Java Power Tools Introductory Tutorials

Introduction

The Java Power Tools (JPT) have been designed to simplify building graphical user interfaces (GUIs) in the Java language. It is based on the current version of Java and extensively uses the Swing package. All the source code is open, however, and the design is such that it interfaces cleanly with any "pure" Java code.

The abstractions are very powerful and allow the programmer work at a conceptual level without worrying about annoying details. Typically, a single statement encapsulates one logical step such as defining a specific JPT component or adding functionality to a defined component. Though this statement may be somewhat complex, it typically has a standard format and the user only needs to supply one or more parameters. A list of these "JPT idioms" is provided at the end of each tutorial. The programmer can just copy the appropriate idiom, replace the parameters with appropriate values, and include it in their program.

The General Introduction section gives an overview of the model-view communication philosophy on which the entire toolkit is built. It should be read before trying to write even a simple program.

Each of the four tutorials that follow is built around a simple program that illustrates, using a concrete example, how to create a simple GUI component and implement the communication between the model and the view. Each detailed tutorial explains all of the steps in writing the program. In addition, the idiomatic Java statements that are used in each program are summarized in an Idioms section.

General Introduction

Model-View Communication

The key idea in building any GUI is the Model-View-Action paradigm – it is the mantra of the JPT. An external view that represents some data visible to the user communicates with an internal data model used in program's calculations and manipulations, as directed by the actions triggered by the user. In JPT this communication between the view and the model is accomplished by representing the data state as a String.

A collection of JPT Views implements different ways in which the user can view the information about a program or supply the inputs to the program. All views implement the Displayable interface, which specifies that the view state can be set from a String and that the current state of the view can be encapsulated in a String.

Every data model that wishes to display its state in one of the views, or expects to use the state of a view to set its own internal state, needs to implement the Stringable
The following chart explains this idea:

```
Data model -> toStringData -> string -> setState -> View
View        -> getState -> string -> fromStringData -> Data model
```

**The Basic Structure of JPT: The Stringable interface**

The **Stringable interface** is one of the two key mechanisms for communication between a GUI view and an internal data model. Each data model object that wishes to set its state from a JPT GUI component or that wants to display a representation of its internal state in a JPT GUI component must implement the **Stringable interface**.

The **Stringable interface** consists of two methods: `fromStringData` and `toStringData`. In practice, these methods are typically inverses of each other. The `fromStringData` method extracts from an input `String` the information it needs to set the internal state of a data model. For example, if the input `String` is "345" and the data model is an integer, the `fromStringData` method should convert the string "345" into the integer 345. Conversely, if the integer value is 345, the `toStringData` method should then return the `String"345"`.

A special set of classes -- that all inherit from an abstract `XObject` class that implements the **Stringable interface** -- are provided. There is one `XObject` for each primitive Java data type (`XInt`, `XDouble`, `XBoolean`, etc.). In addition, numeric `XObject`s are also subclasses of `XNumber`, which allows for generality among numeric types.

**The Basic Structure of JPT: The Displayable interface**

The **Displayable interface** is the second key mechanism for communication between a GUI view and an internal data model. Each JPT GUI object that wishes to communicate with data model objects must implement the **Displayable interface**.

The two key methods in the **Displayable interface** are `setState` and `getViewState`. The first method, `setState`, uses a `String` to modify the values displayed in the GUI component. In some cases the `String` appears in the GUI component "as is". In other cases, the value of the `String` is represented in a different fashion. For example, a `ColorView` uses a string encoding of three component colors to
select the color displayed in the view, and a BooleanView displays its current state as a checked or unchecked checkbox.

The getViewState extracts the information represented by the GUI view and encodes it as a String. Such a String could then be passed to a data model in order to set its internal state. This method is not responsible for error correction, so if the user types in "abc" and the data model is looking for an integer, the getViewState method will faithfully return the string "abc", leaving the error correction to other parts of the JPT. If the GUI view is a check box, the String that is returned will be "0" or "1". A ColorView will return a string encoding of its three component colors.

The following chart explains this idea:

Data model -> toStringData -> String -> setViewState -> View
View         -> getViewState -> String -> fromStringData -> Data model

**The Basic Structure of JPT: Error Handling and Safe Input - The TypedView interface**

User input entered into a view can contain errors. The basic mechanism for communication between the views and models delivers the user input "as is": without any error checking. However, several views are designed to support safe and robust input in situation where correcting erroneous input is necessary. GUI components that implement the TypedView interface must define two functions: requestObject and demandObject. These functions represent the two modes for extracting a data model from a view. In each mode a dialog box appears when the input String is inappropriate for the desired data model. In the mandatory mode, the user cannot exit the error dialog box until valid input is given. In the optional mode, the user can decide not to supply valid data and instead to cancel the input operation. If the user decides to cancel the input operation, an exception is thrown to notify the program that the action dependent on the input should be halted. In each mode, the programmer can choose to supply a suggestion of a possible valid input.

The TextFieldView is one of the GUI components that implements the TypedView interface. It is used to display a value that typically can be represented by a reasonably short string and to record user input that can be entered on one line. Most often this would be a numeric value or a simple String. The TextFieldView determines the specifics of its error strategy by observing the class of the object that found the user input to be in error.

Because a TextFieldView can display and provide data for a number of different objects, convenience methods are provided so that a programmer can demandDouble or requestBoolean, etc.
The Basic Structure of JPT: The GUI components

The basic JPT GUI component is a `DisplayPanel`. This is a `Displayable` JPanel that recursively propagates `Displayable` method calls to contained components that are also `Displayable`. This class is the base class for all containers provided by the JPT.

A `Display` is a panel containing a `Displayable` object with the option of a titled border and an annotation. The inner class `Display.Settings` encapsulates the four values that determine the annotation and title of a `Display`, as well as their relative orientations with regard to the `Displayable` object it contains.

A `DisplayCollection` is a dynamic linear collection of `Displayable` objects, displayed using either a horizontal or vertical layout.

A `DisplayWrapper` is a wrapper for a single `Displayable` object that uses a CenterLayout and faithfully respects minimum, maximum, and preferred sizes.

The most important fact here is that there can be a `Display` within a `DisplayCollection` within a `Display` within a `DisplayPanel` – because all of the components implement the `Displayable` interface. This means that the entire state of such a GUI can be extracted with one `getViewState` call and the state of an entire GUI can be set with one `setViewState` call. Similarly, a `reset` call for the main GUI component recursively propagates `reset` calls to all of the components it contains.