Lab Quiz 2 Template (Section B4)

TA Instructions

Before Lab:

- Copy this template into a new file and put it in the Lab Quiz 2 folder with your lab section in the name. Do not edit this template directly since other TAs will need to copy this
- Fill in all the personalizations for each question template **before your lab section**
    - Feel free to alter the questions beyond the templates as long as you're testing the same concepts. The template provides a minimal amount of variance between labs
- Have a plan to display the quiz to your students
    - Setup a github repo and have students view the code on their computers with close supervision to ensure they only access the quiz code
    - -or- Notify Jesse via Slack that you want your quiz printed

During Lab:

- Give them 1 hour to complete the Quiz. You don't have to start at the beginning of lab but you must end the quiz 1 hour after you start
- **Team Adjustments:** For the remaining time, address any team reformations that may be necessary due to resignations and non-participants. Use your discretion in rearranging teams and bring up any tough calls in the Slack channel for help deciding.

Grading:

- Start grading as the quiz ends. When you're done grading, add their scores into AutoLab and release the grades for your section. Their grades should be posted within 24 hours of taking the quiz. You can hand the quizzes back next week
- Run your code to be 100% sure of the answers

Student Instructions

- This is an exam environment
    - No talking
    - No material allowed except blank paper and writing utensils
    - Only ask TA's questions clarifying what a question is asking

Q1 Template

```
/*
Use the template below as a start and add your personalization to include the following

-Rename everything. The names are generic placeholders that should be replaced with your
example
-Choose a type for the state variable from {Double, Int, String, Long} (Leaving it as Double
is fine)
-The abstract class has at least 2 methods with at least one implemented and at least 1
abstract and a state variable (The template has a constructor, but this can be changed if
you'd like. You could even write a trait instead of an abstract class)
-Create at least 2 concrete classes that extend the abstract. At least one of these will
override the concrete method and at least one will keep the concrete method of the abstract
class
-All of the implemented methods should have a mix of returning simple operations using the
state variable and modifying the state variable

Ideally your example should make sense in the real world


Part 1 Questions (Inheritance) - Give the students a main method 5 println statements and ask
them what will be printed for each (5 points each). These should test their knowledge of
references, inheriting methods, overriding methods, and methods with side-effects

To include all of these you should
-Create instances of each class, store at least one of the reference in more than one
variable, create variable of all class types (include the abstract class)
-Call methods of the same name in each concrete classes such that an overridden method is
called and an inherited method is called (+ if the overridden method is called from a variable
of the abstract type)
-Call a method that changes the state variable of an object (side-effect), then call a method
that relies on that change but from a different variable storing the same reference


Part 2 code
Write two methods that take data structures of the abstract type as parameters such that
-One method returns a value that relies on the state variables of every object in the data
structure
-One method changes the state values of the objects in the data structure

Part 2 Questions (Polymorphism) - Give the students a main method with 2 println calls worth
10 and 15 points. Populate a data structure of the abstract type with
-At least 1 object of each concrete type
-At least one object that is in the data structure multiple times, but in a non-obvious way
(ex. Create two variables storing the same reference and add both variables to the DS)
-Call the method that relies on each state for the first println
```

```
-Call the state altering method at least once then call the other method again for the second
println

*/
abstract class Employee(var salary: Double) {

 def pay(): Double

 def jobReviewBonus(): Unit = {
   this.salary += 200.0
 }

}

class Teacher(salary: Double) extends Employee(salary) {

 override def pay(): Double = {
   this.salary += 100.0
 }

}

class Janitor(salary: Double) extends Employee(salary) {

 override def pay(): Double = {
   this.salary + 25.0
 }

 override def jobReviewBonus(): Unit = {
   this.field -= 15.0
 }

}

def accumulate(things: List[AbstractClass]): Double = {
 var total = 0.0
 for(thing <- things){
   total += thing.methodThatReliesOnStateVariables()
 }
 total
}

def changeState(things: List[AbstractClass]): Unit = {
 for(thing <- things){
   thing.methodWithSideEffect()
 }
}
```

```
Part 1) What is printed to the screen

       def main(args: Array[String]): Unit ={
               val teacher = new Teacher(20000)
               val janitor = new Janitor(5000)
```

Q1 Grading: Can be all or nothing, but you can use your judgement for some partial credit if it’s clear that they made a small mistake and understand polymorphism (ex. A small math error on the 15 point question, but understood everything including the double reference and all side-effects)

Q2 Template

```
/*
Use the template below as a start and add your personalization to include the following

-Rename methods and variables. Keep the class names as Model/View/Controller
-You can leave the view mostly as-is and only update the calls to your newly named Model and
Controller methods and change the text on the buttons. Feel free to make more changes if you
want more buttons, etc
-Choose a type for the state variable in the Model {Double, Int, String, Long} (Leaving it as
Int is fine)
-Write at least 2 API endpoints for the model, at least one of which takes a parameter. All
API methods should return Unit and have side-effects that will be displayed on the GUI
-Have the app respond to button presses and keyboard inputs
-For each user input have the controller call a model method
-Make sure pressing a key that is not explicitly covered has an effect on the app (eg. case _
=> should change the state of the model)

You can use a real-world example for this, but make sure they can’t guess the answers without
understanding the code. For example, if you give them a well-designed GUI with a great user
experience they might be able to figure out the answers just by knowing how the app should
behave and not tracing through the code

Q2 Questions
Give them 5 (10 points each) different sequences of inputs and ask what is displayed after
each sequence. Make it clear that the app is restarted before each sequence (We don’t want
compounding errors). For these sequences start with testing the features of your app in
isolation (Ex. only button presses, or inputs that all call the same model method) then for
the last 1 or 2 sequences combine your features
*/
```

```
class Model {

 var money: Int = 10

 def displayMoney(): Double = {
   this.money
 }

 def deposit(value: Int): Unit = {
   this.money += value
}

 def empty(): Unit = {
   this.money = 0
 }

}


class Controller(model: Model) {

 def b1Pressed(event: ActionEvent): Unit = model.deposit(10)

 def b2Pressed(event: ActionEvent): Unit = model.empty()

 def userAction(event: KeyEvent): Unit = {
   event.getCode.getName match {
     case "A" => model.deposit(10)
     case "B" => model.deposit(20)
     case "C" => model.empty()
     case "D" => model.deposit(30)
     case "E" => model.deposit(40)
     case "X" => model.empty()
     case _ => model.deposit(-5)
   }
 }

}

class QuizButton(display: String, action: EventHandler[ActionEvent]) extends Button {
 val size = 200
 minWidth = size
 minHeight = size
 onAction = action
 text = display
 style = "-fx-font: 30 ariel;"
}


object View extends JFXApp {
 val model: Model = new Model()
 val controller: Controller = new Controller(model)

 var textField: TextField = new TextField {
   editable = false
   style = "-fx-font: 26 ariel;"
   text.value = model.displayField().toString
 }

 stage = new PrimaryStage {
   title = "Quiz GUI"
   scene = new Scene() {
     content = List(
       new GridPane {
         add(textField, 0, 0, 2, 1)
         add(new QuizButton("Deposit $10", controller.b1Pressed), 0, 1)
         add(new QuizButton("Empty", controller.b2Pressed), 1, 1)
       }
     )
   }

   addEventFilter(KeyEvent.KEY_PRESSED, controller.userAction)

   // update the display after every event
   addEventFilter(Event.ANY, (event: Event) => textField.text.value =
model.displayField().toString)

 }

}
```

Q1 Grading: Can be all or nothing, but you can use your judgement for some partial credit if it’s clear that they made a small mistake and understand GUI’s + MVC