CSCI 1933 Lab 7

Generics

Lab Rules

You may work individually or with *one* partner in lab. We suggest that you have a TA evaluate your progress on each milestone before you move onto the next. The labs are designed to be completable by the end of lab. If you are unable to complete all of the milestones by the end of lab, you have **until the last office hours on Monday** to get those checked off by **any** of the course TAs. We suggest you get your milestones checked off as soon as you complete them siwithnce Monday office hours tend to become crowded. If you worked with a partner, both of you must be present when getting checked off to receive credit. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Moodle for this lab.

Attendance

Per the syllabus, students with more than 3 unexcused lab absences over the course of the semester will automatically fail the course. The TAs will take attendance during lab; it is expected that students will be actively working on the lab material. *Your physical presence in lab is not sufficient to be marked as present for the purposes of attendance.*

What are Generics? Why do we want them?

Generics is a fancy term for the definition and use of Java's *generic types and methods*. Generic types and methods differ from what you're used to in that they take a *type* as a parameter. Consider the following:

Java before Generics (Java 4 and earlier)

Java didn't always have Generics. Say we wanted a data structure to hold pairs of some type (say `int`s), we could write the following class:

```
public class Pair {
    private int left;
    private int right;

    public Pair(int a, int b) {
        left = a;
        right = b;
    }

    public void setLeft(int left) {...}
    public void setRight(int right) {...}
    public int getLeft() {...}
    public int getRight() {...}
}
```

But what if we wanted to store `Pair`s of *anything*? If we didn't have Generics, a simple solution would be to use `Object`s as follows:

```
public class Pair {
    private Object left;
    private Object right;

    public Pair(Object a, Object b) {
        left = a;
        right = b;
    }

    public void setLeft(Object left) {...}
    public void setRight(Object right) {...}
    public Object getLeft() {...}
    public Object getRight() {...}
}
```

Great! Now we can make pairs of anything:

```
Pair p1 = new Pair(1, 2);
Pair p2 = new Pair(new Dog("Alice"), new Dog("Bob"));
Pair p3 = new Pair(3.14, 2.718);

Dog alice2 = (Dog) p2.getLeft(); // OK, this works, but it's annoying

p2.setLeft(new Cat("Charlie")); // wait what

Dog bob = (Dog) p2.getLeft() // this compiles, but at runtime I get an exception??!?!one?eleven!?
```

Okay, so maybe we can make pairs of whatever we want, but obviously there are some issues.

Enter Generics (circa 2004, Java 5)

So what exactly is a Generic then? Generics are used to fix the problem we just witnessed above. Using Generics, we can rewrite the `Pair` class so that it takes in a *parameter* when instantiated that tells it the *type* of the Objects we are storing.

Let's consider the `Pair` class once more, this time written so that it takes in a *generic type*.

```
public class Pair<T> {
    private T left;
    private T right;

    public Pair(T a, T b) {
        left = a;
        right = b;
    }

    public void setLeft(T left) {...}
    public void setRight(T right) {...}
    public T getLeft() {...}
    public T getRight() {...}
}
```

Notice that the signature for the class has this extra `<T>` and the code has all these `T`s in it. What's up with that? The `T`s are the *parameter* for the type of the `left` and `right` data and is specified when a `Pair` is instantiated. So, we can use the class as follows:

```
Pair<Integer> p1 = new Pair<Integer>(1, 2);
Pair<Dog> p2 = new Pair<Dog>(new Dog("Alice"), new Dog("Bob"));
Pair<Double> p3 = new Pair<Double>(3.14, 2.718);
```

```
Dog alice = p2.getLeft(); // works just fine, as we'd expect. Hooray!

p2.setLeft(new Cat("Charlie")); // this doesn't compile, just like we expect it to. Yippie!

Dog bob = (Dog) p2.getLeft() // this doesn't compile either. Woo!
```

Using a parameterized type such as `Pair<Dog>`, instead of `Pair`, enables the compiler to perform more type checks and requires fewer dynamics casts. This way errors are detected earlier, in the sense that they are reported at compile-time by means of a compiler error message rather than at runtime by means of an exception.

In addition to Generic classes, Java also has Generic methods. For example:

```
public static <T> void printArray(T[] a) {
    for(int i = 0; i < a.length; i++) {
      System.out.print(a[i] + " ");
    }
    System.out.println();
}
```

can be called on an array of *any* type and will print its contents.

1 Generics Practice

To get accustomed to writing classes with Generics, download the `AList.java` file from Moodle and modify it to use Generics rather than Objects.

***Important*:** To create an array that uses your generic type you will have to type cast an object array like so: `T[] list = (T[])new Object[10];`

Milestone 1: Write a main method to test your generic AList class. Your main method should create ALists for a least two different types of objects.

2 Adding and Removing

Using your Generic `AList` class, write the following methods:

- `public boolean add(int newPosition, T element)` – this will insert an `element` at index `newPosition` in the `list`.
- `public T remove(int givenPosition)` – this will remove and return the element at index `givenPosition`.

Milestone 2: Test the new functionality of the AList class in your main method. When should add return false? What should be returned if you try to remove something from an empty list?

3 More Generics Practice

- `public boolean contains(T element)` – this should return true if the `list` contains `element` and false otherwise.
- `public void add(T[] list)` – this should concatenate `list` to the AList.
- `public String toString()` – this should return the contents of the `list` as a string.

Given the list containing these elements,

these	could	be	strings	null	null

toString() should return the string,

```
[these, could, be, strings]
```

Milestone 3: Again, test the new functionality of the AList class in your main method. Can we assume that any object stored in our list will have a toString() and an equals()?