## Submission Title

***Flexible & Robust Input-Output***

## 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

It is well known that traditional Java input-output is difficult to program and requires a knowledge of arcane details. Even though Java 1.5 promises some improvements, input-output is still not as flexible and robust as possible. In our view, input-output should:

- Work, as far as possible, in the same way in a console or in a GUI component;
- Enable two styles of input: a *demand* style in which the user must supply input and a *request* style that allows the user to cancel a pending input operation;
- Support the most common data types directly;
- Be extensible to user-defined data types;
- Do robust error detection and correction automatically;
- Permit the use of input filters to apply constraints;
- Automatically evaluate mathematical expressions in the user input data;
- Be really easy to use.

The input-output facilities of the Java Power Tools (JPT) are designed to meet these constraints. In order to quickly explain these tools, we will first give several examples and then discuss the general principles.

## Solution Overview: Examples

Let us begin with the minimal information needed to understand the examples.

For console input-output, JPT defines a `console` object that has numerous methods to support robust typed input. The `console` may use the standard system window or may use its own window in which the data streams are color-coded with normal output in black, error output in red, and user input in blue. The input methods for the `console` always require a prompt string and optionally permit a default string and/or a filter. This *prompt-response* paradigm is more user friendly than a model that views interactive console input as the problem of parsing an unprompted data stream.

For GUI input-output, JPT defines `TextFieldView` for text field input and `OptionsView` for text field input with a dropdown list to provide multiple default options. Since, in a GUI, prompt strings are usually handled via separate labels and since a default may be used to initialize the view, the only optional parameter for an input method is a filter.

In both situations, input methods are provided to support the *demand* style and the *request* style. *Request* methods will throw a `CancelledException` if the user decides to cancel.

*Scenario 1*: In the console, demand two double precision numbers to set an x,y pair.

```
double x = console.in.demandDouble("x:");
double y = console.in.demandDouble("y:");
```

Here is sample screen output in which the user enters 37, 53.

```
x: 37
y: 53
```

*Scenario 2*: In the console, demand two double precision numbers to set an x,y pair and make 200 be the default in each case.

```
double x = console.in.demandDouble("x:", 200);
double y = console.in.demandDouble("y:", 200);
```

Here is sample screen output in which the user accepts 200 for x but changes y to 150.

```
x: [200.0]
y: [200.0] 150
```

*Scenario 3*: In the console, demand an integer percent value that is constrained to fall between 0 and 100.

```
RangeFilter filter
  = new RangeFilter.Long(0, 100);
int percent = console.in.demandInt
  ("Enter percent:", filter);
console.out.println
  ("Valid percent: " + percent);
```

Here is sample screen output in which the user first attempts to enter the invalid percent of 125 and then enters a valid percent of 25 after being informed that the original value is not in range.

```
Enter percent: 125
Value not within the range [0,100]
Enter percent: 25
Valid percent: 25
```

Note that error detection and correction is automatic and does not need to be added in some ad hoc fashion at the site of the input call. Note also that a RangeFilter for long data will work fine with integer input.

*Scenario 4*: Demand an integer percent between 0 and 100 from a TextFieldView `tfv` that has been defined and installed in a GUI.

The definition of `filter` is the same as in Scenario 3. To obtain `percent` from `tfv`, use:

```
int percent = tfv.demandInt(filter);
```

In this case, if the text field has invalid data then a dialog will be automatically opened that prints the same error message as above and allows the user to correct the error.

*Scenario 5*: Assume that a class `Address` has a constructor that, for simplicity, requires 3 strings: street, city, state. Write a factory method to demand an `Address` from the console.

```
Address demandAddress(String prompt) {
  console.out.println(prompt);
  return new Address(
    console.in.demandString("Street:"),
    console.in.demandString("City:"),
    console.in.demandString("State:"));
}
```

This method would be `public static` in the `Address` class.

*Scenario 6*: Assume that a class `Person` has a constructor that, for simplicity, requires a name string and an `Address`. Write a factory method to demand a `Person` from the console.

```
Person demandPerson (String prompt) {
  console.out.println(prompt);
  return new Person(
    console.in.demandString("Name:"),
    Address.demandAddress("Address:"));
}
```

This method would be `public static` in the `Person` class.

Scenarios 5 and 6 show that it is possible to leverage a constructor for a class to define a factory method to demand an object of that class created using input from the console. If the class has several constructors then one must decide how to name the various factory methods but other than that the process is straightforward.

It is also possible to define input methods for classes using data in GUIs but we will not describe this within this short submission.

In all the scenarios so far, we have used *demand* methods rather than *request* methods. Since *request* methods may throw an exception if the user decides to cancel, the code must be more complicated due to the necessary *try-catch* clauses. This variation may be postponed until students are ready to learn exceptions and may then provide a motivating example of the value and usage of the exception concept. The next scenario illustrates both how to use *request* methods and the fact that JPT automatically parses mathematical expressions.

*Scenario 7*: Write a loop to evaluate one or more double precision mathematical expressions.

```
double x;
try {
  while (true) {
    x = console.in.requestDouble("Expression:");
    console.out.println("Value: " + x);
  }
}
catch (CancelledException ex) { }
```

Here is sample screen output from the execution of this loop.

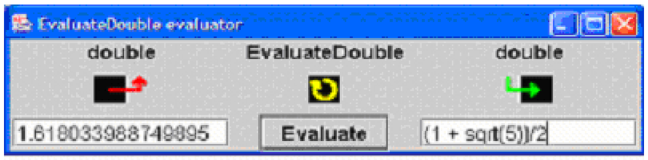
```
Expression: 1*2*3*4*5*6*7*8*9*10
Value: 3628800.0
Expression: exp(1)
Value: 2.718281828459045
Expression: sin(pi/3)
Value: 0.8660254037844386
Expression: (1 + sqrt(5))/2
Value: 1.618033988749895
Expression:
```

As the screen output indicates, if the user simply presses return when asked for input then that is interpreted as a cancel request.

*Aside*: In a separate submission, we have described the *Java Power Framework (JPF)*. Suppose we define the following simple method in that framework:

```
public double EvaluateDouble(double x) { return x; }
```

Then, the following GUI frame is created to invoke this method:



Notice that the input expression *(1 + sqrt(5))/2* in the right hand input text field is evaluated to *1.618033988749895* in the left hand output text field. This is because JPT uses the same code for input expression evaluation in the console and in a text field.

## Solution Overview: Principles

In this section, we rapidly describe the principles and constructs that form the conceptual infrastructure for the tools illustrated in the examples.

In general, when an input operation is performed, one can either change the state of an existing object or one can use some form of factory method to create a new object based on the input data. The input tools in the JPT support both possibilities.

To support input that can change the state of an object, JPT defines the `Stringable` interface. This interface requires two methods: *fromStringData* and *toStringData*. The method *fromStringData* sets the object state from a suitable text string or throws a parse exception if that string is invalid. The method *toStringData* returns a string that encapsulates the object state and may be used to reset that state later. JPT defines basic `Stringable` classes for the 8 primitive types and for String, BigInteger, BigDecimal, Color, and Point2D. JPT also provides recursive mechanisms to permit a programmer to define more complex `Stringable` types.

If the `Stringable` interface is used directly for input, then the caller must be prepared to handle a parse exception. *In practice, we have found it much more convenient to encapsulate the transformation from an input string to an object within factory methods that totally automate the error recovery process.*

For both console and GUI input, the foundational factory methods are *demandObject* and *requestObject* that create and return a suitable `Stringable` object. In both methods, if an input error occurs, an automatic error recovery process starts that describes the error and allows the user to correct it. The difference between *demand* and *request* is that in the *demand* method the user must eventually supply valid input data whereas in the *request* method the user is given the option to cancel the pending input operation. In addition to these general methods, JPT provides convenience methods for the input of the standard types and the simplest object types. These methods have been illustrated in the examples.

The input methods defined in JPT are extremely easy to use in the situations that are most common in introductory computing. In addition, hooks are provided that generalize the techniques to complex model objects and correspondingly complex views.

## Experience with the Solution

The student experience with the JPT input-output methods is very positive. Students can focus on the design of classes and their associated algorithms because input-output can be done without hassle. It is easy to write console test code and it is easy to build GUIs for more sophisticated input-output requirements.

## API Documentation & Related Materials

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

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