CSCI 1933 Lab 7

Generics and Nodes

Rules

You may work individually or with *one* other partner on this lab. All steps should be completed primarily in the lab time; however, we require that at least the starred steps (★) to be completed and checked off by a TA during lab time. Starred steps cannot be checked off during office hours. You have **until the last office hours on Monday** to have non-starred steps checked by ANY TA, but we suggest you have them checked before then to avoid crowded office hours on Monday. There is nothing to submit via Moodle for this lab. This policy applies to all labs.

IMPORTANT: You may now use an IDE to do all your programming for the remainder of the course. The IDE we will be using is IntelliJ IDEA and is installed on the CSELabs machines. If you are not familiar with using IntelliJ, please read the short primer on Moodle here. It briefly goes over initial setup and basic operations.

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 left) {...}
    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 alice1 = p2.getLeft(); // won't compile since getLeft() returns an Object
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 left) {...}
    public Object getLeft() {...}
    public Object 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 too. 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 `Node.java` file from Moodle and modify it to use Generics rather than Objects.

2 Adding to a List ★

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

- `public static <T> void add(Node<T> list, T element)` – this will add a new node to the end `list` with the data being `element`. You may assume that `list` is a *headed* list.
- `public static <T> String toString(Node<T> list)` – this will return a string containing all the elements in the list. Again, you may assume that `list` is a *headed* list.

  For example, if the list was

  $$\texttt{head} \rightarrow \texttt{"abc"} \rightarrow \texttt{"xyz"} \rightarrow \texttt{"lmnop"} \rightarrow \texttt{null}$$

  `toString` should return the String `"[abc, xyz, lmnop]"`.

3 More Node Practice ★

- `public static Node<Integer> xify(int[] x)` – this should return a list with `x[i]` copies of `x[i]`.

  If `x` is `[4, 2, 3]`, you should return the *headed* list:

  $$\texttt{head} \rightarrow \underbrace{4 \rightarrow 4 \rightarrow 4 \rightarrow 4}_{\text{4 total}} \rightarrow \underbrace{2 \rightarrow 2}_{\text{2 total}} \rightarrow \underbrace{3 \rightarrow 3 \rightarrow 3}_{\text{3 total}} \rightarrow \texttt{null}$$

  Use your `toString` to verify its correctness.

- `public static int[] xify(Node<Integer> x)` – this does the same thing as the previous method, but instead of taking in an array and returning a list, it takes in a *headed* list and *non-destructively* returns an array, i.e., does not modify `x` in any way within the method.

  If `x` is `head` $\rightarrow$ `4` $\rightarrow$ `2` $\rightarrow$ `3` $\rightarrow$ `null` you should return the array

  `[4, 4, 4, 4, 2, 2, 3, 3, 3]`

  Use your `toString` to verify that you did not modify `x` in your method.

4 More More Node Practice

- `public static <T> Node<T> reverse(Node<T> list)` – this should *non-destructively* reverse *list* and return a new *headed* list that is the reverse of `list`. You may assume that `list` is a *headed* list.
- `public static <T> Node<T> shuffle(Node<T> list)` – this should *non-destructively* shuffle *list* and return a new *headed* list that is a shuffling of `list`. You may assume that `list` is a *headed* list.

For both of the above methods, use your `toString` to verify their correctness, including that they do not modify the original input list.