Section 1 - Responding to Classes

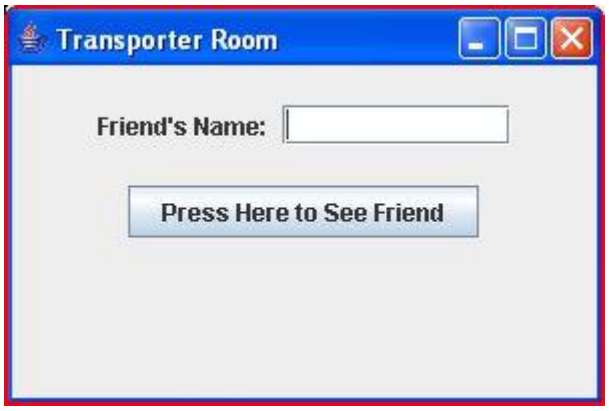
10B.1.1 Transporter, Resurrected

Just for review, and to get your projects filled with the right code, here is where we left our old friend.

Transporter GUI:

```
import javax.swing.*;
import java.awt.*;

public class Test
{
   public static void main(String[] args)
   {
      // establish main frame in which program will run
      JFrame frmMyWindow = new JFrame("Transporter Room");
      frmMyWindow.setSize(300, 200);
      frmMyWindow.setLocationRelativeTo(null);
      frmMyWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // set up layout which will control placement of buttons, etc.
      FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 5, 20);
      frmMyWindow.setLayout(layout);

      // 3 controls: a label, a text field and a button
      JLabel lblMyLabel = new JLabel("Friend's Name: ");
      JTextField txtMyTextArea = new JTextField(10);
      JButton btnMyButton = new JButton("Press Here to See Friend");

      // place your 3 controls into frame
      frmMyWindow.add(lblMyLabel);
      frmMyWindow.add(txtMyTextArea);
      frmMyWindow.add(btnMyButton);

      // show everything to the user
      frmMyWindow.setVisible(true);
   }
}
```

And this was the (unresponsive) GUI that resulted when we executed the program:



Time to make this GUI do something.

Section 2 - Transporter Retooled

10B.2.1 A Fresh Look at our Transporter

Before we add to our earlier transporter application let's re-examine it using OOP vocabulary. For example:

```
      JFrame frmMyWindow = new JFrame("Transporter Room");
      frmMyWindow.setSize(300, 200);
```

As we now know, the first of these two lines is a combination of the two other statements:

```
      JFrame frmMyWindow;
      frmMyWindow = new JFrame("Transporter Room");
```

We can now describe this in ***OOP*** terminology. **JFrame** is a ***class*** which Java defines for us (inside the **javax.swing** package). We are declaring a ***reference*** for this ***class*** named **frmMyWindow**. Then we ***instantiate*** an ***object*** using the **"new"** keyword. After these two statements are executed we have a real **JFrame** ***object*** that we refer to using **frmMyWindow**.

Also, when we instantiate this object, we know that a ***constructor*** gets called. From the form of our instantiation line, we can conclude that the ***constructor*** takes a ***parameter*** and we are passing in the argument **"Transporter Room"**. We also know from looking at the output of the program that this parameter tells the **JFrame** what text to write in the frame title.

Meanwhile, the second of the two original lines, **frmMyWindow.setSize(300,200),** is clearly a ***method call.*** Since we are ***dereferencing*** it through a **JFrame** ***object***, it must be an ***instance method*** (not a ***static class method***) and will cause something specific to happen to that particular ***object***. It's pretty clear from the name of the ***method*** that it is setting the size of the **frmMyWindow** object to be 300 pixels x 200 pixels.

This kind of analysis applies to all of the ***classes*** that we use in this program. We are ***declaring references, instantiating objects*** and ***modifying properties*** of the ***objects*** for each of the various classes, **JFrame, FlowLayout, JLabel, JTextField** and **JButton.** Of course, since we are using pre-defined Java classes, we don't see the definitions of those classes, but merely import them using the import statements and use them as we like.

This should bring us up-to-date on our understanding of the version of the program as it was when we last saw it. Now we make improvements.

10B.2.2 Breaking the Program into Two Classes

Our first modification will involve separating the **JFrame** data and methods from the main class. In other words, the way the "grown ups" write such programs is to create a new class which ***encapsulates*** all of the **JFrame** issues, and to separate the new class from the main class. We will name our new JFrame-type class **TranspoFrame**, and our main class **Foothill**.

Look at the new version:

```
import javax.swing.*;
import java.awt.*;

public class Foothill
{
   public static void main(String[] args)
   {
      // establish main frame in which program will run
      TranspoFrame myTranspoFrame
            = new TranspoFrame("Transporter Room");
      myTranspoFrame.setSize(300, 200);
      myTranspoFrame.setLocationRelativeTo(null);
      myTranspoFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // show everything to the user
      myTranspoFrame.setVisible(true);
   }
}

// TranspoFrame class is derived from JFrame class
class TranspoFrame  extends JFrame
{
   // member data (instance variables)
   JButton btnMyButton;
   JTextField txtMyTextArea;
   JLabel lblMyLabel;

   // TranspoFrame constructor
   public TranspoFrame(String title)
   {
      // pass the title up to the JFrame constructor
      super(title);

      // set up layout which will control placement of buttons, etc.
      FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 5, 20);
      setLayout(layout);

      // 3 controls: a label, a text field and a button
      lblMyLabel = new JLabel("Friend's Name: ");
      txtMyTextArea = new JTextField(10);
      btnMyButton = new JButton("Press Here to See Friend");
      add(lblMyLabel);
      add(txtMyTextArea);
      add(btnMyButton);
   }

} // end class TranspoFrame
```

Don't panic. This is straightforward.

I'll guide you through it.

- The first line of our new class **TranspoFrame** has some interesting syntax

```
class TranspoFrame extends JFrame
```

This means that our new class will ***inherit*** all of the built-in capabilities of **JFrame**. We say that we are ***deriving*** a new class **TranspoFrame** from an existing ***base class*** **JFrame**. Everything we put into the definition of our new class will now be ***added on*** to the **JFrame** data and methods. The main **Foothill** class is now very short. It only has a few lines because all of the hard work is now being done in our **TranspoFrame** class.

- Since **TranspoFrame** is derived from **JFrame**, **main()** can use **TranspoFrame** as it would **JFrame**. We declare a **TranspoFrame** object, instantiate it, and start calling methods like **setSize()** and **setVisible()** exactly as we would if we were using **a JFrame *object***. In fact, we <u>*are*</u> using a **JFrame *object*** when we use a **TranspoFrame *object*** by virtue of the fact that ***TranspoFrame*** is ***derived from JFrame***.
- The GUI controls (the button, label and text field) need to be defined somewhere. They are no longer local variables of the **main()** method. So where would this all be done? Since we now have a class devoted to our application window, the **TranspoFrame** class, it makes sense that these become ***instance variables*** of our new class.
- All the things that we used to do in the **main()** method of our **main** class still have to get done. But now we do these things inside the **TranspoFrame** class. But where? Do you remember what method always gets called before any other method in a class? The ***constructor***. So we instantiate the controls (buttons, labels, etc.) in the **TranspoFrame** constructor. Now, if we look back at the first line of **main()** we see that the instantiation of the **TranspFrame** object, **myTranspoFrame**, is the exact place where all this work gets done. Why there? Because that is where the constructor automatically gets called for us.
- There is one strange line that sticks out::

  ```
  super(title);
  ```

  Recall that in our old version, we passed the ***String literal*** "Transporter Room" to the **JFrame** ***constructor***. Since we are not using **JFrame** directly in our main, but instead are using the derived **TranspoFrame**, we will pass this literal to our **TranspoFrame** ***constructor***. But this constructor is defined by <u>*us*</u> and we don't know how to manually cause a String to be written in the **JFrame** title bar. Therefore we call the special method **super()** which in turn calls the actual **JFrame** ***constructor***. If we wish to pass any arguments to that ***constructor*** (as we certainly do here) we place those arguments into the **super()** call. This is a common technique when we ***derive*** a class and we want our ***constructor*** to call the constructor of the ***base class***. This is called ***function chaining*** or ***method chaining*** or ***constructor chaining.***

- Finally, we can see that some of the ***instance methods*** of the **JFrame** class are no longer ***dereferenced*** through any objects but are called in a vacuum. By this I mean that in the original version we called

-
  ```
  frmMyWindow.setLayout(layout);
  ```

  while in this version we just call

  ```
   setLayout(layout);
  ```

Since **setLayout()** is an ***instance method*** of **JFrame,** and **TranspoFrame** is derived from **JFrame**, we don't have to supply a dereferencing object if we call a **JFrame** method from within a **TranspoFrame** method. And that's what we are doing. We are calling the **setLayout()** ***method*** from inside the **TranspoFrame** ***constructor***. The rule that ***instance methods*** can call ***other instance methods*** or data of *the same class* without a dereferencing object, also applies to inherited class definitions. The methods in the derived classes can access the methods of the base class without dereference. This is a concept we explore more fully in the second course CS 1B.

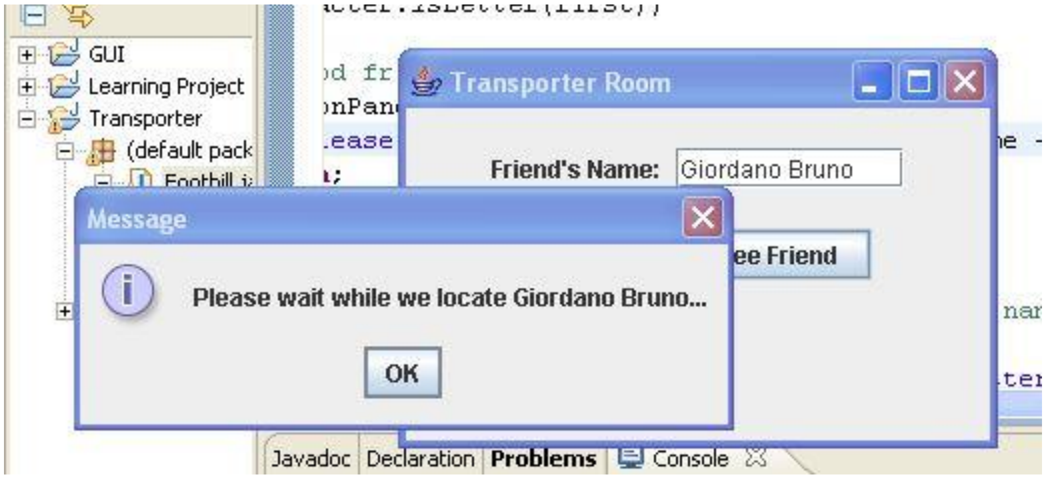
Well this is how it's done in the big leagues. Because **JFrame** is a typical class on which to base GUI applications, it is very common to ***derive*** our special flavor of dialog window from this from **JFrame** and do all of the initialization in our ***derived constructor***. Everything here, the call to **super()**, the placement of the controls in the derived class as ***instance variables***, and the **"un-dereferenced"** use of the **JFrame** ***methods***, are all part of the correct way to implement this GUI. You can almost memorize these things since they will usually be done exactly the same way in future projects.

Section 3 - Events and Listeners

10B.3.1 Handling GUI Events

Our immediate challenge is to get this stubborn program to respond when we push the **Push Here to See Friend** button. For this we will need to define a new ***class***. Luckily, we learned about creating ***classes*** last week, so this is going to be easy.

When we press the button now, we will get the following kind of response from the program:



Please wait while we locate Giordano Bruno...

First some vocabulary.

- **Controls -** The various GUI objects on our **JFrame** are called ***controls***. The **JLabel, JButton** and **JTextField** are three types of ***controls*** that we give the user as a means of interacting with the program. A control can be almost anything that appears on our **JFrame**.
- **Events** - When we do something to one of the controls, like push a **JButton** or type some text into a **JTextField**, we are generating an ***event***. Some ***events*** will cause our program to take actions, while others will be ignored We decide what to listen for and what to ignore. For example, we certainly want to listen for a button press event, but we may not care if the user types a single character into a text field. Still, typing a character, even without hitting the enter key, is an ***event***, and some programs will listen for that ***event***.
- **Listeners** - A class that we create which is intended to respond to an ***event*** is called a ***listener class***. We supply ***listener classes*** only for those events that we care about. Inside the ***listener classes*** we handle the events. ***Listeners*** refer to the ***classes*** that we create to handle ***events***.
- **Event Handlers** - The method that is called as a result of a particular event occurring is called an ***event handler***. All ***event handlers*** must be defined inside ***listener classes***. Therefore, an ***event handler*** is a method inside some ***listener class***. (Some programmers call these event handlers "callbacks" which is an antiquated carry-over from an old style of event programming.)

If you read ahead or look at other textbooks, you may find that some ***events*** seem to be ***handled*** using a strange syntax right in our main or derived **JFrame** class without creating a special ***listener class***. This is a special short-cut called ***anonymous inner class listeners***. Even though they appear to be handled without creating separate ***listener classes***, they really are separate classes. They are simply nested inside the main class for convenience.

The process of getting our program to react to user events is a simple one.

1. **Place the control** somewhere in our container frame, usually in the **JFrame**. (We have already been doing this with the **JFrame add()** method.
2. **Define the listener** class that will handle the event for the control. Sometimes we have a separate ***listener class*** for each ***control***, other times one ***listener*** services several controls.
   - These classes should normally be given a name and be nested within the main **JFrame** class (i.e., they should be ***inner classes***)
   - The classes must be declared on their top line as ***implementing*** a particular listener (e.g., **ItemListener**, **ActionListener**, **KeyListener**, etc). Which listener the class implements depends on which event we are trying to handle.
   - These classes must contain a special ***event handler method*** that has a pre-determined name. The name will depend on the type of event it is to handle.
3. **Register our listener** with the ***control*** that it will listen to, i.e., the ***control*** whose ***events*** it will ***handle***. This is done with a method call: **addActionListener()** or **addItemListener()**, for example.

All of this requires that you import a new package into your program: **java.awt.event.*.**

10B.3.2 ActionListeners

A common kind of event is an ***action event***. This occurs when a button is pressed or the user hits the enter key. To ***trap*** an action event (that's a cool way to say to handle an action event) we define our listener class so that it **implements ActionListener**. The ***event handler*** that is associated with such a ***listener*** is called an **actionPerformed()** ***method***.

Here is the most important new concept with event handlers. We do not call them directly as we have done with other methods. They are called for us by the **JVM** or **JRE** during program execution ***if and when the event occurs.***

What follows is an implementation of a listener class which I called **SeeFriendListener** because it is going to handle the ***push event*** of the **JButton** that bears the text **"Push to See Friend."** You will notice that I defined this class inside the **TranspoFrame** class and made sure that it implemented the **ActionListener** interface.

Compare this listing to the comments in this and the last section. Run the program to see how it behaves.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Foothill
{

   public static void main(String[] args)
   {
      // establish main frame in which program will run
      TranspoFrame myTranspoFrame
            = new TranspoFrame("Transporter Room");
      myTranspoFrame.setSize(300, 200);
      myTranspoFrame.setLocationRelativeTo(null);
      myTranspoFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // show everything to the user
      myTranspoFrame.setVisible(true);
   }
}
```

```
// TranspoFrame class is derived from JFrame class
class TranspoFrame  extends JFrame
{
   JButton btnMyButton;
   JTextField txtMyTextArea;
   JLabel lblMyLabel;

   // TranspoFrame constructor
   public TranspoFrame(String title)
   {
      // pass the title up to the JFrame constructor
      super(title);

      // set up layout which will control placement of buttons, etc.
      FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 5, 20);
      setLayout(layout);

      // 3 controls: a label, a text field and a button
      lblMyLabel = new JLabel("Friend's Name: ");
      txtMyTextArea = new JTextField(10);
      btnMyButton = new JButton("Press Here to See Friend");
      add(lblMyLabel);
      add(txtMyTextArea);
      add(btnMyButton);

      // create a Listener for the JButton and register it
      SeeFriendListener sfFriendListener;
      sfFriendListener = new SeeFriendListener();
      btnMyButton.addActionListener( sfFriendListener );
   }

   // inner class for JButton Listener
   class SeeFriendListener implements ActionListener
   {
      // event handler for JButton
      public void actionPerformed(ActionEvent e)
      {
         String friend = txtMyTextArea.getText();
         JOptionPane.showMessageDialog(null,
               "Please wait while we locate " + friend + "...");
      }
   } // end inner class SeeFriendListener
} // end class TranspoFrame
```

This program is very similar to the last version. The only differences are the inclusion of the ***inner listener class,*** and the registration of the listener with the button. We can shorten it by combining these three lines:

```
      SeeFriendListener sfFriendListener;
      sfFriendListener = new SeeFriendListener();
      btnMyButton.addActionListener( sfFriendListener );
```

into a single line:

```
      btnMyButton.addActionListener( new SeeFriendListener() );
```

Notice that we are doing three new things relative to our previous version.

1. We are taking an action when the user clicks the button (this is the work of our event listener)
2. We are reading data from the text field control (see the **txtMyTextArea.getText()** call)
3. We are issuing a **JOptionPane.showMessageDialog()** from our ***event handler*** - thus putting up a new small message window as a result of the user doing something in our main JFrame window.

Some sloppy unfinished business includes making the member data private (rather than default), and checking for bad user input in the text field. We'll do those things in the next version.

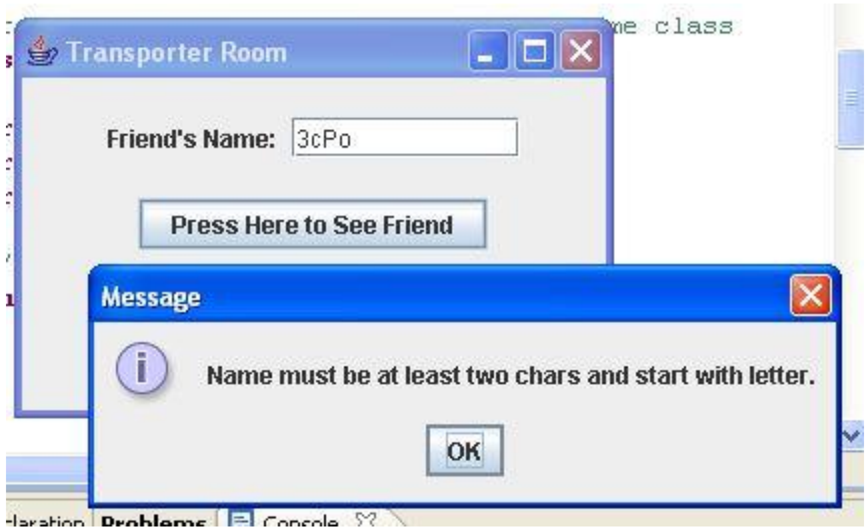
Obviously we haven't beamed our friend into the room yet, but at least now the program is reacting to the user. Progress.

Section 4 - A Discriminating GUI

10B.4.1 Filtering out Bad User Names

Now that the hard work is done, we can have some fun. We use what we know about the language and mix in a couple useful built-in Java methods like **isLetter()**, a ***static class method*** of the class **Character**, and **length()** and **charAt()**, ***instance methods*** of class **String**. I also made the **TranspoFrame** data **private** since there was no need for it to have default access.

First, an example of our program rejecting an unacceptable name from the user:



Name must be at least two chars and start with letter.

Here is the source. Run the program yourself and stand back!

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Foothill
{

   public static void main(String[] args)
   {
      // establish main frame in which program will run
      TranspoFrame myTranspoFrame
            = new TranspoFrame("Transporter Room");
      myTranspoFrame.setSize(300, 200);
      myTranspoFrame.setLocationRelativeTo(null);
      myTranspoFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // show everything to the user
      myTranspoFrame.setVisible(true);
   }
}

// TranspoFrame class is derived from JFrame class
class TranspoFrame  extends JFrame
{
   private JButton btnMyButton;
   private JTextField txtMyTextArea;
   private JLabel lblMyLabel;

   // TranspoFrame constructor
   public TranspoFrame(String title)
   {
      // pass the title up to the JFrame constructor
      super(title);

      // set up layout which will control placement of buttons, etc.
      FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 5, 20);
      setLayout(layout);

      // 3 controls: a label, a text field and a button
      lblMyLabel = new JLabel("Friend's Name: ");
      txtMyTextArea = new JTextField(10);
      btnMyButton = new JButton("Press Here to See Friend");
      add(lblMyLabel);
      add(txtMyTextArea);
      add(btnMyButton);
      btnMyButton.addActionListener( new SeeFriendListener() );
   }
   // inner class for JButton Listener
   class SeeFriendListener implements ActionListener
   {
      // event handler for JButton
      public void actionPerformed(ActionEvent e)
      {
         String strFriendName;

         strFriendName = txtMyTextArea.getText();
         if (strFriendName != null && strFriendName.length() >=2)
         {
            char first = strFriendName.charAt(0);
            if (Character.isLetter(first))
            {
               // good friend's name.  Now confirm
               JOptionPane.showMessageDialog(null,
                  "Please wait while we locate " + strFriendName);
               return;
            }
         }

         // if we fall through they have unacceptable friend's name
         JOptionPane.showMessageDialog(null,
            "Name must be at least two chars and start with letter.");
         return;
      }
   } // end inner class SeeFriendListener
} // end class TranspoFrame
```

How is the logic inside the event handler reacting to good or bad data? What constitutes good or bad data? Should we make this more restrictive? If so, how?