Object Oriented Programming – II

Week-10

PART- I: Generics and Arrays:

Generics and arrays do not mix. The only thing you can do that combines generic types and arrays is to declare an array to hold a generic. For example:

```
T[] array;
```

You cannot create a generic array (new T[10]) nor can you create an array of a parameterized type (new LinkedList<String>[10]). The reason is because arrays can be automatically widened to Object[]. For normal arrays such as:

```
Object[] array = new String[10];
```

Java adds code that verifies every assignment to an entry in an array to make sure that the type being assigned matches the "true type" of the array. That is why "array[0] = new JFrame()" will compile but will throw an exception when run.

On the other hand, the generic information is only used by the compiler, and that prevents a similar type check from being done on an array created with a generic or an array of a parameterized type.

What can you do with array declarations and generics? Well consider this method header:

```
public static <T> void insert(T element, T[] array) {
```

it makes sure that the element and the type stored in the array are the same non-primitive type.

Introduction to JavaFX

The first was Java AWT. It was released almost as soon as Java was. Java AWT is for computer desktop GUI's only.

The second was Java Swing. It was built on top of Java AWT and simplified a lot of the Java AWT routines. Just like Java AWT, Java Swing is for computer desktop GUI's only.

The third is JavaFX. JavaFX is completely separate from Java Swing and Java AWT. As a result, it uses its own API that is separate from the regular Java API.

It was designed to allow a GUI to be built that can work both on the desktop and on a web page. JavaFX comes standard with Java 8, but with newer versions of Java, JavaFX is separate from the Java JDK and must be downloaded if you want to use it.

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.

Notice that the launch method is static while the start method is not static. Usually this means that we have to create an instance of the Application class to run the start method.

However, JavaFX will do this for us automatically when the launch method is run.

How does Java know what true type instance to create? Java uses something called **"reflection"** to find the correct instance to create and then runs it's start method.

You can place launch anywhere inside the class, but the typical thing to do is to place it in the main method for the class.

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.

In lecture, we just placed a button on the scene.

Usually, we will place a "layout manager" on the scene. The layout mananger dictates how the different GUI gadgets are arranged on the screen. And we then place the GUI gadgets into the layout manager.

```
public class MyFirstGUI extends Application {

  public void start(Stage primaryStage) {
    Scene scene = new Scene(new Button("Click me"));
    primaryStage.setScene(scene);
    primaryStage.setTitle("My GUI");
    primaryStage.show();
  }

  public static void main(String[] args) {
    Application.launch(args);
  }
}
```

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.

PART- II: Event Driven Programming Paradigm and Java Interfaces

In the "normal" programming model, a program executes a series of instructions until completion.
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.

Here is the code to set up the GUI and then register for the button click event:

```
public void start(Stage primaryStage) {
    Button button = new Button("click me");    // create the button
    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(button);              // place the button in the center of the layout
    Scene scene = new Scene(borderPane);       // create the scene with this layout
    primaryStage.setScene(scene);              // add the scene to the window
    primaryStage.show();                       // display the window

    button.setOnAction(...);    // register that this object should receive the button clicks
}
```

Now, we have to implement the EventHandler interface, we decided to do a **non-static nested class:**

```
private class HandleClick implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
      --- do something when the click happens ---
    }
}
```

The handle method is what will be called when the operating system detects a click of the button. **How does the Java Virtual Machine know the method is in ButtonGUI?** Because ButtonGUI implements EventHandler. So here is our code again:

```
public class ButtonGUI extends Application implements EventHandler<ActionEvent> {
  public void start(Stage primaryStage) {
    Button button = new Button("click me");    // create the button
    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(button);              // place the button in the center of the layout
    Scene scene = new Scene(borderPane);       // create the scene with this layout
```

```
        primaryStage.setScene(scene);             // add the scene to the window
        primaryStage.show();                      // display the window

        // register that the "HandleClick" object should receive the button clicks
        button.setOnAction(new HandleClick());

    }
```

Next, we decided to create to buttons, and use the same EventHandler instance on both, but we wanted the action to be different.

There is a method in ActionEvent called getSource that gets the object that produced the event. The return type of getSource is Object, but since we are creating the GUI, we know that the only thing that can produce that event is the Button, so we can safely typecast it.

```
public class ButtonGUI extends Application implements EventHandler<ActionEvent> {
    private Button button1;
    private Button button2;

  public void start(Stage primaryStage) {
        button1 = new Button("click me");
        button2 = new Button("and me");

        BorderPane borderPane = new BorderPane();
        borderPane.setTop(button1);               // place the button at the top of the layout
        borderPane.setBottom(button2);            // place the button at the bottom of the layout
        Scene scene = new Scene(borderPane);      // create the scene with this layout
        primaryStage.setScene(scene);             // add the scene to the window
        primaryStage.show();                      // display the window

        HandleClick handler = new HandleClick();

        // register that the "HandleClick" object should receive the button clicks
        button1.setOnAction(handler);
        // register that the "HandleClick" object should also receive this button's clicks
        button2.setOnAction(handler);

  }

private class HandleClick implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        Button b = e.getSource();
        if (b == button1) {
            -- do something when button1 is clicked --
        }
```

```
            else {
               -- do something if a different button is clicked --
            }
         }
      }
```

A couple things to note:
1) Why did we have to make button1 and button2 fields?
2) We chose to not have getter/setter methods for button1 and button2.

This violates our O-O coding guidelines, but we are okay with that because we do not want other code modifying these buttons. If we do want to let another class extend this GUI and change how the buttons work, we will need to create getter/setter values.

3) Why does the nested class have access to the instance fields of the outer class? Because the nested class is not static. We can think of the nested class as belonging to a specific instance of the GUI and not the the class.

<u>**Java Interfaces and Types**</u>
How is JVM using the EventHandler object? Exactly like the interface example of the prior lecture!

The method setOnAction has its input as type EventHandler, and so inside setOnAction, this object will have current type EventHandler.

```
   setOnAction(this)   ← this is being typecast here to EventHandler
```

Inside setOnAction, the input is stored in a parameter variable: ActionEvent event.
When an action occurs, the JVM calls event.handle(some ActionEvent object).
This call is legal because the current type of event is EventHandler, and EventHandler has a handle method. Which method gets called, the true type's overridden version of EventHandler. In our case, that will be the version we wrote in ButtonGUI.

Static Initializers (Static Constructors)
Constructors are the methods used by new to initialize the object when it is created in the heap. Likewise, when a class is loaded into the heap, a "static constructor" also called a static initializer can be used to initialize the class structure (basically the static fields).

The static constructor is just a compound statement preceded by static:

```
  static {
     .. the code to run when the class is first loaded into the heap
  }
```

Static Constructors and JavaFX

JavaFX uses static initializers in a way that creates some issues with how we can test and call JavaFX methods. For example, if we create a button

```
    Button b = new Button();
```

JavaFX loads a bunch of classes when doing new Button, and these classes have static initializers that will throw an error if the JavaFX Applications's launch method was not run.

As a result, if you try to use new Button() in the interactions pane or in a method that is called outside of a JavaFX Application (such as a JUnit test), the code will throw an exception.

This makes JavaFX hard to use with the interactions pane. When you call Application.launch(), the code does not return control to the interactions pane. As a result, you can't do anything more in the interactions pane. But if you don't call Application.launch(), then calling new Button() in the interactions pane will throw an exception.

Later in the course, when we discuss "threads", I will show you how you can get around this so you can use the interacitons pane, but for now, the best way to debug a JavaFX application is to use a debugger.

PART- III: Nested Classes and Anonymous 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 nested class goes into the heap just as normal classes do. A nested class contains everything that normal classes do plus they have access to the elements of the containing class.
- 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.

Static Inner Classes

Example 1: A public static nested class

We first created a public static nested class for our event handler.

```
public static class RotateClick implements EventHandler<ActionEvent> {
  .... some code here ....
}
```

To create an instance of the nested class RotateClick from outside the Button4GUI class, we can use exactly the same Java terminology we use to create other objects and to access static fields and methods:

Ex: to access a static field, we use classname.field so to access a static nested class we use outer_classname.nested_classname

To create an instance, we use new.

```
Button4GUI.RotateClick c = new Button4GUI.RotateClick();
```

Generally, static nested classes are useful, but in this case, we wanted access to the text area of the ButtonGUI. That requires access to the object, but the static nested class belongs to the class and not to the object.

Non-static Inner Classes

Example 2: A non-static nested class

Next, we made SpinClick to be a non-static nested class that implements EventHandler.
By making the nested class non-static, the nested class has access to all of the non-static fields and methods of the containing class.

```
public class SpinClick implements EventHandler<ActionEvent> {
  .... necessary code here ....
}
```

As a non-static nested class, the class belongs to the object.
So, we can access the non-static fields and methods of the object.

```
public class SpinClick implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    button1.setRotate(button1.getRorate() + 10);
  }
}
```

Two issues with non-static nested classes.

1) The first (which I did not cover in lecture) is that there are two "this" values active. "this" refers to the object of the nested class: this.button1.setRotate(will give an error because button1 is not a field of the nested class, it is a field of the outer class.
2) The other this is "Button4GUI.this".
   Button4GUI.this.button4.setRotate(is correct.

Or, we can just use

```
    button1.setRotate(
```

and have Java automatically add the correct this.

We can also access the nested class from outside the containing class (since it is public).
The key thing to remember for a non-static nested class is that, just as with non-static fields, you have to create an instance of the containing class before you can create an instance of the nested class.

The syntax is a little weird when you are outside the containing class:

```
    Button4GUI app = new Button4GUI();
    Button4GUI.SpinClick l = app.new SpinClick();
```

For our final version, we made the nested class private. By making the nested class private, it cannot be (directly) accessed by code outside of the Button4GUI class. As a result, we are indicating that this event handler only belongs with the Button4GUI class.

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.