Submission Title
GUI Widgets

Contact Information
Richard Rasala
rasala@ccs.neu.edu
Viera K. Proulx
vkp@ccs.neu.edu
College of Computer & Information Science
Northeastern University
Boston MA 02115

Problem Statement

Java has many GUI widgets that may be used to construct GUIs. This submission describes several widgets defined in the *Java Power Tools* that make it easier to use GUIs in pedagogy.

Solution Overview

This section is divided into parts corresponding to the problems that the widgets solve.

A Panel for Painting Graphics

The standard way to paint graphics on a Java component is to override its *paintComponent* method. This algorithmic approach requires either that the graphics code be given inline or that there be some auxiliary data structure that provides the necessary data and instructions. This is not an easy model for beginning students to use.

A simpler model to understand is to imagine that painting is done onto a *canvas* that can capture the results of a painting operation and can accumulate the net result of several such operations. This is the facility provided by the *JPT* `BufferedPanel`.

It is easy to define a `BufferedPanel` by providing its width and height, say, 640 by 480, to give an example.

```
BufferedPanel window = new BufferedPanel(640, 480);
```

The panel `window` will then encapsulate a `BufferedImage` of size 640 x 480 that will capture the results of painting. To paint onto the `window`, you actually paint onto the hidden `BufferedImage` by using the *graphics context* of the `BufferedImage`.

```
Graphics2D g = window.getBufferGraphics();
```

At this point, you may use standard Java 2D graphics calls to set the current paint and current line stroke; to turn on anti-aliasing; to fill or draw a shape; to render an image; or to draw text in a specified font. For example to fill a ***shape***, use:

```
g.fill(shape);
```

See Jonathan Knudsen's text on *Java 2D Graphics* for details.

When the program is ready to display the results of various paint operations, use one of the following calls:

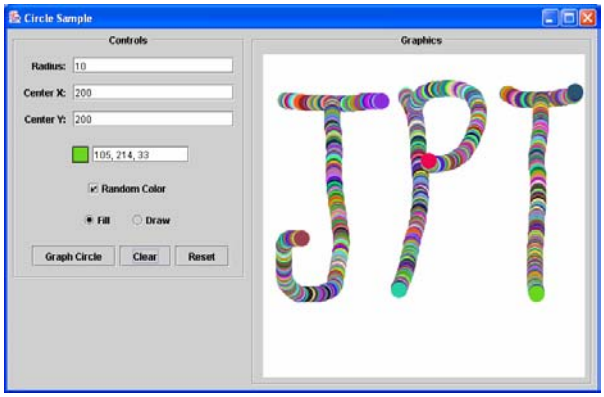
```
window.repaint();       // the standard Java repaint
```

or

```
window.quickRepaint();  // the fast JPT repaint
```

The `quickRepaint()` call will bypass Java's delayed repaint and will repaint immediately. This is useful for animation.

A `BufferedPanel` has an associated `MouseActionAdapter` that allows mouse interaction with the panel. As an example of the effects that may be easily programmed, we include a snapshot of mouse driven graphics from Chapter 1 of the planned *JPT Book* that is posted on the *JPT* web site.



Text Input Widgets

The most fundamental text input widget is `TextFieldView`. This widget is discussed in the submission on *Flexible & Robust Input-Output*. Also, see the description of the `Stringable` interface in that submission.

A `TextFieldView` can be set to read any data type that satisfies the `Stringable` interface and whose input data may be typed using a single line of text. The methods *demandObject* and *requestObject* will produce a `Stringable` object and will automatically check for errors and permit the user to correct them. Convenience methods permit direct input of the 8 primitive types and String, BigInteger, and BigDecimal. If desired, input may be filtered. See the above submission for an example of how `TextFieldView` is used.

The `TextFieldView` constructors are designed to support robust error correction. Usually the caller will supply to the constructor the initial text that is displayed in the text field and the width of the text field. In addition, the caller may also supply:

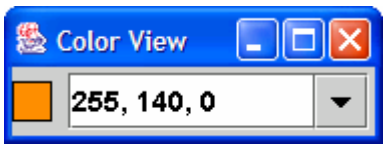
- the *error prompt* to be displayed in an error dialog
- the *dialog title* of the error dialog
- a *suggestion* that may be displayed to let the user see an example of valid input

In this way, a `TextFieldView` is prepared in advance for error handling so that this work need not be done in an ad hoc fashion.

In the most recent release of *JPT*, the `DropdownView` widget that was originally intended for the input of items in a fixed dropdown list has been generalized to permit editable text. This means that a `DropdownView` may now be considered as a text field that has a set of the most common default values supplied in its dropdown list. Taking this new perspective, we plan to add methods to the `DropdownView` class to make its behavior more closely parallel that of `TextFieldView`.

ColorView

The `ColorView` widget is a composite widget that allows the user to select a color from a color swatch; by typing R, G, B, Alpha values in decimal or hexadecimal; or by choosing a color by name from a drop down list. The screen snapshot below shows a `ColorView` alone in a frame.



If the color swatch on the left is clicked then a standard Java color chooser dialog will be opened to select the color. The user may also type the color values into the text field on the right or may select the color by name from the dropdown list. In the above example, the color *dark orange* was selected from the dropdown list. A sample of the color was placed in the swatch and the corresponding R, G, B values were placed in the text field.

Widgets for Strings and String-Object Pairs

In GUI design, a list of strings may be used to initialize a family of radio buttons or a dropdown list. In JPT, we provide several variations of such widgets. In each case, we construct the widget using a `String[]` array to provide all string labels at once. Optionally, the caller may provide an action to be performed whenever any option is selected or an array of actions that are paired with corresponding options.

For radio buttons, the base class is `RadioPanel`. The user will normally select a button interactively. The program can retrieve this selection by requesting the *selected index*, *selected label*, or *selected button*. Using this information, the program can perform an appropriate action or method.

Originally in JPT, the class `RadioPanel` did not exist and radio buttons were handled by `OptionsView`. The class `OptionsView` implements the JPT `TypedView` interface. Since, in practice, we found that we were not often using the features of `TypedView`, we decided to define `RadioPanel` as a base class for radio buttons with no reference to `TypedView`.

Further reflection on the usage of radio buttons in a GUI led us to realize that frequently the role of a radio button is ultimately to select some particular object in the model. This model object may correspond to a data setting or to a choice of algorithm. In any case, we realized that it would be very convenient to be able to get the selected model object directly from the radio button panel without the need to go through the selected index or label. This idea was the rationale for `StringObjectRadioPanel`.

In a `StringObjectRadioPanel`, each string label is associated with a corresponding model object that is specified at construction. This specification may be done directly by passing an `Object[][]` array consisting of string-object pairs or by using a data structure `StringObjectMap` introduced to support this panel. With this data in place, when the user selects a button, the program can request the *selected object* directly and have no concern for the actual index or label of the button.

Since `StringObjectRadioPanel` is so new, only a few samples use its features so far. We expect to use it frequently in future code and to gradually refactor legacy code to use it as well. We have also introduced the class `StringObjectDropdown` to provide the same functionality for dropdown lists.

Other Simple Widgets

The `SliderView` class is a `TypedView` designed to input an `int` value in a bounded range and to support mouse manipulation of the base Java `JSlider` object. The caller may add one or more actions to be executed either while the slider is sliding or when the slider is released.

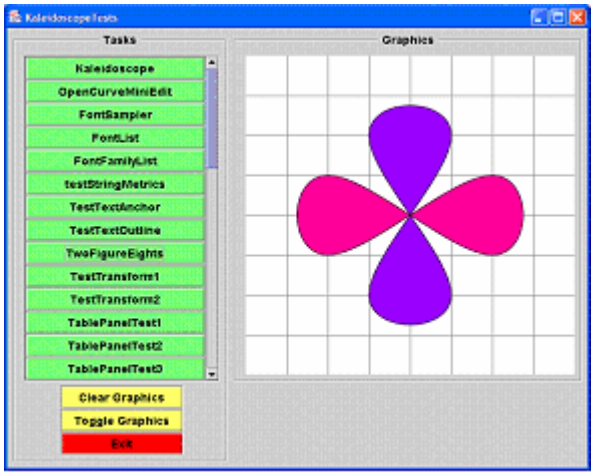
The `BooleanView` class is a `TypedView` designed to input a `boolean` value using a Java `JCheckBox`. If the check box is checked, then the state is true, and if it is unchecked, then the state is false. If the caller wishes some action performed when the user changes the check box state, such an action may be installed.

Scroll Pane Support

Java provides support for scrolling a single component via the wrapper class `JScrollPane`. However, it is often convenient to control the size of the inner viewport window in order to limit scrolling to either vertical or horizontal. This is not easy to do directly in `JScrollPane`.

The class `JPTScrollPane` extends `JScrollPane` and provides the necessary methods to manage the viewport size. One can set the exact size of the viewport or set an upper bound on that size. The latter option is very helpful in practice because it permits the viewport to downsize if it is holding a small component and to maintain a maximum size no matter how large a component is installed.

In the submission on the *Java Power Framework*, we have shown the following screen snapshot of the *JPF* automatic GUI:



The large green panel of automatic buttons is managed using a `JPTScrollPane` that is set up so the maximum viewport width is the width of a button (so no horizontal scrolling is needed) and the maximum viewport height is an exact multiple of the height of a button (so no button appears partially covered). This example shows that viewport bounds can be under algorithmic control.

Experience with the Solution

One of the fundamental goals of *JPT* is to make GUIs easy enough to build that faculty and students will not be hesitant to use them. In this regard, it is essential that there be as few steps as possible from the idea of a GUI to its realization in code. The widgets presented in this submission are rapid to build and provide additional richness of functionality. Combined with the use of `TablePanel` and `TableLayout` for GUI composition and the use of actions to specify behavior, it is possible to build GUIs very quickly. We can create GUIs on the fly in class and have done so as demonstrations in faculty workshops. This is something that would probably not be attempted using the raw tools in Java. All in all, we are quite satisfied with the GUI tools in *JPT* for pedagogical purposes.

API Documentation & Related Materials

The main JPT site to access documentation, code, and the jpt.jar:

http://www.ccs.neu.edu/jpt/