Generic Types

The Java Collections collect data, but what kind of data? Python doesn't care about types, so it will let you have a list where one element is an integer, the next is an object of class Person, the next is a boolean, and so forth. Java does care about types. Generally all of the objects in a collection have to have the same type. But what type?

One option would be to have separate implementations for collections with different types of data -- one implementation for lists of integers, one for lists of strings, and so forth. This seems wasteful and error-prone.

Another option is make only one list structure and make its base type be Object (the root of the class hierarchy in Java; all classes are descended from Object). We could cast all data into Objects when we put them into the list and cast them back into their real types when we take them out.

This would be both ugly and inefficient, and would negate many of the advantages of having type checking in Java.

Java's solution to this is to allow classes to parameterize types. For example, in Lab 2 you will implement a class called MyArrayList. Here is the start of this class declaration:

```
public class MyArrayList<E> {
        E [ ] data;
        int size;
        public MyArrayList( ) {
                size = 0;
                data = new E[2];
                ....
```

A specific list might have type

MyArrayList<String>

We will make a new array list of Strings with

MyArrayList<String> L = new MyArrayList<String>();

Note that the constructor we call is

new MyArrayList<String>( )

though in the class declaration the contructor is defined as

public MyArrayList( )

Look again at the class declaration:

```
public class MyArrayList<E> {
        E [ ] data;
        int size;
        public MyArrayList( ) {
                size = 0;
                data = new E[2];
                ....
```

E is used as a type throughout this class declaration. Of course, each instance of E refers to the same type.

We could also have classes that use several type parameters:

```
public class Pair<A, B> {
        A first;
        B second;
        public A getFirst() {
                return first;
        }
        ....
```

The actual types put in place of the type parameters need to be *reference types* -- classes or arrays. Primitive types, such as int, boolean, and float are not allowed. Fortunately, Java provides *wrapper* classes for each of the primitive types. For example, Integer is a Java class that holds a single int value. Java even automatically wraps and unwraps primitive types.

For example, suppose you want to make an ArrayList of ints. The declaration is

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

We could then call the add method for this list to put a value into L with

```
L.add(23);
```

Java automatically wraps 23 into an Integer to fit into this list, as though you had written L.add(new Integer(23));

Similarly, you can say

```
int x = L.get(0);
```

even though L.get( ) technically returns an Integer, not an int.