Object Oriented Programming – II

Week-7

PART- I:

Types and generics:

The generic type is used by the compiler so it only affects the current type. The compiler makes sure that the types that you specify match.

new LLNode<String>("Hi", null); ← legal! "Hi" is type String and that matches the specified generic.

new LLNode<String>(new JFrame(), null); ← ILLEGAL! JFrame() is not a String. The generic was specified to be String, and so the element's type must match.

LLNode<String> node = new LLNode<Object>("Hi", null); ← ILLEGAL! The type of variable node is LLNode<String> so only LLNode's that have the generic specified as String can be assigned to the variable.

LLNode<Object> node = new LLNode<String>("Hi", null); ← ILLEGAL! The type of variable node is LLNode<Object> so only LLNode's that have the generic specified as Object can be assigned to the variable. It does not matter that String is narrower than Object!

SIDE NOTE: Wrapper Classes: For each primitive type (including void), Java provides a wrapper class that allows you to store the primitive inside an object. This allows us to use primitives where the code is expecting Objects. For the most part, the wrapper class is the same name as the primitive, but with a capital letter:

```
Double d = new Double(4.3);
```

except for Integer and Character. Starting with Java 5, Java will automatically convert between the wrapper class and the primitive, when needed and when it is clear what type is needed.

The == operator is not a primitive only operator, so the intValue() method is not called. As a result, the addresses of the Integer objects are compared, and since these are two different objects stored at two different locations, the == evaluates to false.

Static methods and generics:

First we wrote a method to print the contents of a linked list. However, if we make the method static, what should we use for the generic?

```
public static void printList(LinkedList<.....> list) {
        LLNode<....> nodeptr = list.getFront();
        while (nodeptr != null) {
                ????? e = nodeptr.getElement();
                System.out.print(e.toString() + " ");
                nodeptr = nodeptr.getNext();
        }
        System.out.println();
}
```

Solution 1: use a generic T. However, we cannot use the generic of the containing class (if it has one). The generic is specified by the creating of an instance to the class, and there is no instance in a static method. Instead, we can declare a generic in a method by placing the generic declaration before the return type.

```
    public static <T> void printList(LinkedList<T> list) {
        LLNode<T> nodeptr = list.getFront();
        while (nodeptr != null) {
          T e = nodeptr.getElement();
          System.out.print(e.toString() + " ");
          nodeptr = nodeptr.getNext();
         }
         System.out.println();
    }
```

The compiler will set T to be whatever generic is used in the current type for list. It will then verify that list.getFront returns an LLNode with the same generic specification.

Solution 2: use a wildcard. Java provides a wildcard ? that means "Don't Care" for the generic. We can use the wildcard if we really do not care. That means we don't do anything where we require the generic to be anything other than Object.

```
    public static void printList(LinkedList<?> list) {
      LLNode<?> nodeptr = list.getFront();
      while (nodeptr != null) {
        Object e = nodeptr.getElement();
        System.out.print(e.toString() + " ");
        nodeptr = nodeptr.getNext();
      }
      System.out.println();
    }
```

<u>PART- II: Loops and Abstract Data Types</u>

We can try to think of everything that a program that uses our LinkedList will want to do, but that is impossible.

Instead, other programs will need to write their own loops to run through the linked list.

This creates a problem: to run through the linked list requires having a node pointer, but if our linked list is to be a abstract data type, we do not want to require outside classes to have to deal with the linked list implementation details.

Java provides a pair of interfaces that let us provide a means for other code to loop through our linked list while still hiding the implementation details. These interfaces are Iterable and Iterator.

Iterable: indicates we can loop through the data stored in an instance

Iterator: the instance that performs the routines needed to do the loop

The Iterator interface

The Iterator interface is used to generalize the idea of a loop.

Basically, something that is type Iterator (implements Iterator) can be used as a loop. The Iterator interface has two non-default instance methods (that means we have to override these):

boolean hasNext() → returns true if there are more elements in the list

T next() → returns the next element in the list, and "iterates" so the next time we call next(), we get the next element of the list

(Notice from the API page that the Iterator interface takes a generic, so when we implement it, we need to specify the generic. Here we will specify the generic of Iterator by creating a generic in our class that implements the Iterator.)

Assuming these two methods have been properly overridden, we can now write a loop using just the Iterator interface.

```
Iterator<?> iterator =  ?????                 // pretend we have some way to get an iterator

        while (iterator.hasNext()) {

                System.out.println(iterator.next());

}
```

Note that the iterator lets us write loops for an abstract data type.

If we create an Iterator for our LinkedList class, the programmer writing the loop only needs to know how Iterators work and not any details about the LinkedList nodes.

Recall how we write a loop for a LinkedList:

```
        LLNode<?> nodeptr = list.getFront();      // assuming getFront is public!
        while (nodeptr != null) {
                System.out.println(nodeptr.getElement());    // print each element
                nodeptr = nodeptr.getNext();
         }
```

To write the Iterator class, we need to go back to our loop using LLNodes. The key parts of the loop are:

1) initalization: We have to set up the loop. Here the code is "LLNode<T> nodeptr = getFirstNode()"

2) condition: We need to indicate when loop should continue and when it should stop: "nodeptr != null"

3) retrieval: We need to get the next element from the list "nodeptr.getElement()"

4) increment: We need to move to the next element in the list "nodeptr = nodeptr.getNext()"

Now, we need to create a class for our linked list that implements the Iterator interface.

What should the hasNext do? The condition of the loop (nodeptr != null).

What should the next do? Both the retrieveal (return nodeptr.getElement()) and the increment (nodeptr = nodeptr.getNext())

That leaves the initialization step (nodeptr = getFirstNode()). Where should that happen? The constructor!

```
public class LinkedListIterator<T> implements Iterator<T> {
    private LLNode<T> nodeptr;

    public LinkedListIterator(LLNode<T> firstNode) {
      nodeptr = firstNode;
    }

    @Override
    public boolean hasNext() {
      return nodeptr != null;
    }

    @Override
    public T next() {
      T element = nodeptr.getElement();
      nodeptr = nodeptr.getNext();
      return element;
    }
  }
}
```

Notice the use of the generic. Iterator takes a generic (we can see that by looking at the API page for Iterator.)

So, we declare a generic in the LinkedListIterator header (the LinkedListIterator<T>), and now that we have a generic T declared, we use it when we implement Iterator (Iterator<T>). That forces the value returned by the next method to have the type of what we are iterating over. (The type that is stored in the linked list.)

(You may notice that we violated the OO-rules of using getter/setter methods. I chose not to do that so that the connection between the Iterator methods and the linked list loop is easy to see. We probably should add in getter/setter methods. Otherwise, creating a class that extends LinkedListIterator will be more challenging to get correct.)

However, the API for Iterator says that we need to throw a NoSuchElementException if next() is called when there are no more elements in the linked list.

We did not have time to add that in lecture, but it is easy to add an if statement to see if the nodeptr is null, and if it is throw the exception.

Now that we created the a class that implements Iterator, how do we connect that class to the LinkedList class? By using the Iterable interface.

The Iterable Interface

Iterable is an interface of the API.

The Iterable type represents an object that contains data that we can loop (or iterate) over.

By having the LinkedList class implement Iterable, we are indicating that we can iterate over the elements of the Linked List.

Every class that is an abstract data type and can store multiple elements should implement the Iterable interface.

Notice (from the API page) that the Iterable interface takes a generic. We will use the same generic that is stored in the LinkedList.

The Iterable interface has 1 (non-default) method:

```
Iterator<T> iterator()
```

This method returns the iterator for the abstract data type. Again, we are going to make sure that the generic used by iterator matches the generic stored in the linked list. And how do we do that? By having the Iterable interface use exactly the same generic.

```
public class LinkedList<T> implements Iterable<T> {
```

Now we have to override the iterator method inherited from Iterable:

```
@Override
public Iterator<T> iterator() {
// we need to return an appropriate object that implements Iterator
}
```

What should the iterator method return? An instance of the LinkedListIterator that we created above!

```
@Override
```

```
public Iterator<T> iterator() {
  return new LinkedListIterator<T>(getFirstNode());
}
```

Note that when we override a method, the name and parameter signature must be identical.

The return type, if non-primitive, is allowed to be narrower than the overridden method's return type. So we could also write:

```
@Override
public LinkedListIterator<T> iterator() {
  return new LinkedListIterator<T>(getFront());
}
```

Using the Iterable/Iterator interfaces:

Now, we can write a loop outside of the LinkedList class. (We could not before because the getFirstNode() method is protected.)

```
Ex:
  LinkedList<String> list = new LinkedList<String>();
  list.addToFront("Cleveland");
  list.addToFront("Cincinnati");
  list.addToFront("Columbus");


  Iterator<String> it = list.iterator();
  while (it.hasNext())
  System.out.print(it.next() + " ");

System.out.println();
```

Foreach loops:

Foreach loops are a Java shortcut for Iterable classes and for arrays.

The form of a foreach loop is:

```
for (T i : Iterable<T>)
```

and it reads as "foreach type T in iterable"

For example, if list is a LinkedList<Integer>, we could have:

```
for (Integer i : list)
```

which reads as "foreach Integer in list"

Here is an example:

```
LinkedList<Integer> list = new LinkedList<Integer>();
```

```
list.addToFront(1);
list.addToFront(2);
list.addToFront(3);
for (Integer x : list) {
        System.out.print(x + " ");
}
```

Note that the foreach loop is just a shortcut. Java takes the foreach loop on Iterable and coverts it to use the iterator:

```
        for (Double element : list)
```

is automatically converted by the compiler to:

```
        for (Iterator<Double> it = list.iterator(); it.hasNext(); ) {

          Double element = it.next();
```

This was not covered in lecture, but the foreach loop also works with arrays eventhough array are not Iterable type.

```
        double[] a = {1, 2, 3, 4, 5};
        for (double x : a) {
          System.out.println(x * x + " ");
        }
```

The foreach loop on array is also a shortcut that is automatically converted by the compiler to a normal for loop

```
        for (String s : a)
```

is the same as

```
        for (int index = 0; index < a.length; index++) {

          String s = a[index];
```

PART- III:

Restricting Generic Types and Wildcards:

Suppose we have the following method:

```
public static void displayFrames(LinkedList<JFrame> frameList) {
        for (JFrame frame : frameList)
                frame.setVisible(true);
}
```

This will only work on LinkedLists that have the generic type set to JFrame. What if we want to restrict the generic?

```
LinkedList<GeometricFrame> list2 = new LinkedList<GeometricFrame>();

list2.addToFront(new GeometricFrame());
```

list2.addToFront(new GeometricFrame());

However, if we call displayFrames(list2) we get a compile error because you can't convert LinkedList<GeometricFrame> to LinkedList<JFrame>.

Our solution is to restrict the generic type of LinkedList to be anything that is JFrame or narrower.

E extends JFrame : When you declare generic type E, you restrict E to be JFrame or narrower

? extends JFrame : When you use the "don't care" wildcard, you say the generic type must be JFrame or narrower

? super JFrame : When you use the "don't care" wildcard, you say the generic type must be JFrame or wider

You can also extend generics:

```
E extends T
? extends T
? super T
```

Here are the two ways to write displayFrames. First, declaring a generic restricted to JFrame or narrower:

```
public static <E extends JFrame> void displayFrames(LinkedList<E> frameList) {
  for (JFrame frame : frameList)
    frame.setVisible(true);
}
```

Or using the "Don't care" wild card:

```
public static void displayFrames(LinkedList<? extends JFrame> frameList) {
  for (JFrame frame : frameList)
    frame.setVisible(true);
}
```

In both cases we declare frame to be type JFrame. We know whatever is stored in frameList is JFrame or narrower.

PART- IV: The Comparable Interface

- one of the two most used interfaces in Java (along with Iterable). Indicates that instances of the type can be ordered

1. The Comparable interface takes a generic type that indicates the type this object will be compared to.
2. The Comparable interface requires a compareTo method that takes a parameter of the generic type.

It should return < 0 if this object comes before the parameter in the default ordering of the type.
It should return > 0 if this object comes after the parameter in the default ordering of the type.
It should return = 0 if the two objects are equivalent in the default ordering of the type.

We made the Employee class Comparable.
We decided that the default ordering would be to order employee's by their employee number
So, we made Employee implement Comparable:

```
public class Employee implements Comparable<Employee> {
```

Note that we specified the generic of Employee to state that we must compare Employee's to other Employee's.(The easy way to see what the generic should be is to look at the API and see where it is used. The API for Comparable<T> shows the method is - int compareTo(T e) - so we see that the generic needs to be the type that we want to input to the compareTo method.)

Now, we override the compareTo method. The class decided that the default comparison of employee's should be by employee number. Remember that we want the method to return negative if this object comes before the object in variable employee.

```
public int compareTo(Employee employee) {
   return this.getNumber() - employee.getNumber());
}
```

Using Comparable objects.
Where we would like to write "if (e1 < e2)" we instead write

```
if (e1.compareTo(e2) < 0)
```

Note that the < operator is the same, we just moved its location. This is the reason the return value for the compareTo is specified the way it is.

Using Comparable and Restricting the Generic Type:
The class example was to create a method in LinkedList that inserts elements in order into a linked list. To be able to order the elements, we need to be able to restrict the type stored to something that can be ordered. That is, something that is Comparable.

We are going to make the method static. Remember that a static method **does not inherit** the generic of the containing class so we have to declare it.
We made the method static so that we could give the method its own generic and restrict that generic to be Comparable.

If we want to keep the method non-static, we have to use the unrestricted generic T of the LinkedList class, and we would need to use typecasts and instanceof to check that the instance is a Comparable type and do the typecast so we can change the current type can call the compareTo method.

How do we insert in order? There are three cases: the element being added goes first, goes middle or goes last. It turns out that we can combine middle and last.

For going first, either the element is smaller than the first thing in the list or the list is empty. Otherwise, we have to loop through the list to find where to put the element. We do the normal linked list loop, but we have to be careful to stop at the right spot.

FIRST TRY:

```
public void insertInOrder(T element) {
   ...
}
```

We already have a problem. We want to call the compareTo method on elements but that will require the type of element to be Comparable.
Right now, T can be any type. We don't want to restrict the T on LinkedList because we want the LinkedList to still be able to store all possible types.

Solution 1: Use instanceof and typecasts everywhere. If we are going to do that, we lose the power of the generic typecasting.

Solution 2: Create a class that extends LinkedList and in the extending class we restrict the generic to be Comparable.

Solution 3: Make the insertInOrder method static so it no longer has the generic T. Now, we declare a new generic for the method and restrict that generic to be Comparable.

We decided to do solution 3. We will call the new generic S to distinguish it from T (but we could use T if we wanted to because T does not exist in static methods).

```
public static <S> void insertInOrder(S element, LinkedList<S> list) {
    if (list.isEmpty() || element.compareTo(list.getFirstNode().getElement()) <= 0)
      list.addToFront(element);
    else {
      // to be added later
  }
}
```

If we compile this, we get an error because S may not contain the compareTo method. We remembered to create a new generic for the method but we forgot to restrict it! We need to restrict S to be only things that implement the Comparable interface. We don't use the word "implements" because S is a type and not a class. The only time we use "implements" is when a class implements an interface. All other situations use "extends".

SECOND TRY: (The only change is in the generic declaration)

```
public static <S extends Comparable<S>> void insertInOrder(S element, LinkedList<S> list) {
    if (list.isEmpty() || element.compareTo(list.getFirstNode().getElement()) <= 0)
            list.addToFront(element);
    else {
          // to be added later
    }
}
```

This compiles and works great on a linked list of Employee, but it does not compile if we try to use it on a linked list of HourlyEmployee:

```
        LinkedList<HourlyEmployee> list = new LinkedList<HourlyEmployee>()
        list.insertInOrder(new HourlyEmployee("Harold"))    <= compile error!
```

To see why we have the error, look at the class types involved.

```
        public class Employee implements Comparable<Employee>
        public class HourlyEmployee extends Employee
```

- Remember that HourlyEmployee "is-a" Employee, and Employee is both an Object and a Comparable<Employee>. So HourlyEmployee is both an Object and a Comparable<Employee>

There is the problem!
insertInOrder states that the generic type is limited to <S extends Comparable<S>> (note that the S used in Comparable must be exactly the same as the type of the generic) and thus it works on a linked list storing "Employee implements Comparable<Employee>" but not on "HourlyEmployee implements Comparable<Employee>".

The solution is to change the restriction on Comparable to allow the Comparable's generic to be above the generic type used in the method.

FINAL SOLUTION: (The only change is again in the generic declaration)

```
public static <S extends Comparable<? super S>> void insertInOrder(S element, LinkedList<S>
list)
{
        if (list.isEmpty() || element.compareTo(list.getFirstNode().getElement()) <= 0)
          list.addToFront(element);
        else {
         // to be added later
        }
}
```

Now, we are stating that we do not care what type S is, but it must be Comparable - either because it implements Comparable directly or it inherits the Comparable from a parent class. Either way, we don't care how it became Comparable, but we know it has a compareTo method.