Object Oriented Programming – II

Week-12

PART- I: Review

A review of Nested and Anonymous classes

Static nested classes:

- used the same as "normal" classes except they are located inside another class.
- these classes have access to all the static elements of the containing class.

Non-static nested classes:

- you must create an instance of the containing class before you can create an instance of the nested class.
- these classes have access to all the static and non-static elements of the containing class.
- there are two "this" values in the instance methods of the nested class. One for the nested class instance and one for the containing class instance.
- there is the different way of creating objects using new when outside of the containing class.

Anonymous classes:

- A local class that is created at the same step as the "new" operator.

Nested Classes and Generic Types

When we look at the API for LinkedList, we see no mention of an LLNode class, but there -must- be one to be a linked list! Why don't we see one?

They must have the LLNode as a private nested class. In class, we did the same thing in

List1: a linked list with a private static nested class for the nodes

List2: a linked list with a private non-static nested class for the nodes.

Generics and Static Nested Classes:

Because the nested class is static the nested class -does not- have access to the generic type of the containing class. (The generic type only applies to instances of the class.)

Thus, we need to define a generic for the nested class. Specifically, our nested class Node needs its own generic type declaration.

As a result, we specify the generic on our Node class as:

Node<T> node or for it's "full name" List1.Node<T> node

To create a member of the nested class Node from outside the List1 class, we can use exactly the same Java terminology we use to create other objects and to access static fields and methods:

```
List1.Node<String> node = new List1.Node<String>("Hi", null);
```

Note that the generic goes on the Node class. We do not need to specify the generic for List1 because we are not creating an instance of List1.

(It is not an error to specify the generic for List1, but Java will ignore that generic since no instance is being created.)

Generics and Non-static Nested Classes
With a non-static nested class, the nested class -does- have access to the generic of the containing class. Thus the nested class Node of List2 does not need its own generic.

Note that in List2, the Nodes no longer have generic specifications and our code is a little cleaner.

In List2, we left the iterator as a static nested class to demonstrate how we have to now specify the generic on Node.

In the List2 iterator, we need to make sure the List2's Node's generic matches the generic of the List2Iterator. We can't say Node<T> because Node does not take a generic. The generic goes with the containing class List2.

So, we give Node's full name List2.Node, and we place the generic on List2 (the generic is declared in List2's header, not Node's).

```
List2<T>.Node node
```

In general, non-static nested classes can be more useful because they have access to all the non-static fields and method of the containing class, but they can be a little trickier to use.

To create a member of the nested class Node from outside the List2 clas, we do something similar as with the non-static nested class, but now we must create an instance of the outer class first. Note the difference with the new operator and note that we now specify the generic with the outer class:

```
List2<String> list = new List2<String>();
List2<String>.Node node = list.new Node("Hi", null);
```

PART- II: Java 8 Shortcuts for Anonymous classes: The "lambda" expression and the "method reference" expression.

Java has created two shortcuts that can be used for creating anonymous classes in the special case that the class is implementing an interface that contains a single method stub.

a. The "lambda" shortcut
b. The "method reference" shortcut

The Java "lambda" shortcut is not a true lambda. (In programming languages, a lambda expression is an anonymous function that can be stored to a variable, input to another function, returned from a function, etc.)

While the "lambda" shortcut makes it -appear- that we are giving a function as input to a method, we are really giving an instance of an anonymous class as input to the method.

Likewise, the "method reference" is not truly a method reference. It is written to -appear- as if we are passing a method as input to another method, but that is not what is happening. Instead, the "method reference" is effectively another form of an anonymous class.

The "lambda" shortcut for anonymous classes

Example 1: A basic anonymous class with a method that takes one input and no return value: Recall the EventHandler interface. It has a single method stub handle(ActionEvent e). Here is code creating an anonymous class implementing the interface:

```
b.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent e) {
    System.out.println("I was clicked");
  }
});
```

Here is the "lambda" shortcut. Because there is only one method in EventHandler, there is no ambiguity about which method is being overridden. All we need to know is what the input parameter name is and what the body of the method is.

Java uses the " **->** " symbol to indicate that we are using the shortcut. How does Java know that this is an EventHandler anonymous class? Because that is the parameter type for setOnAction!

```
b.setOnAction((ActionEvent e) -> {
                        System.out.println("I was clicked");
                          }
                       );
```

We can shorten it a little more by noting that the parameter type is also not needed:

```
b.setOnAction((e) -> {
                System.out.println("I was clicked");
                 });
```

Even more, since there is only one line in the body, we can remove the { }, and since there is only one input parameter, we can remove the ( ) around it to get the very short:

```
b.setOnAction(e -> System.out.println("I was clicked"));
```

Example 2: A non-void method in the interface

We did an example with the Comparator interface. It has many methods but a single non-default method stub: compare
Here is an example:

```
public static Comparator<Employee> compareBySalary() {
  return new CompareBySalary();
```

```
    }

    private static class CompareBySalary implements Comparator<Employee> {
      public int compare(Employee e1, Employee e2) {
        return (int)((e1.getSalary() - e2.getSalary()) * 100);
      }
    }
```

We will use the lambda expression. Java knows that we are creating and returning a Comparator<Employee> because that is the return type of the method.

Comparator has a single (non-default) method stub, and so there is no confusion about which method we are overriding.

Therefore, we can drop the interface name, the method name, and the input type. All that we need is the name of the input parameters and the body of the method.

```
public static Comparator<Employee> compareBySalary() {
  return (e1, e2) -> {
                        return (int)((e1.getSalary() - e2.getSalary()) * 100);
                    };
}
```

As before we can drop the { } because it is one line, and since the only line is a return statement, we can drop the return. Java knows that the method has to return a value. It just needs to know what value.

```
public static Comparator<Employee> compareBySalary() {
  return (e1, e2) -> (int)((e1.getSalary() - e2.getSalary()) * 100);
}
```

The "method reference" shortcut for anonymous classes

Consider the following anonymous class:

```
    b.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        reset();
      }
});
```

where reset is a private method:

```
    private void reset() {
      -- code to reset the application --
    }
```

In this case, the method we are overriding is simply calling another method. On one hand, this leads to a simple "lambda" shortcut:

```
b.setOnAction(e -> {reset()};);
```

However, if we change the other method so that it's input parameters are identical to the anonymous class method:

```
b.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent e) {
    reset(e);
  }
});
```

where reset is a private method:

```
private void reset(ActionEvent e) {
  -- code to reset the application --
}
```

Then we can use the "method reference" shortcut:

```
b.addActionListener(this::reset);
```

Note the syntax. To the right of the **::** is the name of the method our anonymous class method will call. To the left of the :: is the object or class that we will call the method on.

So, you can think of " **this::reset** " as being the same as "this.reset"; however, the :: makes it clear we are using the shortcut.

Java will effectively replace the "this::reset" with an anonymous class and its only method will call this.reset with the same input as was passed into the anonymous class function.

PART- III: Designing a Program.

In this class, we designed a program that played the game Chutes and Ladders. There are two ways to design a complex program or piece of a program.

1) Top-down design
- We start with the entire idea for the program and break it into the separate tasks.
- We then break each task into sub-tasks.
- We continue breaking down the sub-tasks further until we get to a stage where we finally see how to write code for it.

Example: to write the Chutes and Ladders program we must:
1) Create the game board

2) Create the players
3) Create the die
4) Play the game

For 1: how to create the board?
- create the spaces
- create the chutes and ladders
- create the path the players will follow

We need to think of two "boards" in our program. The visual board displayed on the screen and the model of the board used inside our program.

These two might be completely different as the model should closely match the way the game is played but the visual is how we want the game to appear.

2) Bottom-up design
In this technique, we do not worry about the entire program at first. Instead, we start by figuring out what piece of the program we can code first.

For example, I may not know how to code the game, but I do know how to generate a die roll. So, I write code for that, and then I figüre out what to do next.

In this technique, we will often need to change or throw away what we write first (because we do not know how all the pieces will fit together), but it has the advantage that we can at least have something to write and start playing with as we try to grasp the full complexity of the desired program.

Please see the Chutes and Ladders sample program. We needed to make some initial decisions before coding:
1) How to represent a board?
2) How to represent a player?

In both cases, we should focus on the behavior we need, and ignore what it looks like. For example, while the board game is a grid of squares, the actual game is played linearly, by running through the squares from 1 to 100, occassionally jumping to a new square.

As a result, we will use a single dimensional array to represent the board. We will make the array size be 101 (so entry 56 corresponds to square 56).

This will make the code easier to read since we will not be constantly subtracting and adding 1 to match the array index with the square number, and it only costs us a trivial amount of additional memory.

Also, it means we start all players at 0, instead of -1. What type should be stored in each array entry?

For the players, all we need to record is which square the player is on. Since each square is an entry in an array, we can store the index for the player.

Thus, we will represent each player by an int, and so the set of players will be an array of int.

Finally for the board, all we care about is, if the player lands on a certain square, do they need to jump to a new square? So, we will represent the board as an array of int. The value of the board[x] will be the square you need to jump to if you land on square x.

For all non-chute and ladder squares, we will just store board[y] = y to indicate the player should stay at this spot.

For playing the game, we used a bottom-up design.
First we created a way to test for a winner.
Then we created a way to move a player.
Then, we created code to change players after a turn.
Finally, we put it together in a loop.

Because we did a bottom up design, things were not quite as nice as they could be.

For one, there are a lot of magic numbers in our code. This can be later corrected to improve the program.

For another, we ended up coding methods that we did not actually need to get the game to work.

Making Code Easy to Test

One thing that may look strange in the program is that we wrote each task of the game as a separate method and we passed the gameboard in as a parameter - even though the gameboard is a field!

We did this to make it easier to write JUnit tests. If we rather used the field, then we would have to write our test cases for the entire Chutes and Ladders gameboard.

By having the board as input, we could custom craft specific small boards to test certain features of the game. This will greatly decrease the length of time it takes to write our testing code, and it did not significantly increase the time it takes to write the game program.

A Useful Tool to Help with Testing

We ended the class by demonstrating the "Code Coverage" tool of DrJava.
The code coverage works with your JUnit tests and it lets you see how much of your code was actually tested by the unit tests.
For example, you can click on the method name in the code coverage window and lines of code that are not hit by any unit test show up as red.

As a warning: the code coverage tool is really good for making sure your tests hit all parts of an if and do not miss a method.

However, it cannot distinguish between a "test 1" and a "tests many" in a loop. You still have to use the loop testing framework to guide you and not rely on the code coverage tool only.

(Note: the 2019 version of DrJava has a bug in it's code coverage tool. It seems that this version of DrJava was compiled with an outdated package. The code coverage tool works with the 2016 version DrJava. If you are using Oracle's java, you can use the 2016 version of DrJava, but for the Amazon Corretto jdk, you need to use the 2019 version to compile Java. In this case, you can launch both versions of DrJava. Use the 2019 version for coding and compiling but the 2016 version for using the code coverage.)

Code coverage is a common tool. If you are using an IDE other than DrJava, you should be able to find it somewhere.