Submission to the ACM Java Task Force
The Need for a Console Class

Eric Roberts
Stanford University
eroberts@cs.stanford.edu

Summary

This proposal defends the utility of a **`Console`** class that supports text-based I/O similar to that of a traditional console. At the same time, the proposal does not seek to specify a detailed design for such a class. There are many tradeoffs in such a design that need to be considered, and I think it would be most appropriate for the Task Force to make the final decision based on an examination of the many designs for such a class that already exist.

1. The problem

In the papers that served as background to the Java Task Force's "Taxonomy of Problems in Teaching Java" [8], by far the most widely cited problem—identified as problem A4 in the report—was the lack of any simple facility for accepting input from the user. In the weeks leading up to SIGCSE 2000, the SIGCSE-MEMBERS list carried an active discussion on this topic that was summarized later that year in a paper by Scott Grissom [3]. The upshot of the SIGCSE-MEMBERS discussion, as expressed in the summary paper, was that the community needed "to create a Java package to support I/O operations."

In my taxonomy of problems in teaching Java, I concluded that the problems of user input would be "largely fixed in Java 1.5" through the introduction of the **`Scanner`** class in the **`java.util`** package. That assessment may have been overly optimistic and seems in any case premature. While there is no question that the **`Scanner`** class improves the situation by giving novices access to simple input methods, several problems remain unresolved. Chief among these is choosing an appropriate input source. The standard stream **`System.in`** is an obvious candidate, but suffers from the fact that it is not always easily accessible. Several runtime environments, particularly in browsers, hide the window containing the standard input and output streams, making it difficult to use **`System.in`** as the default input source.

2. The idea of a `Console` class

A common solution to the problem of establishing a source for user input is to create a **`Console`** class that provides an interactive window for input and output within the framework of the standard Java graphical APIs. Such classes have been implemented on many occasions by developers and educators within the Java community, although they have not as yet been standardized. The Java textbooks published by Holt Software Associates [4], for example, define a **`Console`** class that has the necessary functionality. Similar packages have also been developed by David Eck at Hobart and William Smith College [1] and by Ron Poet at the University of Glasgow [5]. (Other texts, such as Bruce Eckel's *Thinking in Java* [2], specify a **`Console`** class but implement it using static methods that make it hard to use as a general tool.)

Although any of these implementations would be appropriate as a starting point for the Java Task Force, it is easiest to describe the intent of the mechanism in the context of my own version of the **`Console`** class, as described in the abbreviated **`javadoc`** listing in Figure 1. Using this implementation, one first creates a new **`Console`** object with a declaration like

```
Console c = new Console();
```

installs that **`Console`** object as a Swing component to ensure that the console window is displayed on the screen, and then uses the methods provided by the **`Console`** class to read and write data to that window. One could, for example, prompt the user for the size of a data set using the following code:[†]

```
c.print("Enter the size of the sample: ");
int sampleSize = c.readInt();
```

3. Is `Console` merely a throwback to imperative code?

The standard objection to the inclusion of a class like **`Console`** in a teaching library is that the underlying computational paradigm, as illustrated by the examples in the preceding section, seems fundamentally imperative in its structure. In the "Taxonomy of Problems in Teaching Java" article [8], I note that several authors

> have suggested that a console-like input facility is a throwback to an out-of-date procedural programming style and that such techniques should be replaced by dialog input that fits the more modern, interactive style.

While I am sympathetic to the philosophical position represented by such objections, I believe that it would be useful for the Task Force to endorse some implementation of a **`Console`** class for the following reasons:

• *Such a class has considerable utility in its own right.* In my own programming, I've found quite a number of uses for the **`Console`** class beyond having its obvious role as a framework for implementing traditional imperative programs. Interpreters, for example, can use it to provide a *read-eval-print* loop capability. Similarly, I have found it indispensible in coding interactive test suites. All in all, I've found it an extraordinarily useful tool.

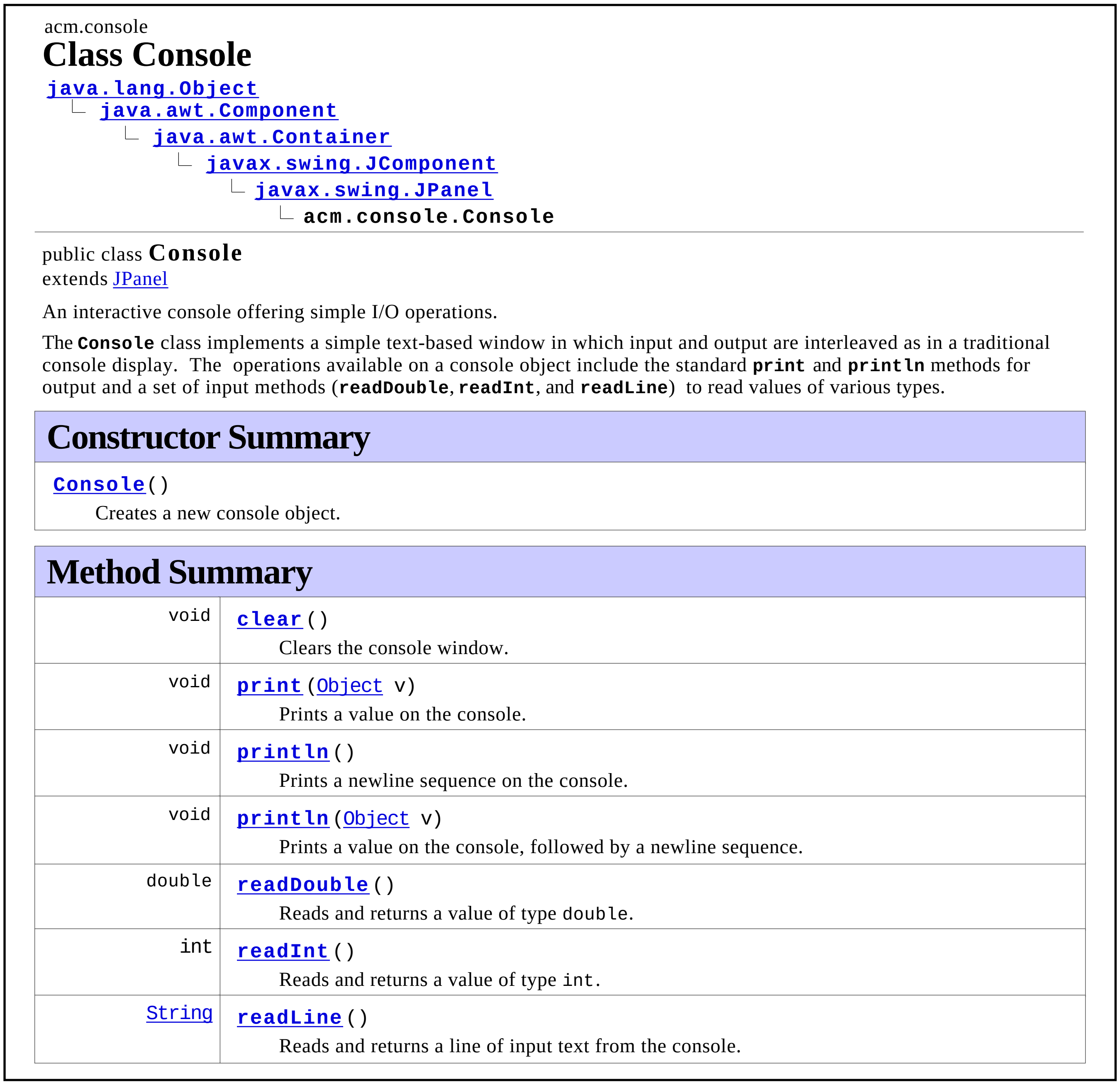
• *The existence of a console mechanism enables teachers without objet-oriented programming experience to teach Java much more effectively.* While it is perhaps correct in some idealized sense to argue that computer science teachers should make the transition to an objects-first approach, such a strategy is impractical given the skill sets, experience, and self-efficacy of teachers, particularly at the secondary school level. I believe it would be valuable for the Java Task Force to offer teachers a mechanism that falls within their domain of expertise, at least as an interim measure until object-oriented programming skills are more widely distributed.

This last point is further supported by the continuing controversy over whether an objects-first approach is really as workable as its proponents claim. I admit to feeling some sympathy with Stuart Reges's posting on SIGCSE-MEMBERS [7], in which he writes

> I have a challenge for the "objects early" experiment. I'd like to be convinced that a broad range of teachers can teach it well, that a broad range of students can master it, and that this approach solves more problems than it creates. Currently I'd say the experiment fails on all three fronts.

[†] In point of fact, students using my software packages rarely allocate their own **`Console`** objects but instead use the console window allocated automatically by the **`ConsoleProgram`** class, which is described in a separate submission on the **`Program`** class hierarchy. The two packages work well together, but are not necessarily linked. Each is useful in its own right.

Figure 1. Simplified user documentation for my implementation of the `Console` class



acm.console

Class Console

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ javax.swing.JComponent
              └ javax.swing.JPanel
                  └ acm.console.Console
```

public class **Console**
extends JPanel

An interactive console offering simple I/O operations.

The **`Console`** class implements a simple text-based window in which input and output are interleaved as in a traditional console display. The operations available on a console object include the standard **`print`** and **`println`** methods for output and a set of input methods (**`readDouble`**, **`readInt`**, and **`readLine`**) to read values of various types.

Constructor Summary

`Console()`
Creates a new console object.

Method Summary

`void`	`clear()` Clears the console window.
`void`	`print(Object v)` Prints a value on the console.
`void`	`println()` Prints a newline sequence on the console.
`void`	`println(Object v)` Prints a value on the console, followed by a newline sequence.
`double`	`readDouble()` Reads and returns a value of type `double`.
`int`	`readInt()` Reads and returns a value of type `int`.
`String`	`readLine()` Reads and returns a line of input text from the console.

While I am not nearly as pessimistic as Stuart, I can't dismiss his concerns out of hand. I believe that the jury is still out on the question of the objects-first approach, even though Java has become the overwhelming language of choice. I don't think it is wise for the Task Force to endorse a strategy that might not stand the test of time. Enabling teachers to use Java more effectively in an imperative style will give us a wider audience and provide some insurance against the possibility that objects-first fails to live up to expectations. Providing a standard **`Console`** class would go a long way toward supporting those who choose to adopt a more traditional approach.

References

1. David Eck. Introduction to Programming Using Java, fourth edition. **`http://math.hws.edu/javanotes/`**.
2. Bruce Eckel . Thinking in Java, third edition. Englewood Cliffs, NJ: Prentice Hall, 2002.
3. Scott Grissom. A pedagogical framework for introducing Java I/O in CS1. SIGCSE Bulletin, December 2000, pp. 57-59.
4. J. N. Patterson Hume and Christine Stephenson. Introduction to Programming in Java. Toronto: Holt Software Associates Inc., 2000.
5. Ron Poet. Java on the MscIT course at the University of Glasgow. Java in the Computing Curriculum 4, January 2000. **`http://www.ics.ltsn.ac.uk/pub/Jicc4/poet.doc`**.
6. Ira Pohl and Charlie McDowell. Java by Dissection. Boston: Addison-Wesley, 1999.
7. Stuart Reges. The new CS. Posting to SIGCSE-MEMBERS, March 23, 2004.
8. Eric Roberts. Taxonomy of problems in teaching Java. ACM Java Task Force, February 27, 2004.