Object Oriented Programming – II

Week-11

PART- I: Review

Introduction to JavaFX

To create a JavaFX application, you need to extend the abstract class javafx.application.Application.

For now there are four important methods of the class you will use, but two you can ignore.

1- The **init()** method is used to initialize your GUI, but you should not build the graphical part of the GUI here. The init method acts like your constructor for the class, but because of how JavaFX works, you should put your initialization code here and not in the constructor. Generally, the default init we inherit is good enough.

2- The **start(Stage primaryStage)** method is an abstract method that your class must override. The method should contain all the code to build and display your GUI.

3- The **stop()** method contains code that you want to run when the user exits the GUI. Usually the default will be good enough for us.

4- The **launch(String... args)** method is a static method that runs the GUI. When you run launch, it runs the init() method first and then the start() method. There is one catch to the start method.

JavaFX uses theater terminology.

- The start method takes a "Stage" as input. A stage is basically a window.
- Inside the start method, we will create a "Scene" that we place on the "Stage". The "Scene" contains what our GUI displays.
- Once the "Stage" is ready, we "show" it. The "Scene" can contain any JavaFX gadget.

A couple issues with JavaFX:

1) JavaFX uses the threads of your computer. We will discuss that later, but what that means for us now is that when you run the JavaFX application, it takes over your interactions pane. You will no longer be able to enter anything in the pane. If you need to debug a JavaFX application, the best way is with the debugger.

2) JavaFX uses a "static initializer" to set up the GUI. A static initializer is like a constructor, but for classes instead of for instances. We will discuss this later, but for now what it means is that you can't create an instance of a JavaFX gadget (like a button) in the interactions pane unless you first launch the application.

3) Because JavaFX uses a "static initializer" to set up the GUI, once you run the application, you can't relaunch it unless you reset the interactions pane.

Event Driven Programming Paradigm and Java Interfaces
In the event driven paradigm, the program waits for an event to happen. For example, it waits until the user clicks a button, and then the program responds to the event.

For this paradigm, we need:
- the program must register with the operating system to receive events
- the operating system needs to know how to inform the program when desired event occurs.

In Java, the JVM handles most of the registering, but we still need to register our objects with the JVM to receive certain events.

In this class, we created a window with a button, and we registered to receive button events. We then had our program do something when a button occurred.

Registering for button clicks in JavaFX: We will register our program to be informed when the button we creat is clicked.

The method to register for button clicks is in the Button class, and the API for the method is:
setOnAction(EventHandler<ActionEvent> h)

What you see is that the method setOnAction takes a value of type EventHandler as input. If we lookup EventHander, we see: Interface EventHandler<T extends Event>

So, EventHandler is an interface that takes a generic type, where the generic type can be anything that extends the Event class. We also see that setOnAction for Button specifies that this generic type should be ActionEvent.

Finally, we see that the EventHandler interface has a single abstract method:void handle(T event) Since we know that the generic is going to be specified as ActionEvent, we are going to have to override the method: void handle(ActionEvent event) in our class that implemets EventHandler.

Nested Classes
A nested type is a type defined inside another type. A nested class is a class defined inside another class. This lecture will focus on nested classes, but everything holds for interfaces and abstract classes as well.

A class inside a class is a member of the containing class. Just like methods and fields, it can be public or private (or package or protected), and static or non-static.

The rules for accessing a nested class are essentially the same as for accessing a field. When a class extends a class with a nested class, the rules are the same as for fields.

- A static nested class has access to the static fields and methods of the containing class.

- A non-static nested class also has access to the instance fields and methods of the containing class.
- Nested classes go into the hierarchy the same as other classes.

The only real difference between a nested class and a "normal" class is where it is located.

PART- II: Anonymous Classes:

An anonymous class is one defined in the new expression that is creating the instance of the class.

```
new InterfaceToImplement() {
      code to override the various methods here
};
```

or

```
new ClassToExtend() {
      code to extend the class by overriding methods here
};
```

Note, that you should not create new methods when writing an anonymous class, but you should just override existing methods. To see why, think about the polymorphism rules. What is the current-type of an instance of the anonymous class?

In ButtonGUI, we created an anonymous class for another button's event handler

```
button3.setOnAction(new EventHandler<ActionEvent>)() {
      // create an instance of an anonymous class
      @Override
      // that implements the interface EventHandler
  public void setOnAction(ActionEvent e) {
      // Here we must override the abstract method
      .... necessary code here ....
  }
});
```

The value passed into the setOnAction method is the instance of a class that implements the EventHandler interface.
This class has no name, but because it is an EventHandler, we can use this instance anywhere that expects an EventHandler.
Anonymous classes are very useful when we only need to create a single instance of a class and we are only overriding methods.

In these cases, why should we create a separate public class file when we can just define the class where needed in the code? Because the anonymous class is created at a specific line of code, it has access to everything available at that line.

For example, if you create the anonymous class inside a method, it will have access to the local variables of the method.

However, this can create a problem because local variables are allocated from the stack while the instance of the Anonymous class is allocated from the heap. Thus, when the method ends, the anonymous class still exists but those local variables are gone.

So, if you want an anonymous class to use a local variable, that variable must be declared "final". Recall that a "final" variable is a variable that will not change values. Thus, Java can safely copy the value of the variable to another location that is not the stack.

There will be no chance that the variable value will change. If a final variable is not appropriate, then you need to change the local variable to a field so that it can be stored in the heap.

PART- III: 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);
```

A Java 8 Shortcut for When Using Generic Types

In Java 8, you can drop the generic type specification in (most) places where you create an object.

For example:

```
new LLNode<Double>(3.5, null)
```

can be written as:

```
new LLNode<>(3.5, null)
```

Java will notice that the 3.5 is type double, and thus the likely type of the generic is Double. Please note, this will cause a problem if you do want Double as the generic, and you have

```
new LLNode<>(3, null)
```

because the type of 3 is int, and so Java will assume the generic type is Integer and not Double. In this case, you should use the full

```
new LLNode<Double>(3, null)
```

Similarly, we can shorten:

```
LinkedList<String> list = new LinkedList<String>()
```

to

```
LinkedList<String> list = new LinkedList<>()
```

In this case, Java sees that we are assigning the new object to a variable with the generic specified as String. Since generics have to match exactly, there is only one possible value for the generic of the new, and that is String. Thus, we can drop the generic here.

Note: This shortcut only works with the generic used with new. You still have to provide the <>. This is incorrect:

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

-- it is still a mistake because by not including <> you are not specifying a generic (and thus not typechecking).