list!! It's called the *for-each* loop.

```
public int sumEven(List<Integer> listy)
{
 int sum = 0;
 for (Integer i : listy)
     if (i % 2 == 0)    //autounboxing here too, cool!!!
          sum += i;
  return sum;
}
```

3. Inspecting the for-each loop in 2 above:

```
 for (Integer i : listy)
```

This line does the following:

1. It declares an `Integer i`
2. The loop then visits the current location in `listy` and gives `i` a reference to that object.
3. The body of the loop executes.
4. The current location in `listy` is advanced to the next location.
5. Repetition back to step 2 until all elements in listy have been visited.

4. Read the colon (`:`) above as the word "in". So think "for each `Integer i` in `listy`" when you read code like this.

5. This loop may also be used with static arrays as well!!!

```
public int sum(int[] a)
{
    int result = 0;
    for (int i : a)
       result += i;
    return sum;
}
```

SUMMARY/ REVIEW:

The `List` concept in original Java left much to be desired. Happily, the language evolved to deal with the unwieldy practice of typecasting through the use of generics and autoboxing. Another type of loop was introduced in later versions of Java to easily traverse arrays when "bulk" processing of the elements therein is required.