Generic Types

We have been using the <E> notation, called **generic types**, in our code and cheat sheet without explaining much about it. In a generic type, the <E> is a place-holder at compile-time for the actual object which supplied at run-time. Thus, if the cheat sheet shows the data structure ArrayList<E>, you know that you must specify the <E> in your program with ArrayList<String> or ArrayList<Widget> or something similar. You even wrote TJArrayList<E> that you used to store Widgets and then again to store Strings.

Before Java introduced generic types, all items put into a Collection were reduced to type Object. When you accessed the item, you had to remember to cast the Object explicitly. For example,

List myList = new ArrayList();

list.add(″TJHSST″);

list.add(″Colonials″);

String str0 = (String) myList.get(0); //cast

String str1 = (String) myList.get(1); //cast

(If you recall, ListNode and TreeNode also store Objects, which means you had to cast when you accessed their items. In other words, ListNode and TreeNode are not generic.)

If your codes uses generic types, the compiler "knows" what the objects in the ArrayList are, as specified inside the < > notation. For example,

List<String> myList = new ArrayList<String>();

list.add(″TJHSST″);

list.add(″Colonials″);

String str0 = myList.get(0);

String str1 = myList.get(1);

The entire point of generics is that these compiler-supplied casts are guaranteed to be *type-safe*. If you use generics, and the code compiles, then your codes is guaranteed not to throw a ClassCastException at run-time.

(turn over . . .)

Generic Methods

Java (beginning in Java 8) supports **generic methods** that allows the method to work with any given object (or type). You don’t need to cast in your code, nor do you need to write a different version of the method to work with each different object. A generic method specifies the **type parameter** <T> in the header, placed before the return type. The type parameter <T> defines dependencies among the types of the arguments to the method and/or its return type. The compiler guarantees that the dependencies among these types are satisfied, and therefore are *type-safe*.

In today’s lab you will see this **generic method** header:

public static <T> void output(Collection<T> array)

The first <T> says there are no dependencies, that any given type T stored in a Collection will work with this method. On the other hand, this generic header

public static <T extends Comparable<T>> void sort(List<T> array)

limits the kind of T that is allowed in List<T>. It says that, if you want to sort using this method, the objects to be sorted must be part of a hierarchy that implements the Comparable<T> interface. That dependency makes sense, because, as you know, in order to sort objects, the object T must possess a compareTo method.

Writing generic methods correctly is a complicated business, and we will not write any generic methods this year. In today's lab you are given generic type headers, but you must write the bodies of the methods.

If you are interested, read some more at

<https://docs.oracle.com/javase/tutorial/extra/generics/methods.html>

**Assignment**: The methods in this lab are **generic** methods. The headers are given to you. You are to write generic methods that sort any given type of object. Use the Selection Sort algorithm. You will be storing the objects in an ArrayList<T>. Your generic methods will first sort Widgets and then Strings. The data is in two files, widgets.txt and strings.txt. The shell is called SortingGenerically.java