The Automatic Problem Set Framework Tutorial

Introduction

The automatic Problem Set Framework is a Java framework that can be effortlessly tailored to create simple problem sets or to create test code for other Java classes. There are two Java classes that make this possible: `ProblemSetClass` and `ProblemSetApplication`. The file `ProblemSetClass.java` is the Java file where new “problems” may be solved by the implementation of simple member functions. Indeed, for every member function in the class `ProblemSetClass` that is `public`, `synchronized`, `void`, with no arguments, and not `static`, the application class will automatically create a button in the user interface that executes this function. To see how this works, let us look a snapshot from the main window of the tutorial version of the Problem Set Framework and then see the code behind the snapshot.

![Snapshot of the Problem Set Framework](image)

The 6 buttons `PrintTo10`, `SumTwoNumbers`, `SumWhileReading`, `MakeArrayAndSum`, `PaintCircles`, and `ReadAndEvalDouble` correspond to the member functions of the class `ProblemSetClass` that are `public`, `synchronized`, `void`, with no arguments, and not `static`. 
As we will see, there are other member functions of the class `ProblemSetClass` that act as helper functions but do not generate buttons in the GUI because they do not satisfy the conditions `public, synchronized, void, with no arguments, and not static`. The last two task buttons, “Clear Window” and “Exit”, are always automatically added to the GUI in the Problem Set Framework.

The Sample Methods in ProblemSetClass

The sample methods in the `tutorial version` of the class `ProblemSetClass` serve as a guide to show the programmer how to define methods that will generate buttons in the GUI and how to define helper methods that will not be visible in the GUI. In practice, after a programmer has examined these sample methods, the methods may be deleted and replaced with the actual methods to be executed in the Problem Set Framework. The `vanilla version` of the Problem Set Framework provides a version of the file `ProblemSetClass.java` that has no sample methods and is ready to use for a fresh project.

In this tutorial, we will examine each of the sample methods in the `tutorial version` of the file `ProblemSetClass.java` to explain what techniques the methods illustrate.

```java
/** Sample 1. */
public synchronized void Print1to10() {
    for (int i = 1; i <= 10; i++)
        console.out.print(i + " ");
    console.out.println("\n");
}

Sample 1 illustrates a simple output loop that prints the numbers 1 to 10 and then prints two newlines. It uses the Java Power Tools `console` object that will be explained in detail in the appendix to this tutorial. The methods `console.out.print` and `console.out.println` print an object to the separate `console` window. The `console.out.println` method adds a newline after printing the `String`. These methods work in exactly the same fashion as the standard Java methods `System.out.print` and `System.out.println`.

```java
/** Sample 2. */
public synchronized void SumTwoNumbers() {
    int x = console.in.demandInt("Enter x:");
    int y = console.in.demandInt("Enter y:");
    console.out.println("Sum: " + (x + y) + "\n");
}

Sample 2 illustrates simple input using the `demand` family of `console.in` methods. These methods will prompt for input and check for errors. If a valid value is entered, it will be returned to the caller. Otherwise, an error message will be printed and the user will be prompted for the input again.
/** Sample 3. */
public synchronized void SumWhileReading() {
    int sum = 0;
    XInt x = new XInt();
    while (console.in.reading("Enter value to sum:", x))
        sum += x.getValue();
    console.out.println("\nSum: " + sum + "\n");
}

Sample 3 illustrates loop input in which the console.in.reading method is used to control a loop. If the user enters input at the prompt, it is stored in the XInt parameter x and the reading method returns true so that the loop body may be executed. If the user simply presses return at the input prompt, the reading method returns false and the loop terminates. Note that the class XInt is a Java Power Tools class that encapsulates an int that is mutable which means that its value may be changed after the XInt object x has been constructed. It is essential that x be mutable since otherwise the console.in.reading method would be unable to set its value during the input operation.

/** Sample 4. */
public synchronized void MakeArrayAndSum() {
    int[] data = null;
    boolean random = console.confirm("Make random data?", true);
    if (random)
        data = randomIntArray();
    else
        data = readIntArray();
    // ask to echo data
    if (console.confirm("Show the data?", random))
        printIntArray(data);
    else
        console.out.println();
    // compute and print sum
    int sum = 0;
    for (int i = 0; i < data.length; i++)
        sum += data[i];
    console.out.println("Sum: " + sum + "\n");
}

Sample 4 illustrates the use of console input-output, dynamic arrays, and helper methods. Using the console.confirm function, the method MakeArrayAndSum determines whether the user wants a random array of data or an array that is entered manually from the keyboard. In either case, a helper function is used to obtain the data. Then the user is asked whether or not to print the array data. The default is set to true if the data is random and false if the data has been entered manually. Finally, the method concludes by doing a simple calculation, a sum, to illustrate the use of the array.
/** Sample 5. */
public synchronized void PaintCircles() {
    // do graphics setup
    window.clearPanel();
    window.repaint();

    Graphics2D G = window.getBuffer Graphics();
    Ellipse2D.Double E = new Ellipse2D.Double();

    // request the number of circles from a dialog
    int n = SimpleDialog.demandInt
        ("How Many Circles?", "PaintCircles", "100");

    // draw filled circles
    for (int i = 0; i < n; i++) {
        // define diameter with 10 <= diameter <= 50
        int diameter = 10 + (int)(40 * Math.random());
        int maxX = window.getBufferWidth() - diameter;
        int maxY = window.getBufferHeight() - diameter;

        // define top-left corner
        int x = (int)(maxX * Math.random());
        int y = (int)(maxY * Math.random());

        // define circle
        E.setFrame(x, y, diameter, diameter);

        // define r, g, b color
        int r = (int)(255 * Math.random());
        int g = (int)(255 * Math.random());
        int b = (int)(255 * Math.random());

        // fill the random circle with random color
        G.setPaint(new Color(r, g, b));
        G.fill(E);
    }

    // refresh
    window.repaint();
}

Sample 5 illustrates the use of the 400x400 graphics window object built into the Problem Set Framework. From the window object, we extract its graphics context object G and use G to paint the circles. All of the facilities of Java 2D graphics are available using the graphics context G. See Jonathan Knudsen, Java 2D Graphics, O’Reilly, 1999 for more information.

The method PaintCircles opens by clearing the graphics window panel. Then, we call repaint to make this action immediately visible to the user. Next, we obtain the graphics context object G from the window object and we create an ellipse object E that will be used to paint all of the circles. Next, we open a dialog box to ask for the number of circles to be painted. Next, we execute a loop to draw the desired number of circles with random diameter, location, and color. This code is pure Java 2D graphics. Finally, we call repaint again to make the newly painted circles visible.
/** Sample 6. */
public synchronized void ReadAndEvalDouble() {
    // build the read and eval text fields
    final TextFieldView read = new TextFieldView("", 200);
    final TextFieldView eval = new TextFieldView("", 200);

    // place the read and eval text fields with prompts into a panel
    TablePanel readAndEvalPanel = new TablePanel(
            new Object[][] {
                { "Enter Double Expression:", read },
                { "Value of the Expression:", eval }
            },
            5, 5, EAST);

    // define the read and eval action
    Action readAndEvalAction = new SimpleAction("Read And Eval Double") {
        public void perform() {
            eval.setViewState("" + read.demandDouble());
        }
    }

    // define the read and eval dialog
    GeneralDialog readAndEvalDialog = new GeneralDialog(
            readAndEvalPanel, "Read And Eval Double",
            new Object[][] {
                { readAndEvalAction, DialogAction.KEEP_OPEN }
            });

    // make the read and eval button the default button
    readAndEvalDialog.setDefaultButton(readAndEvalAction);

    // open the read and eval dialog
    readAndEvalDialog.setVisible(true);
}

Sample 6 illustrates the use of a general dialog with two text fields, two prompts, and a button with the label “Read And Eval Double”. The user may enter a double precision arithmetic expression in the upper text field and then click the button or press return to see the numeric value of this expression in the lower text field. There is no visible code for evaluating the input expression since Java Power Tools automatically does arithmetic expression parsing on all input operations. If the user has typed an error, an appropriate error dialog will appear automatically and will assist the user in correcting the error.

The first two statements in Sample 6 create the upper text field called read and the lower text field called eval. The third statement places these text fields together with their prompts into a table panel that will form the upper half of the dialog. The fourth statement defines the action that will be triggered by the button in the dialog. This action asks the upper read text field to evaluate its contents as a double; the double is then converted back to a numeric String using the Java idiom "" +; and finally the String is set as the contents of the lower eval text field. The fifth statement creates the dialog using the table panel and the action. The dialog is instructed to remain open when its button is triggered. The sixth statement informs the dialog that its button is a default button meaning that it should also be triggered if the user presses the return key. Finally, the last statement opens the dialog for use.

A screen snapshot of the dialog created by Sample 6 is shown on the next page.
The expression in this dialog computes, as an example, the famous golden ratio that appears in a study of the Fibonacci numbers and in many situations in the natural world.

The Helper Methods in ProblemSetClass

The 6 methods described above are the only ones that generate buttons in the GUI because these methods are the only ones that satisfy the conditions public, synchronized, void, with no arguments, and not static. In practice, we make all helper methods protected so that they will not generate GUI buttons. We now describe the helper methods found in the tutorial version of the class ProblemSetClass.

```java
/** Return an int value >= 0 that is read from the console. */
protected int demandPositiveInt(String prompt) {
    int value = 0;
    do {
        value = console.in.demandInt(prompt);
        if (value < 0)
            console.out.println("Input value must satisfy: value >= 0");
    } while (value < 0);
    return value;
}
```

The method demandPositiveInt uses console.in.demandInt to read an integer value from the user. If this value is greater than or equal to zero, it is returned to the caller. Otherwise, an error message is printed and the user is prompted to enter the value again. This helper method illustrates the fact that it is easy to adapt the generic demand methods to require constraints on the input value.

```java
/** Function to read an array of int from the console. */
protected int[] readIntArray() {
    int length = demandPositiveInt("Data array length:");
    int[] data = new int[length];
    for (int i = 0; i < length; i++)
        data[i] = console.in.demandInt("data[" + i + "]:");
    console.out.println();
    return data;
}
```
After requesting the array length from the user and then constructing a new `int` array of the desired length, `readIntArray` loops to fill the array with data using `console.in.demandInt` to read the data.

```java
/**
 * Function to create a random array of int with the size read from
 * the console.
 */
protected int[] randomIntArray() {
    int length = demandPositiveInt("Data array length:");
    int min = console.in.demandInt("Data minimum:");
    int max = console.in.demandInt("Data maximum:");
    int[] data = new int[length];
    for (int i = 0; i < length; i++)
        data[i] = MathUtilities.randomInt(min, max);
    console.out.println();
    return data;
}
```

The method `randomIntArray` must ask the user not only for the array length, it must also ask for the minimum and maximum array values. Once this information is obtained, the method fills the array using the utility method `MathUtilities.randomInt`.

```java
/** Function to print an array of int to the console. */
protected void printIntArray(int[] data) {
    if (data == null) {
        console.out.println("Null data array");
    } else {
        int length = data.length;
        console.out.println("Data array length: "+ length);
        for (int i = 0; i < length; i++)
            console.out.println("data[" + i + "] = " + data[i]);
    }
    console.out.println();
}
```

The method `printIntArray` uses straightforward tools of the `console` object to print the array data.
Using The Problem Set Framework to Solve Problems

To use the Problem Set Framework, you must define your own methods. Normally, you would delete the sample methods or modify these methods to suit your needs. Of course, if any of these sample methods are directly useful to you then you may retain them in your code. You may also start with the *vanilla version* of the Problem Set Framework and begin with a clean slate.

The crucial issue is to decide which methods you wish to directly execute via buttons in the GUI. These methods must be *public, synchronized, void, with no arguments,* and *not static.* All other helper methods must fail to satisfy these conditions in at least one way. The easiest way to accomplish this requirement is to declare helper methods *protected.*

You may not only create methods within the *ProblemSetClass,* you may also create additional classes that you can then add to your project. In that way, you can utilize the code in these classes for the solution of your problems. You may also turn this technique inside out. You may use the *Problem Set Framework* to write test code to confirm that other classes are working correctly. We use this technique ourselves to test the Java Power Tools and it is the most efficient technique we know about to write systematic tests for classes.

In general, we recommend that you *do not change* the *ProblemSetApplication* class in any way and that you *do not rename* the *ProblemSetClass.* These rules will ensure that everything will work together properly. If you need to distinguish between several problem sets, the simplest method is to change the name of the enclosing folders in the file system rather than doing this by changing the file names of the Java files.

If you wish to change the title of the window frame for the framework GUI, you should change the following member data variable in the *ProblemSetClass*:

```java
/** Change this title if you desire a special frame title. */
public String title = null;
```

By default, the Problem Set Framework opens a separate console window and provides a graphics panel in the same window as the task buttons. This behavior is controlled by the following variables in the *ProblemSetClass*:

```java
/** If true, show console. */
public final boolean showConsoleWindow = true;

/** If true, show graphics window. */
public final boolean showGraphicsWindow = true;
```

If you do not need one or the other of these windows, simply set the corresponding variable to *false.*
Appendix: Quick Summary of Console Functions

Input Functions

Input functions from the `console` use the prefix `console.in` followed by a particular input function. Let us, for example, look at integer input. There are two forms:

```java
int x;          // the variable that will get a value
int d = ...;    // some default value used in form 2
// form 1
x = console.in.demandInt(prompt);
// form 2 with a default value d
x = console.in.demandInt(prompt, d);
```

Here `prompt` represents a `String` argument that will be printed in the `console` to signal to the user that input is being requested. The `prompt` may be a `String` constant, a `String` variable, or even the return value of a `String` function.

In the `demand` functions, the user must supply valid data. If the data has an error, the demand functions will prompt and prompt until the error is corrected. Thus, the program can be guaranteed that it will receive valid data to work with. In the case of form 2, if the user simply presses return, then the default value `d` will be returned. This default value is, of course, displayed in the `console` after the `prompt`.

There are demand functions for each of the other primitive types that are named `demandByte`, `demandShort`, `demandLong`, `demandFloat`, `demandDouble`, `demandChar`, and `demandBoolean`. There are also demand functions for the `String` type named `demandString`.

The Java Power Tools has the general notion of a `Stringable` type that means a type whose data state can be encapsulated into a `String` and can be set using a `String`. There are demand functions for `Stringable` objects `X` that have the method signatures:

```java
public void demand(String prompt, Stringable X);
public void demand(String prompt, String default_data, Stringable X);
```

Thus, the methods calls on `X` would look like:

```java
// form 1
console.in.demand(prompt, X);
// form 2 with default data
console.in.demand(prompt, default_data, X);
```

There are `Stringable` types `XByte`, `XShort`, `XInt`, `XLong`, `XFloat`, `XDouble`, `XChar`, `XBoolean`, and `XString` built into the Java Power Tools.
In many instances, you would like to prompt the user for data but if the user has no more data to supply then the program should simply move onward. The way to accomplish this is with the `reading` function that returns `true` if the user did supply data and `false` otherwise. The best way to understand this function is through an example.

```java
XInt X = new XInt();
while(console.in.reading(prompt, X)) {
    int x = X.getValue();
    // now do what you want with the value x
}
```

In this example, the loop continues precisely as long as the user supplies data for `X`. It stops when the user simply presses return.

It is possible to prompt for more than one data item and stop if any item is missing.

```java
XInt X = new XInt();
XInt Y = new XInt();
while(console.in.reading(Xprompt, X) && console.in.reading(Yprompt, Y)) {
    int x = X.getValue();
    int y = Y.getValue();
    // now do what you want with the values x and y
}
```

There are three functions that control input-output flow rather than specifically supply data so these are called with `console` rather than with `console.in`. These functions calls look like:

```java
// ask a yes-no question and return true-false as a result
// the boolean value default_response is printed as Y or N
// if the user presses return then the default_response is returned
if (console.confirm(question, default_response)) ...
while (console.confirm(question, default_response)) ...

// print the standard prompt and wait until the user presses return
console.pressReturn();

// print the supplied prompt and wait until the user presses return
console.pressReturn(prompt);
```

There are some more specialized `console` functions that are explained in the JPT javadocs.
Output Functions

The object `console.out` is actually a standard Java `PrintStream` object that comes equipped with `print` and `println` functions for the standard 8 primitive types and for the `String` type. The `println` function does a `print` and then adds a `newline`. Thus typical output statements have the form:

```java
int x = ...;
console.out.print(x);
console.out.println(x);
console.out.print("some message ...") + x);
console.out.println("some message ...") + x);
```

Remarks

The functionality of the objects `console.in` and `console.out` mimics as far as possible the behavior of the standard input and output objects `System.in` and `System.out`. On the input side, much additional functionality has been added including direct input of the numeric types and robust error checking and correction. Moreover, JPT always performs full numeric expression evaluation on numeric input so, for example, it is possible for the user to enter expressions of the form:

```
2.5 + 3.7/4.9
sqrt(2)
sin(0.7 * pi)
```

as well as entering simple numbers.