

## Object Oriented Programming – II

### Week-13

#### **PART- I: Review**

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- II:** Example GUI coding

In this section, we will practice doing event programming with JavaFX to create a simple drawing program. This is a good time to practice nested and anonymous classes.

Create a JavaFX application with a Canvas and do some drawing. You will need to reference the API in this lab. Be sure you have the JavaFX API opened up, and not just the standard Java API. Many of the JavaFX classes have similar names to classes in the standard API.

Create a new class called Painter that extends Application. Add a BorderPane to the Scene, add the Scene to the Stage, show the Stage and create a main method that calls Application.launch(.).

Once you have the JavaFX application set up, create a Canvas object and add it to the center of the BorderPane. I suggest making the Canvas a large size like 500x500 or bigger. (Our canvas is not going to be resizable in this lab, so you want something big enough to work with.)

Now we want to be able to draw on the canvas by clicking and dragging on it with a mouse. For this you need to add an EventHandler to your canvas that listens for MouseEvents. You want to call canvas.setOnMouseDragged(...) where canvas is the variable storing your Canvas object, and the input to the method is a EventHandler<MouseEvent> object. There are multiple ways you can do this:

1. Have the Painter class implement EventHandler<MouseEvent>.
2. Create a nested class that implements EventHandler<MouseEvent>.
3. Create an anonymous class that extends EventHandler<MouseEvent>.
4. Use the "lambda" shortcut that provides the body of what you want to occur in the handle method of EventHandler.
5. Write a method that takes a MouseEvent as input and use the "method reference" shortcut

The body of the handle method (or the body of the "lambda" shortcut, or the body of your method reference) should do the following:

1. Get the GraphicsContext for the Canvas. Canvas has a method getGraphicsContext2D that returns the GraphicsContext.
2. Set the color to draw by using the setFill method of the GraphicsContext. For now, set the color to Color.BLACK.
3. Draw a circle on the Canvas at the current location of the mouse. Use the fillOval method of the GraphicsContext context. Set the width and height parameters to 10 and get the x and y coordinates from the MouseEvent that is the parameter of the method.

Compile and test your program to see if you can draw!

### **Add Some Color**

Now let us add some color to the pen. JavaFX has a gadget called the ColorPicker that lets a user choose a color.

Create a new ColorPicker and use the constructor that sets the initial color to Color.BLACK. Then place the ColorPicker at the top of your BorderPane.

Now change the EventHandler you use in the setOnMouseDragged method to have the GraphicsContext use the color from the ColorPicker rather than always using Color.BLACK. Use the method getValue to get the current selected color from the ColorPicker.

### **Change the Pen Size**

Let us improve our drawing program by letting the user change the pen size.

We will use a Slider to select a size. Create a new Slider, and add it to the right side of the BorderPane. Use the constructor to set the minimum value to 1, the maximum to 101, and the default to 10. Next you want to set the orientation to vertical. Try to figure out how to do this using the JavaFX API.

Second, let's make the slider look nice by calling the following methods, replacing "slider" with whatever you chose to name your Slider:

```
slider.setShowTickMarks(true);
slider.setShowTickLabels(true);
slider.setMajorTickUnit(10);
slider.setBlockIncrement(1);
```

Now change your EventHandler used in setOnMouseDragged to use the current value of the slider, found by using the getValue method of Slider for the height and width of fillOval

### **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 figure 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, occasionally 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 its 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.