

User Interactions

Goals

In this lab you will learn a little about programming user interactions using the Model-View-Controler pattern for organizing the responsibilities.

The JPT library allows you to concentrate on the key concepts and avoid the pitfalls/details typically associated with GUI programming.

The Model and the View

The diagram below (on the next page) describes the classes already included in this application:

Here is a brief description of the role these files play in the application.

The model

The program deals with balloons (for now just three of them).

- `class Balloon` This class represents one balloon object, allows the user to move it, paint it, and to compare two balloons for closeness to the top of the graphics window.

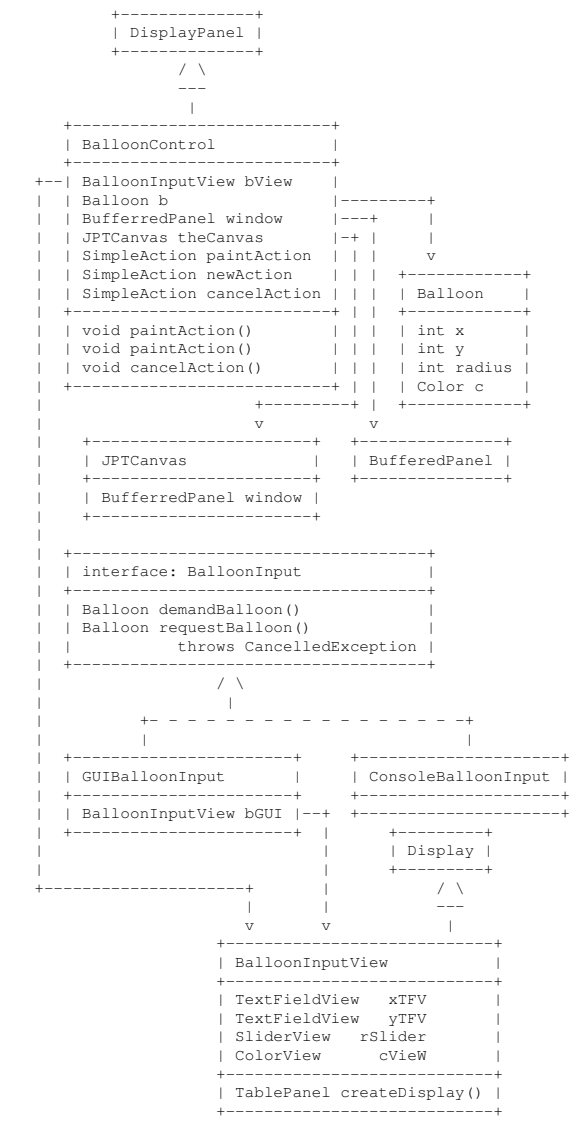
  We could have other classes here, such as a list of balloons, or a list of tied-up balloons and a list of floating balloons, etc.

The views

We can view the information about a particular `Balloon` object in several different ways. The `BalloonInput` interface provides two methods for getting the data needed to construct an instance of a new `Balloon`.

To display the information about a `Balloon` object, we can print a `String` that represents the `Balloon` object in the console, or paint it in the given window, or display the values of its fields in a GUI.

To get the data from the user that is needed to instantiate a new `Balloon` we can read from the console, or from a GUI.


DisplayPanel
BalloonControl
BalloonInputView bView
Balloon b
BufferredPanel window
JPTCanvas theCanvas
SimpleAction paintAction
SimpleAction newAction
SimpleAction cancelAction
void paintAction()
void paintAction()
void cancelAction()
Balloon
int x
int y
int radius
Color c
JPTCanvas
BufferedPanel
BufferredPanel window
interface: BalloonInput
Balloon demandBalloon()
Balloon requestBalloon()
throws CancelledException
GUIBalloonInput
BalloonInputView bGUI
ConsoleBalloonInput
Display
BalloonInputView
TextFieldView xTFV
TextFieldView yTFV
SliderView rSlider
ColorView cVieW
TablePanel createDisplay()

- `interface BalloonInput` contains two methods: `demandBalloon()` and `requestBalloon()` that allow us to instantiate a `Balloon` object from the source that implements the methods.
- `class ConsoleBalloonInput` implements the `BalloonInput` interface used for reading the input from the console.
- `class BalloonInputView` defines a GUI to request the user input for the data needed to initialize one `Balloon` instance. It contains two `TextFieldView`s, one `SliderView`, and one `ColorView`. It also allows us to display the data that represents an instance of a `Balloon`.
- `class GUIBalloonInput` implements the `BalloonInput` interface for extracting the user input from the `BalloonInputView` GUI.

The control

- `class BalloonControl` adds to the GUI `Action`s. These are buttons that allow the user to choose an action, such as *r*ead the `Balloon` data from a GUI and display the `Balloon` in the given canvas. (Our canvas is a window – a buffered panel.)

Build a *Configuration* with `main` method in the `Interactions` class and run the code. Note the behavior in response to the various buttons.

Getting Familiar with the Environment

1. **The model**

   Read the code for the `class Balloon`. Add the method `eraseBalloon` which will paint the balloon in a white color (`Color.white`). Make sure you have the examples and tests for this method.

2. **The console input**

   Read the code for the method `runConsoleInput` in the `class Interactions`. Describe to your partner what the method does. Look at the `ConsoleBalloonInput` class and see how the methods `demandBalloon` and `requestBalloon` are implemented. Run the code and see what happens if you type in a wrong data, or when you do not provide any input.

3. **The actions**

   Find the code for the action for the **New** button in the class `BalloonControl`. Currently, it only reads the data from the `bView`, constructs a new `Balloon` with the fields given by the GUI and makes it the new value of the `Balloon` b.

   Add to this action a call to the method which paints the balloon, from the `class Balloon`. Make sure it works.

4. **Text input from a GUI**

   Find all places where the `xTFV` is defined or used. It is constructed in the class `BalloonInputView`. This class also defines the methods `demandBalloon` and `requestBalloon`, each of them produces a new instance of a Balloon from the user inputs.

   In the class `BalloonControl` user input to the `BalloonView` initializes the value of a `Balloon` object that represents our model.

   Extend the model in the class `BalloonControl` with one more balloon. Define a new actions `b2Action` that initializes and paints the second balloon.

5. **Connecting slider with a text field**

   Look at the class `BalloonInputView`.

   Look how the two `TextFieldViews` that represent the `x` and `y` inputs are defined.

   Define a new `TextFieldView` named `rTFV`, to represent the numerical value of the `Balloon radius`. Give it the *default value* 20.

   Test the behavior of the slider. Does it have any effect on the balloon? Does it have any effect on the value displayed in the `rTFV` field? Change the value of the `rTFV` field. Does it affect the slider? Does it affect the balloon?

   The two views represent the same value and so should be designed to mimic each other. The slider has to act by changing its position whenever a new value is typed into the text field. The value in the text field has to change when the slider is moved, so it reflects its current position.

   In the class BalloonInputView **define** two new `SimpleActions` and the corresponding methods — an `rTFVaction` and a `SliderAction`.

It does not matter what you choose for the label, because we are not going to use the actions with a button.

The first one `void rTFVaction` will be invoked when the value in the field `rTFV` changes. It should set the value of `rSlider` to the value displayed in the `rTFV`. To set the state of the `rSlider` use the method

```
rSlider.setViewState("" + rTFV.getViewState());
```

The second method `void rSliderAction()` will be invoked every time the location of the slider (and the value it represents) changes. It must then set the view state of the `rTFV` calling the method `setViewState` in a manner similar to the above. If you run the program now, you may be surprised to see that these changes have no effect. Can you think of the way to test that the methods work correctly?

6. **Listening to changes in the values**

   Now you have to tell the `rSlider` and the `rTFV` to perform this action when their values change. The following two statements have to be added at the end of the method `void createViews()`:

   ```
   rTFV.addActionListener(rTFVaction);
   rSlider.addSlidingAction(sliderAction);
   ```

   The first one tells the `rTFV` to perform the `rTFVaction` whenever its value changes. The second one tells the `rSlider` to perform the `sliderAction` whenever the position of the slider (and thus the value it represents) changes.

   Test that this works. When entering an ew value in the `rTFV` you need to hit return before the new value registers and affects the slider.

7. **Reporting changes in the model to the view**

   Now that you have seen the method `setViewState`, add such method to the `class BalloonInputView`. Here the method setViewState should take as input an instance of the `Balloon` and set the GUI display values to the values of the fields of the given `Balloon`. To see that is works, we need to modify some of the fields of a `Balloon` instance and invoke the method. Try it.

8. **Adding mouse actions**

   In the last part you will control the balloon with the mouse. You need to define what should happen when the mouse is clicked (or dragged,

or released, etc.). You need to specify which GUI component should *l*isten to the mouse and the user mouse actions. You then need to connect the `MouseListener` with the action it should trigger.

Build a separate frame

The first thing you need to do is to change the manner in which the GUI is displayed. Look at the code in the `class Interactions` for the method `runBalloonControl()`. Copy the methd and rename it `runBalloonControlMouse()`. Replace the line which calls the method `showOKDialog` with the following:

```
JPTFrame.createQuickJPTFrame("Balloon Control", bc);
```

This places the *BalloonControl GUI* into a window that runs *in its own thread*, i.e. independently of the rest of the application. That allows the rest of the application to watch out for the mouse movement and clicks inside of the graphics window.

Run the program and you will see that while the *BalloonControl GUI* is open, you can use all the other buttons in the GUI built by the *Interactions* application.

Define a mouse action

Start by adding the following import statement at the top of the `BalloonControl.java` file:

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

The first mouse action you will build will increase the radius of the balloon by ten, every time you click the mouse. All of this is in the `class BalloonControl`. Start by defining the method `protected void click(MouseEvent mevt)` which does the following:

- Print into the console a message that the mouse was clicked.
- Erase the balloon
- Increase the balloon radius by 10
- Set the view state of the `BalloonInputView bView` to the current values of the balloon. (Only the radius has changed, but it is easier to let the `BalloonView` do the whole job by invoking the method `setViewState`.
- Finally, paint the changed balloon.

9. **Defining and installing Mouse action adapter**

   Install a `MouseActionAdapter` for the `BufferedPanel`as follows:

   - After the definition of the `BufferedPanel`, add the definition:

     ```
     public MouseActionAdapter mouseAdapter;
     ```

   - Inside of the constructor for the `class BalloonControl` first initialize the `mouseAdapter` as follows:

     ```
     mouseAdapter = window.getMouseActionAdapter();
     ```

   - Add the action to perform when the mouse is clicked as follows:

     ```
     // Respond to mouse clicks
     mouseAdapter.addMouseClickedAction(
        new MouseAction() {
            public void mouseActionPerformed(MouseEvent mevt){
               click(mevt);
            }
        });
     ```

     At this point you should test that your program runs as you expected.

10. **Tracking the mouse movement**

    Finally, you will make the balloon move when the mouse moves. Do all the steps you have done for the clicked action, but do not get a new `mouseAdapter`. The following code will add the action:

    ```
    // track mouse motions
    mouseAdapter.addMouseMovedAction(
       new MouseAction() {
           public void mouseActionPerformed(MouseEvent mevt){
              track(mevt);
           }
       });
    ```

    Inside of the `track` method get the coordinates of the mouse as follows:

    ```
    b.x = mevt.getX();
    b.y = mevt.getY();
    ```

    and see what your program does. (Probably nothing - you still have to erase the old balloon, before you make the changes, paint the new balloon, and as a courtesy, set the view state for the view.) Now you should have fun.

Sound Library

The tunes library complements the *world games* packages by allowing the programmer to play a tune (or several tunes) in response to either a tick event or a key event. Our goal is to show students how a completely different kind of information can be represented as data and how musical sequences can be composed into tunes.

Each `Tune` is defined by the instrument that plays the tune and the chord – a collection of notes. `Note` is defined by its pitch and (optionally) its duration. You can play up to 15 instruments plus the percussions at a time. The interface `SoundConstants` contains the names for all *MIDI* instruments, for the 15 instruments in the current *MIDI* program and mnemonics for the pitches in the middle of the piano keyboard range. The World comes with two buckets for carrying the tunes: `tickTunes` and `keyTunes`.

On each tick, the programmer adds to the `tickTunes` bucket the tunes that should start playing at this time. The library then starts playing these tunes, using the timer for measuring the duration of the notes.

A separate `keyTunes` bucket is used to play the desired tunes in response to the key press. The tune plays until the key is released.

At the end of the game the final tunes in the `tickTune` bucket are played for a short time and then both buckets are emptied (the notes they have played are silenced and the bucket contents is cleared).

Initially, the *MIDI* player is initialized with 15 instruments. You can change the instrument selection using the `programChange` method. Here the word *program* refers to the instrument selection for the *MIDI* synthesizer, not a program for our computer.

Music explorations

The *Tunes Demo* applet on the *SoundLib* web pages allows you to try out 30 different instruments and explore the way music can be illustrated graphically.

Run the program and explore what it can do, how it represents the music, how it interacts with the user.

Run the other samples and read the code - especially the parts that are used to generate the music and the sound effects.

Note:

The *MIDI* synthesizers can read *MIDI* files that represent the sequence of notes to be played, the instruments to play, etc. However, the format is

more complex, you need to worry about timing of all notes, specifying the start and stop times, etc. You can also specify the *velocity* — this represents the volume at which the note should be played. You may want to read about it, but for this assignment make it simple: initialize a sequence of notes or chords to play (for example saved as an `ArrayList` of `Note`s or `Chord`s and design a method that will select next chord to play on each tick.

For example, if your note sequence represent the tune *Frere Jacques*, you may design a *canon* by adding the notes from the same sequence, offset by some number of ticks. Or you can play it in two octaves concurrently by changing the pitch of each note by the difference between the octaves (13 pitches). Or you can compose *Bach-like* fugues by playing the same tunes at two different speeds concurrently, playing if forwards and backwards concurrently, and making other such combinations of musical sequences.