CS 2300 Lab 09 MVC GUI – Part 1 Week 08

Start Project.

Enums. Model Class. JPanels and JFrames

This project has two main goals:

1. Build a simple application with a simple GUI (Graphical User Interface).

2. Illustrate major Java and OOP concepts.

The project is organized according to the **MVC (Model – View – Controller)** paradigm. While this project is as simple as possible so that it can be completed in five or six weeks, much more useful applications can be created similarly. Because a lot of code is required for a GUI, we will let Intellij IDEA provide as much as possible. This project will be continued for several weeks.

0. **Preview**. There will be five classes that are "top level" in the sense that all other classes occur as **fields** in one of the top level classes. The classes are:

0.1. **Model**. This class holds all the "state data" that defines how the view is drawn. The model class "super top level", in that it has fields that are references to the other top level classes.

0.2. **View**. This class and its subclasses are responsible for drawing the view.

0.3. **Controller**. This class presents various controls that allow the data in the model to be changed.

0.4. **Title**. This is a very simple class that simply draws a large title across the top of the application.

0.5. **Frame**. This class is like a canvas that lives on a computer monitor. It has **controls** and a **view** of the application painted on it.

Develop the GUI: Step by Step.

In the early parts of this project, detailed instructions for each step will be provided. In the later parts, you will have more flexibility to do what you want.

0. Fire up Intellij IDEA. Probably a good idea to go to the "File" tab, and click "Close Project" and close any open projects.

Now click the "New Project" button and give the new project the name `MVC22F`, then "create".

1. As a preliminary step, we will create a helper class called an **`enum`**. It will be used later to determine what kind of picture the application will draw.

Enum Discussion. Enumerated types, or, "enums", are a good way to keep track of what option has been selected. In the old days, if you wanted to have a bunch of options, you just used an integer with values like 0, 1, 2, 3, …, for the cases. Then your code would have a bunch of things like this:

```
if (option == 0){

  // Do something;

}
```

This way of representing options has a lot of down sides. A first improvement is to collect all the `if` statements into one `switch` statement. Another improvement in languages like C and C++ are **enum**s, which are basically names to be used in place of the numbers (so you don't forget what is what). This notion has been greatly expanded in Java, where `enum`s are a special kind of `class`, and can do many cool things.

1.1. Create a special **enum** for this project. Right click on **scr** (below `MVC22F)`, and NEW and JAVA CLASS. Before you type in the name, click ENUM, and then the name **`mvc22f.Task`**. This will create a package named mvc22f, and the following special type of class:

```
public enum Task {
}
```

Add the following code inside the enum:

```
    INFORMATION,

    RECTANGLE,

    ELLIPSE,

    SIERPINSKI_GASKET
```

These are the enum cases. They are separated by commas. This is a very simple type of enum, but enough for this project.

2. The **`Model` class** is center of the entire project. It holds the **state** (all the information) of the application. It is also the communication system that allows the other parts of the project to communicate

2.0. Create a class called `Model`. **This and all other classes should be in the `mvc22f` package**. Do this by right clicking on the package mvc`22f` and selecting "Java Class". Change the code to look like this:

```
public final class Model {

}
```

The `final` specifier means the class cannot be extended by a subclass, which provides some extra protection. Good coding practice is to make everything final unless there is a reason not to.

The `Model` class is the central class for the entire project. All the data defining the current state of the project will live in this class! All other classes communicate by passing messages through the Model class. We will be adding one thing at a time to the model.

2.1 Put a c-tor in `Model`, like this

```
public Model(){
    System.out.println("Model c-tor: this = " + this);

    System.out.println("End Model c-tor");
}
```

The print statements are to help us understand what is happening when we launch the application.

2.2. We will also need a `main` method somewhere to launch the entire project. One option is to put this right in the `model` class.

Skip a couple lines after the c-tor. Generate a `main` method. Put the following code

```
System.out.println("Start MVC GUI Application main method");

SwingUtilities.invokeLater(() -> {

    Model model = new Model();

});

System.out.println("End MVC GUI Application main method");
```

inside the main method. The SwingUtilities will turn red. Fix this by clicking on it and hitting ALT + ENTER. This will "fix imports" by adding `import javax.swing.*;` at the top. You will probably have to do this often.

We will explain this in a minute, but first run the program (click green triangle). This should produce the following:

```
Start MVC GUI Application main method

End MVC GUI Application main method

Model c-tor: this = MVC22F.Model@79f931b4

End Model c-tor
```

(the number `79f931b4` will be different)

Discussion: Here is what is happening:

2.2.1. The action starts in `main` and writes out `Start MVC GUI Application main method`

2.2.2. Then `SwingUtilities` **launches another thread** to host the GUI application we are creating! (Doing this has a lot of benefits!)

2.2.3. However, the new thread does not start just yet. First the next line gets printed: End MVC GUI Application main method.

2.2.4. Then the original thread closes, and the new thread is launched. The only thing it does is instantiate an object of class `Model`. Then the `Model` c-tor gets called. All it does (so far) is print out two more lines:

Start Model c-tor

End Model c-tor

2.3. The code above uses the new concise **lambda** feature (added in Java 8). For comparison, here is the "old style" version of the same thing:

```
SwingUtilities.invokeLater(new Runnable() {

    @Override

        public void run(){

            Model model = new Model();

        }

});
```

This might be slightly easier to understand, but a lot longer. Feel free to consider this pure magic that nicely launches the GUI framework as a **separate thread**. Most GUI programs are multi-threaded to improve performance.

2.4. If you are **NOT** interested in how this all works, jump ahead to 3. If you are interested, Google on `Runnable` and `SwingUtilities`. Roughly speaking, here is what is happening. We are creating an **anonymous object** of the **`Runnable` interface**. An anonymous object means we do not give the object a name or a type, but just construct it where it is needed, specifying whatever code is needed. This maneuver is done all the time in GUI programming.

The **`Runnable` interface** requires that the object be provided with a method called **`run`**. In this case the **`run`** method only contains one line: `Model model = new Model()`. What the `SwingUtilities.invokeLater` does is launch a new thread that that will "run the GUI". Congratulations! You are now writing a **multithreaded** program! You can now click shift+F6 to run the program. It does not do anything, so it will be hard to know it is running! Next we will make it do something we can see.

3. Next we will make some classes that can show up on the monitor! This is easy to do with **inheritance**. The basic rectangle that can be placed on the monitor is called a `JPanel`. By making a class **extend** `JPanel`, the child class knows how to present itself on the monitor.

4. Create a `final` class called `Title` by right clicking on `mvc22f` and closing `new / Java Class`. **Be sure to create this and all other classes in the `mvc22f` package!** Intellij IDEA will write it like this:

```
public class Title{}
```

5. Make the `Title` class `final` and **extend** the `JPanel` class so that it drawn as a rectangle on the screen. To do this, change the above to this:

```
public final class Title extends JPanel{}
```

You should get some error messages and JPanel should turn red. Click on the red word. Intellij IDEA will "fix the import" if you enter ALT + ENTER. Do that. Notice the new line: `import javax.swing.*;` at the top. This will allow you to use stuff from `swing`, including `JPanel`.

5.1. Using the word `final` with a class means the class cannot have any subclasses. Making classes `final` avoids security issues and other problems. Whenever you get an error message about "Constructor Calls Overridable Method", making the class `final` will fix it. The only reason not to have a class `final` is if you want to extend it with subclasses, which we will rarely do in this project.

5.2. The class is empty to start. Click inside the class and hit ALT + INS to start "automatic code generation". Select CONSTRUCTOR. A pop up box will ask you to select one of four things. Pick the last one, which is the simplest, and click "OK". This generates an "empty –looking" method. This method actually does something: it calls the **`JPanel`** super constructor, which does some important stuff. We don't have to think about what it does, because our new class **inherits** what it needs from **`JPanel`**.

5.3. Add the lines

```
System.out.println("Title c-tor");
setBorder(BorderFactory.createLineBorder(Color.RED, 2));
```

to the c-tor. Fix imports. We will add more later.

5.4. Now go to the `Model` class. You can do this by using the "Project" panel on the left side of the IDE, and clicking on Model.

5.4.1. At the top (above the c-tor), add this line:

```
private final Title title = new Title();
```

5.4.2. Run the program by clicking the green triangle on the second row from the top. This is an easy way to run the project you are working on. The other way is to first go to the tab holding the main method, and then entering SHFT + F6 (which is more work).

6. Create a class called `Controller`. Intellij IDEA will create it like this:

```
public class Controller {}
```

As above, make this class `final` and extend the `JPanel` class.

This class needs to communicate with the model. This is accomplished by having a **reference to the Model class**. You can think of this as being similar to having a phone number for the Model class, so you can call and ask for stuff whenever you need to. To make this happen, add one field (at the top of the class):

```
private final Model model;
```

In this context, the keyword `final` means that once the value is set, it can never be changed. Because there is no reason why this should ever be changed, the `final` allows the compiler to help us out by guarding for an accidental change to the variable. It is a good program practice to always make variables `final` unless there is a reason not to.

You will get an error message about having no default c-tor. Here is how to fix that. Recall that hitting `alt+ins` brings up a context sensitive menu for creating simple methods for us. Do it like this: click `alt+ins` to bring up the menu, and select constructor (also called a c-tor).

Choose the bottom line. The reason for this choice is to choose options for the c-tor for the **superclass**, which is `JPanel`. When you have a class that extends another class, the first thing a c-tor does is automatically call the c-tor of the parent class. This goes all the way up to the class `Object`, which is at the top of the class hierarchy.

Now click OK. Now the IDE offers you the chance to select “model:Model”. Click OK. This will generate a c-tor contains the statement

```
this.model = model;
```

which stores the address of the `model` so other parts of the program can use it for communication.

Also add these lines at the end of the c-tor:

```
System.out.println("Controller c-tor: model = " + model);

setBorder(BorderFactory.createLineBorder(Color.BLUE, 2));
```

6.1. **Discussion.** This GUI project is organized by the **MVC** (Model – View – Controller) **paradigm**. Model is a class that plays two major roles.

The `Model` object acts as a “central switchboard” for communication between different parts of the GUI.

The `Model` object holds references to all the other classes in the project. Thus it can “talk to” any other class by using the reference to the other class. You can think of these references as the “phone numbers” for the other parts of the project. We will see how this is done below.

Any other class can “talk to” the `model` object if it has a **reference** to the object. This reference is the “phone number” for the `Model` object. Everyone who wants to talk to the `Model` need a copy of this number.

To make this work, the c-ror for the other objects have the line:

```
this.model = model;
```

which stores the reference to the model. This is analogous to when you enter someone’s phone number into your phone.

Conclusion. The first task to building this project is setting up the classes so they can communicate with each other.

6.2. Return to the `Model` class. Add this line at the top (right after the one for Title):

```
private final Controller controller;
```

This will cause an error for the following reason:

We cannot put `= new Controller(this)` right here, because this c-tor needs a **reference** of type `Model` as the `this` argument. But this reference cannot be accessed until the `Model` c-tor has run. This complication is dealt with like this: Put the following statement **inside** the `Model` c-tor:

```
controller = new Controller(this);
```

6.3. The argument this is a copy of the reference (or, address) of the `Model` object.

Click the green triangle, and notice that now there is something like:

```
Model c-tor: this = mvc22f.Model@46123953

Controller c-tor: model = mvc22f.Model@46123953
```

The hex number `46123953` is a reference to the `Model` object is stored in memory. It MIGHT BE the actual address where the object is stored, but you can’t be sure of that. It doesn’t matter because you never need to know where it is stored.

Inside the Model class, this reference is called `this`. Inside the other classes, it is called `model`. Note that the two hexadecimal numbers are the same – the actual place in the computer’s memory where the object of the Model class is stored.

Don’t worry, you will never have to use this address directly, you just write: `model`.

7. Create a class called `View`, and **do the exact same stuff** as you did for the `Controller` class. The only things to change are

7.1. Make the color `GREEN`

7.2. Change the text `Controller` to `View`.

Now add a couple lines to the `Model` c-tor, so it looks like this:

```
private final Title title = new Title();
private final Controller controller;
private final View view;

public Model(){
    System.out.println("Model c-tor: this = " + this);
    controller = new Controller(this);
    view = new View(this);
    System.out.println("End Model c-tor");
}
```

Run the program. The output should look something like this:

```
Start MVC GUI Application main method

End MVC GUI Application main method

Title c-tor

Model c-tor: this = mvc22f.Model@79f931b4

Controller c-tor: model = mvc22f.Model@79f931b4

View  c-tor: model = mvc22f.Model@79f931b4

End Model c-tor
```

8. A **Frame** is a class that can be drawn on the monitor, and contain an entire project. The last main class we need will be a `JFrame` rather than a `JPanel`.

8.1 Create a class called `Frame`, like this:

```
public final class Frame extends JFrame
```

As above, add the field:

```
private final Model model;
```

Position the cursor after this line.

Then enter ALT+INS, and select C-TOR. Select the "parent class" that has ZERO arguments (the simplest one is usually the best bet). Click OK. Click OK again. This will generate the following:

```
public Frame(Model model) throws HeadlessException {
    this.model = model;
}
```

Don't worry about `throws HeadlessException`. We might discuss this in class – otherwise regard it as some magic that needs to happen.

Add this statement to the c-tor:

```
System.out.println("Frame c-tor: model = " + model);
```

8.2. Return to the `Model` class. Add

```
private final Frame frame;
```

after the other three fields. Add

```
frame = new Frame(this);
```

in the c-tor (after the other instantiation statements).

Click the green triangle to be sure things are working.

9. Go to the Model class. Click right after the c-tor, hit ALT + INS, choose "getters". Choose all four fields listed (can do this with shift + down-arrow). Click "OK". This will generate "getters" for the title, controller, view, and frame fields.

The reason for this is so the model class can pass these references to other classes. This is how the model acts like a "central switchboard" for the entire project.

10. Next we will add some **state variables** to the model. These control the state of the view (i.e., how it looks).

10.1. Go to the `Model` class. There are already four fields defined at the top, above the c-tor. Skip a line and add five more:

```
private Task task = Task.ELLIPSE;

private int xSize = 66;

private int ySize = 44;

private Color color = Color.BLUE;

private boolean solid = true;
```

10.2. The current values (Task. *ELLIPSE*, `66, 44, Color.BLUE, true`) are **default values**. These are so GUI looks OK when it first starts up.

10.3. Notice that these fields are **not** `final`, because we are going to add **controls** to allow users to change them.

Thus we need both **getters** and **setters** for these variables. The role of getters is so that other parts of the GUI can obtain information on how to draw the image. The role of setters is so the we can add **controls** that will change the way the image is presented.

10.4. Go to the model class. After the c-tor, but before the `main` method. Hit ALT+INS to bring up the "Code Generator". Select "Getter and Setter". The code generator will offer to create getters and setters for the five new fields (`task, xSize, ySize, color, solid`). Here, "select" means click on the line, which turns it blue. Click OK.

10.5. Notice that ten new methods have been created: a getter and a setter for each of the five state variables. The code generator makes it easy to add common methods to classes. This greatly speeds up software development.

11. Now we will add some stuff to the `Frame` class so a GUI is produced.

11.1. In `Frame` c-tor, after the existing statements, add the following:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

setSize(1000,700);

setTitle("MVC 22 Fall");

setVisible(true);
```

Try to guess what each of these lines does. The names are pretty good, so you should be able to figure out the basic idea!

12. Run the program (click the green triangle).

Now a GUI window should appear. Pretty cool.

If you try to click on the green triangle again, you will get a warning.

Close the windows by clicking the X in the upper right corner.

13. Do a little experimenting with the last three lines you added. For example, eliminate one and see what happens when you run it. Or, use a different size or title. This is how you figure out how stuff works.

How to turn in Project. This is slightly complicated, so follow these instructions for submitting exactly:

1. Open file explorer
2. Find your project (likely C:\Users\<user>\IdeaProjects\MVC22F)
3. Right click on your project folder (this should contain src/mvc22f with all your .java files)
4. Select **Send to > Compressed (zipped) folder**
5. Submit MVC22F.zip to Blackboard