Object Oriented Programming – II

Week-8

PART- I:

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- II: 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 **<u>does not inherit</u>** 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("Orhan"))    → 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.

<u>PART- III: Exception Handling</u>

Consider the following class:

```
public class Averager {

        // return the average of the numbers (represented as strings) in the array
        public static int average(String[] values) {
                int sum = 0;
                for (int i = 0; i < values.length; i++) {
                        sum += Integer.parseInt(values[i]);
                }
                return sum / args.length;
        }

         public static void main(String[] args) {
                int average = average(args);
                System.out.println("The integer average of the inputs is " + average);
         }
}
```

Since we have a main method, this program can be run stand alone, and it averages (using int average) the command line arguments

```
        java Averager 4 5 6
        > 5
```

What can go wrong?

1. If the user does not enter an int (for example 4.5 or apple). In this case, the program will generate a **NumberFormatException.**
2. If the user does not enter anything. Then args.length is a 0, and the program will generate an **ArithmeticException.**

How to deal with errors:

1. Print an error message. This solution should only be used in routines that are directly interacting with the user. Otherwise the error message will be ignored.
2. Return a special "error" value. This solution should only be used when either the error value makes sense or there is no possibility that the error value could be confused for legitimate output.
3. Use a separate channel to send an error indication.
   For this technique, Java uses exceptions.
4. Deal with the error.

We have already seen several types of exceptions: NumberFormatException, ArithmeticException, NullPointerException, IndexOutOfBoundsException

Each of these exceptions are called unchecked exceptions.
All unchecked exceptions are subclasses of either Error or RunTimeException.

With unchecked exceptions, the programmer does not have to explicitly state what to do if an exception occurs (is thrown).
The default is to stop the method and throw the exception on the the calling method.

The other types of exceptions (for example IOException), are checked exceptions. With checked exceptions, the programmer must specify what to do if the exception is thrown. It is a compiler error to use code that can generate a checked exception but not explicitly state how to deal with it.

Key point: exceptions are just objects and they otherwise behave exactly like all other objects in Java.

- Exceptions are not errors.
- Throwing an exception is just another way (besides the return statement) of getting data out of a method.

However, throwing an exception is like "break" in a loop. Not only does it stop the method execution, but the calling method might simply throw exceptions to the methods that call it. As a result, exceptions are harder to reason about logically than "normal" return, and overuse will make our code more difficult to manage and harder to determine correctness.

Exceptions should be used for handling infrequent situations, and thus are often used for errors.

How to Handle an Exception:
There are two things the programmer can do for an exception: handle the exception or throw the exception on to the calling method.

To throw the exception, we can do either or both of these:

1. Place "throws ExceptionType" in the header of the method. This must be done if throwing a checked exception. This notation in the header both informs the compiler that the specified exception may be thrown and sets the default behavior of the method to throw the exception should it occur.

2. Explicitly throw the exception with the throw statement.
```
throw e;
throw new ExceptionType();
```

To handle the exception, we use the try/catch statement.

```
try {
  - code that could throw an exception
}
catch (ExceptionType e) {
  - code that is executed if an exception of type ExceptionType occurs inside the try block
  - e is a variable that stores the exception object address, and it exists inside this block
}
finally {
  - code that is always executed upon exit of the try and catch blocks
}
```

There may be 0 or more catch blocks with a try and at most one finally block.
There must be at least one catch block or a finally block with the try statement.

Note that the catch statement is a variable declaration. The variable e will hold the address of the exception that was thrown in the try block code.

Example 1: Dealing with the integer divide by 0:

The divide by 0 occurs in the line: return sum / values.length;

We can place a try/catch block around it and catch ArithmeticExceptions

```
try {
  return sum / values.length;
}
catch (ArithmeticException e) {
  // do something?
}
```

In class, we decided to print an error message, and in this case, the error should be handled in main, and not in average.

Exceptions have the benefit that we can handle the exception where it makes the most sense, not necessarily where they occur. So, we do not place the try block around "return sum / values.length;" but instead place it around the code in main that calls average.

```
 public static void main(String[] args) {
   try {
     int average = average(args);
     System.out.println("The integer average of the inputs is " + average);
}
catch (ArithmeticException e) {
     System.out.println("You entered nothing to average.");
   }
 }
```

Example 2: Dealing with the NumberFormatException

If the input contains a String that does not represent a number, a NumberFormatException occurs. We catch the exception and try formatting the input as a double. (Note we could have just formatted it as a double to begin with, but then we would not have as much fun dealing with errors.)

Then, we need a second try/catch block inside the catch block in case the number is also not a double.

We can use more than one catch statement with a try:

```
try {
}
catch (ExceptionType1 e) {
}
catch (ExceptionType2 e) {
}
```

On an exception, Java will run through each type from the top to the bottom and stop at the first one that matches. It is important that if one of the exception types is the parent of the other, the child type must come first. Otherwise both parent and child types will match the first parent type declaration, and the catch block of the second child type declaration will not be executed.

Another important point is that this will catch multiple exceptions that occur in the try block. An exception that occurs inside the first catch block is not caught by the second catch. Anything caught must be inside a try, and so to catch the error that occured inside the catch, we needed to nest another try/catch inside the catch block.