Generics, and the List Interface

When you see an object type declared with another type surrounded by a pair of less than and greater than symbols (referred to as a diamond hereafter) like: `List<String>`

It means that you are looking at a JAVA type that uses GENERICS. When reading the type, you would say the type is a "List of String."

What does this mean? Generics allow a class to be written in a general way so it can work effectively with any type a programmer specifies. The type is placed inside the `< >` angle brackets (more commonly identified as a less than and greater than symbol respectively).

Note that the "of" must be an object; you cannot put a primitive JAVA type like `int`, `double`, or `boolean` in the diamond.

When it is necessary to store a JAVA primitive in a class that uses generics, you must use a "wrapper" class instead. A wrapper class is the object equivalent to the primitive type with the same or similar name.

Primitive Data Type	Wrapper Class
`int`	`Integer`
`double`	`Double`
`boolean`	`Boolean`

Note: There are many more primitive data types and parallel object types in JAVA but only these three are covered by the AP JAVA subset.

The primary class you must be familiar with that uses generics IMPLEMENTS the `List` INTERFACE to provide a much more flexible version of an array.

Recall that an interface is a list of methods that must exist in any class that implements it.

When specifying a List type, you must include the type that makes up the list, for example: `List<String>` is a list of String(s). When looking at the methods that a List has, the type that makes up the list is unknown before it is declared, thus it is generic, and so a capital "`E`" is used wherever the generic type would appear (look at the add, and set methods below).

The List interface contains the following methods that are tested in the AP JAVA subset:

- `int size()` – returns the number of elements in the list (equivalent to `.length` for arrays).
- `boolean isEmpty()` – returns `true` if the list has no elements.
- `boolean contains(Object element)` – returns `true` if the list contains the element.
- `boolean add(E element)` – returns `true` if the `element` is added to the end of the list.
- `void add(int index, E element)` – inserts the `element` at `index`.
- `E get(int index)` returns the `element` at `index`.
- `E set(int index, E element)` puts `element` at `index` and returns the object that was replaced.
- `E remove(int index)` removes the object at `index` and returns that object.

Note: There are **many** more methods in the `List` interface, but only these are tested!

The most confusing aspect of the `List` interface is that it cannot be instantiated even though it can be used as a type. Just remember, an interface doesn't include **any** code so it is **not** a concrete class and therefore it cannot be instantiated!

Therefore, to use a `List`, what we actually need is a concrete class that implements the `List` interface: the `ArrayList` class does just that!