Educational and technical design of a Web-based interactive tutorial on programming in Java

Claudia Bieg, Stephan Diehl∗

FR 6.2 Informatik, Saarland University, Germany

Received 3 September 2003; received in revised form 6 February 2004; accepted 17 February 2004
Available online 10 June 2004

Abstract

Most of the Java tutorials found on the Web are more or less well reformatted versions of lecture notes or textbooks. As a consequence these tutorials have just as little interactivity and adaptivity as the printed originals. The tutorial JOSH-online presented in this article enables students to learn Java programming step by step, by interactive trial and error. The tutorial not only provides exercises, but encourages students to consolidate and extend their newly acquired knowledge by means of own hypotheses and their experimental verification. In this paper we describe the design of the tutorial, the underlying interpreter and its integration.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Java; Interpreter; Programming languages; Learning

1. Introduction

JOSH is an interpreter for the Java programming language [8] originally designed to ease teaching Java to beginners. Recently the interpreter was restructured into a server-based interpreter applet and integrated into an online tutorial on Java programming called JOSH-online [4]. In the following we briefly introduce both systems. Then we elaborate on the pedagogical design of the tutorial. Finally we discuss some implementation details of both interpreters and give a comparison with other approaches. More technical details can be found in [4] and [7].

∗ Corresponding author.
E-mail address: diehl@acm.org (S. Diehl).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.02.003
1.1. Stand-alone and distributed Java interpreters

*JOSH* is a stand-alone interpreter for Java. The programmer and in particular Java novices can evaluate expressions, execute simple statements or declare variables and methods by simply entering the program text and pressing the return key. There is no need to define any classes.

In the following we use the term *simple code fragment* to denote simple expressions, statements or variable, method or class declarations. We use the term *code fragment* for a sequence of one or more simple code fragments. Whenever the user has completely entered a code fragment, *JOSH* evaluates or executes it. More precisely, after pressing the RETURN key *JOSH* checks whether the text entered so far is a code fragment, a prefix of a code fragment or something else. In the first case, the fragment is executed; in the second the user may enter more text to complete the fragment. In all other cases a syntax error is indicated and the input is ignored.

The first programs of Java novices often look as follows:

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

To understand the meaning of all the constructs in this program, they would have to know about classes, static methods, packages, strings, arrays etc. In *JOSH* they can simply start with the following statement:

```java
> println("Hello World");
Hello world
```

The input `println("Hello World");` is read and executed by the interpreter producing the text *Hello World* as feedback. The following session exemplifies some other features of *JOSH*:

```java
> 3 + 'c'
102 : int
> class A { int x; void setX(int n) {x=n;}; }
class generated
> A a = new A();
Field added
> a.setX(9);
> a.x
9 : int
```

A special feature of *JOSH* is that the evaluation of expressions not only shows the value but also the type of the result (see also Fig. 1), thus helping the students to better understand the type system and type conversion in Java.
1.2. JOSH-online

When we used JOSH in programming courses and encouraged students at home, we found that installation was often an issue. First students had to install the Java development environment JDK; second they had to set the environment variable CLASSPATH as well as some paths in the configuration file of JOSH correctly. To facilitate the access to JOSH as well as to make it accessible to a wider audience, we developed JOSH-online. JOSH-online integrates the interpreter into an interactive, Web-based tutorial on programming in Java. Fig. 1 shows the definition and execution of the method to compute factorial numbers with the help of the interpreter integrated into the tutorial.

2. The tutorial

The tutorial starts from first principles such as simple data types and variables, followed by method declarations and finally classes. Every unit of the tutorial ends with examples and exercises which can be completed by means of the integrated interpreter. A novel feature of the tutorial is the communication between the text parts and interpreter applet. Every source-code example in the text can be directly tested by clicking at an icon shown next to it. The source code is automatically inserted into the text field of the applet.
There the user can edit it before it is executed by the interpreter. Some exercises rely on those of previous sections or chapters. The interpreter maintains the state of previous exercises, i.e. those variables, methods and classes that have been defined before. In the course of the tutorial the student is encouraged to think of their own examples and test those with the interpreter. For this, the tutorial makes suggestions as to what aspects of the programming language to explore.

2.1. Pedagogical design

Computer-based learning systems have met criticism (see [3]). For example, some parts of complex systems distract from the learning content (e.g. cover stories) or make it difficult to use. We tried to design JOSH-online to be user-friendly by relying on a functional design and on a clear system structure.

The tutorial covers basic knowledge about programming in Java. There are seven learning units. In each unit there are two goals: provision and consolidation of knowledge. Therefore each learning unit consists of the following instruction sequence: first all information is presented, then questions about the presented content are posed. The interpreter analyses and evaluates these answers. If the input is incorrect or the answer is wrong the user has the chance to reconsider and correct it. Otherwise the user can proceed to the next learning unit. The examples in the learning units are designed in such a way that they can be easily tested by the user with the built-in interpreter giving feedback on the user input. The exercises are designed in the same way. The tutorial gives hints to solve the exercises. The user enters the answer in the interpreter and starts the evaluation (Fig. 2). If the user was not able to find the correct answer after several trials, the tutorial offers a sample solution.

A big advantage of the interpreter is its flexibility. The users are not tied to the examples or exercises given. They are encouraged to invent new examples on their own. Gaps in the users’ knowledge can be closed by their own experiments. We believe in the old saying that ‘The proof of the pudding is in the eating’; consequently the tutorial offers the possibility to experiment actively to the user. JOSH transits from Drill & Practice to explorative
learning: that means learning based on the scientific method of **Hypothesize & Test**. Instead of fixed exercises it is possible to execute as many (guided) experiments as the user likes. The results should be reflected and put into relation by the user. Further tests can consolidate these results and the user will be able to realize the underlying abstract concepts. **JOSH** thus supports the **learning cycle** (Fig. 3) introduced by David Kolb [11]:

1. **Experiencing.** The users immerse themselves in the task. In **JOSH-online** the tutorial gives hints. Examples are presented with the intention to lead the user to an assumption. In this stage of experiencing we integrate the **Drill & Practice** scheme shown in Fig. 2.

2. **Reflection.** The user is involved in stepping back from the original task. In this stage the aim is to gain experience from the given results.

3. **Conceptualization.** The users are involved in the interpretation of their inputs. In a second step they are to put the results into a bigger context. Thus new concepts are gained. Theoretical aspects given by the tutorial help to explain the results of the experiments.

4. **Planning.** The results archived are used to predict what is likely to happen in the next tests. In this way new insight can be gained and extended.

**JOSH-online** realizes the circular flow in Fig. 3 by encouraging the users to come up with their own experiments. Knowledge gained in the tutorial is related to the experimental results archived in the stage of reflection finally leading to the formulation of new concepts. If it should be necessary, arbitrarily new experiments can be conducted. The interpreter approves or disproves the results expected by the user after the stage of planning.

### 2.2. Classification as a learning environment

Tutorial systems are described as **linearly organized programs with a high degree of system control** [6]. Usually the learner pursues the sequences given. Interactivity is not supported. By the following features, **JOSH-online** is extended from a tutorial to a more complex system.
Multimedia. The contents are visualized as in a book and completed with many figures. The design of JOSH-online is simple due to clarity and simplicity of use.

Adaptiveness. Learning content is offered to the user who can decide the order of processing. If it is necessary the user is led to the right way of learning. Advanced learners can start with later chapters. In each chapter the examples and exercises already processed are marked. The exercises can be solved with the aid of the tutorial. The number of tests is unlimited. If the user is not able to find a suitable solution, the tutorial offers one.

Interactivity. The integrated interpreter supports the communication between user and system. The test input is executed in the interpreter and the results are given to the user. In the case of a correct input the result of the evaluation is shown; in the case of an incorrect input the interpreter points to the source of error.

The design of JOSH-online is simple but versatile. The core is the interactivity of the system: interpreter and user conduct a conversation. At the moment, JOSH-online is passive adaptive. But the construction of the system allows us to expand it to an active adaptive system as described in Section 2.3.

2.3. Possible extensions

JOSH-online’s implementation is based on a client–server architecture. The evaluation of code fragments is done by the server where the user profile is also managed. It would be possible to make use of this fact to gain a more adaptive design of the system. At the moment the data is reliable as long as the user session lasts. By introducing a user name a persistent user management could be established. In this case it would be possible to save information such as the user’s learning progress or the learning units already processed. Moreover the level of difficulty of the exercises could be adapted to the user’s knowledge.

JOSH-online was developed for use in the intranets of schools, e.g. for tests or examinations. All user input goes to the server. So teachers have the opportunity to check the input of the pupils and also to determine the learning progress. Examinations could be done online. To control the time spent by the students on each question the lifetime of the applet could be limited. Some of these ideas are part of our future work.

3. Implementation details

The JOSH-online interpreter was implemented as a front-end for the Java compiler [4,7]. Thereby we can guarantee that the Java syntax and semantics are preserved. The implementation of our interpreters is totally different from those of other Java interpreters.

3.1. Implementation of the stand-alone interpreter

JOSH inserts complete code fragments into a class skeleton to produce a running Java application. For example, if the user enters `int x=0;` the following Java source code is generated:
package tempclasses;

public class InterpreterAux0
    extends jinterpreter.InterpreterBase{
    public static void main(String [] argv) {
        InterpreterAux0 self=new InterpreterAux0();
        self.startInterpreterAux();
    }

    public void startInterpreterAux(){
        println("Field added");
        dump("/tmp/tempclasses/state.dump");
    }

    int x=0;
}

The source code is compiled by starting the Java compiler as an external process. If the compilation is successful, then the generated byte code is executed by the Java Virtual Machine (JVM) as an external process. As we generate new applications for each code fragment and execute these applications as individual processes, the problem arises of how we preserve state, i.e. how can we start the next application with the final state of the previous application. By state we mean the set of instantiated objects and their attribute values at runtime. In addition the next application has to know all the methods and classes of the previous one. To achieve this the next application inherits from the previous one. In addition, at the end of the execution of an application the currently instantiated objects are serialized and stored in a file; see method call dump() above. Continuing our previous example, if the user now enters the expression x++, JOSH generates the following source code:

package tempclasses;

public class InterpreterAux1
    extends InterpreterAux0{
    public static void main(String [] argv) {
        InterpreterAux1 self=new InterpreterAux1();
        InterpreterAux0 previousState= (InterpreterAux0)
            undump("/tmp/tempclasses/state.dump");
        self.x=previousState.x;
        self.startInterpreterAux();
    }

    public void startInterpreterAux(){
        javaInterpreterEvaluate(x++);
        dump("/tmp/tempclasses/state.dump");
    }
}

Now the class InterpreterAuxm can access a serialized object of its superclass by calling the method undump() and setting the values of those variables that are not redefined
in the class `InterpreterAux` to the stored values. If the execution of a successfully compiled code fragments leads to a runtime error (or an infinite loop), the state can be reset to that of the previous state. Thus serialization enables us to go backwards in the state history.

### 3.2. Implementation of the distributed interpreter

In making `JOSH` available online, there is no immediate way to turn it into an applet, because this would require the applet to have access to the Java compiler of the client. Instead we separated different phases of the stand-alone interpreter such that some are executed on the server and some by the applet. In particular, code fragments are compiled on the server, so the Java compiler has only to be available there.

In the `JOSH-online` applet the user enters code fragments in a text area. After clicking a button the code fragment is sent to the server; the server parses the fragment. If it is complete, the fragment is compiled in much as before. If the compilation was successful, the client dynamically loads the newly generated class from the server and executes its main method. All output produced by the compiled code fragment is redirect to the text area of the applet. `JOSH-online` also uses inheritance to preserve the state, but instead of the previous state being serialized and stored in a file, the state is stored in a class variable and kept in the client’s memory. Thus instead of calling the method `undump()`, an instance of the parent class is accessed; e.g.,

```java
InterpreterAux0 previousState= 
    InterpreterAux0.self;
```

Note, that this method does not allow recovery from all kinds of runtime errors like the one used by the stand-alone interpreter.

In addition to preserving the state, `JOSH-online` has also to keep track of different users that concurrently access the server. For each user a directory with a new and unique name and thus a new package with the same name are created. In addition, for every user the server has to keep track of all variables defined so far. This is not necessary for user-defined classes and methods, as their “values” do not change.

### 4. Related work

Recently several interpreters for Java have been developed including MiniJava [13], DynamicJava [10] and BeanShell [12]. While MiniJava is not publicly available and there are no details available on its implementation, both DynamicJava and BeanShell are written in Java. Both are open source. They build an abstract syntax tree and then traverse the nodes of these trees and execute their semantics functions. JIN [15] is a commercial Java interpreter also written in Java that presumably works in a similar way. While DynamicJava supports class and method declarations, BeanShell only supports method declarations and JIN supports neither.

DrJava [1,2] is a Java programming environment that uses DynamicJava within its interaction window. For classes defined in other windows it calls the Java compiler.
So far, the above interpreters have not been integrated into online tutorials. The interactive online programming tutorials that we found on the net have been developed for LISP [9], Ruby [14] and NESL [5] and are purely text based, i.e. the user enters the program code into a Web form; then the code is sent to the server which executes the program code and returns the result as text to be displayed by the client. In contrast, JOSH-online executes the program code on the client and preserves the program state. As a consequence exercises can rely on the results and side-effects of previous exercises and APIs such as the Abstract Windowing Toolkit can be used on the client (Fig. 4):

```java
    > java.awt.Frame f
        = new java.awt.Frame("My first window");
    >>
    > f.setSize(300,100);
    >>
    > f.setVisible(true);
    >>
    > f.getGraphics().drawString("Hello World",20,70);
    >>
```

5. Comparison of three interpreter architectures

In Table 1 we compare three different architectures:

**DynamicJava.** Classical interpreter architecture which builds an abstract syntax tree for each fragment and performs semantics actions while traversing the tree.

**JOSH.** Java classes are generated for each fragment, compiled externally and executed.

**JOSH-online.** Java classes are generated for each fragment, compiled externally (on the server), dynamically loaded and executed.

Although the implementation of DynamicJava that is used in DrJava seems to be the most elaborate classical Java interpreter available, we found that it does not implement resolution of overloaded methods correctly; in particular, it did not detect ambiguous method invocations.

When it comes to performance, the generation-based approaches suffer from an initial generation overhead. On a 700 MHz Pentium III, generation takes about 1 s. For method invocations we found that the interpretation overhead of DynamicJava was higher by about a factor of 600 as compared to JOSH and for memory access (reading and writing to
### Table 1
Comparison of three interpreters

<table>
<thead>
<tr>
<th></th>
<th>DynamicJava</th>
<th>JOSH</th>
<th>JOSH-online</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error recovery</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Preserving transient state information</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Semantics</td>
<td>Hand coded</td>
<td>Relies on compiler and JVM</td>
<td>Relies on compiler and JVM</td>
</tr>
<tr>
<td>Size of implementation (including parser)</td>
<td>28 (45) KLOC without GUI</td>
<td>1 (7) KLOC</td>
<td>3 (10) KLOC without GUI</td>
</tr>
<tr>
<td>Performance</td>
<td>No generation overhead, interpretation overhead during execution</td>
<td>Generation overhead, fast execution of compiled code</td>
<td>Generation overhead, fast execution of compiled code</td>
</tr>
</tbody>
</table>

![Array Access Diagram](image_url)

Fig. 5. Performance for accessing arrays.

an array) the factor was about 45. Some performance results (including those for the BeanShell interpreter) are shown in Figs. 5 and 6.

Because it relies on the compiler and the JVM, the size of the source code of JOSH’s implementation is only 4% of that for DynamicJava. The implementation of JOSH is simple in the sense that we do not have to implement the semantics of each Java construct.
We could even use the Java grammar provided with JavaCC without any modifications. We only had to invoke some internals of the generated parser in addition to the standard way of using it to be able to detect valid prefixes and extract class, method and variable names from declarations. The non-trivial problem to solve was preserving the state.

In the file josh.properties the user can actually configure what compiler, APIs and JVM are used by JOSH for external compilation and execution:

```plaintext
classpath = c:\JOSH\josh0.02.jar;
java.command = c:\programs\java\jdk1.4\bin\java
javac.command = c:\programs\java\jdk1.4\bin\javac
```

As JOSH compiles fragments into byte code and then executes this byte code on the JVM, it is even possible to use the Java Debugger instead of the standard JVM in combination with JOSH. If we try this with a classical interpreter the whole interpreter would have to be executed by the debugger and we would find ourselves debugging the meta-level and not our actual program.

6. Conclusion

There are many reasons for choosing Java as a first programming language at schools and universities, including pedagogical aspects such as its present relevance and the resulting motivation, as well as technical ones such as the variety of APIs. Unfortunately the first programming tasks are complicated by the fact that one has to define complete classes. Java interpreters such as JOSH relieve the programmer of these initial difficulties by executing program fragments directly. Integrating such an interpreter into a tutorial on programming enables the student to immediately test examples in the text and to work
on programming exercises directly in the same browser window without any need to start external programming tools. Furthermore, the tutorial encourages the students to come up with their own hypotheses and verify these by programming experiments.

References