A new Approach for Implementing stand-alone and web-based Interpreters for Java

Stephan Diehl
FR 6.2 Informatik
Saarland University
66041 Saarbrücken, Germany
diehl@acm.com

Claudia Bieg
FR 6.2 Informatik
Saarland University
66041 Saarbrücken, Germany
cbieg@cs.uni-sb.de

ABSTRACT

Compared to imperative programming languages like Pascal or Basic, or functional programming languages like ML or Haskell learning Java is hampered by the fact that to get even the simplest running program the learner has to define a public class and a method with a certain signature. We present both a stand-alone and a web-based interpreter which execute Java fragments and relieve the learner from programming all the extra code. The implementation of these interpreters extremely differs from other Java interpreters and exploits the Java compiler as much as possible to preserve the original semantics of Java and allow access to all features and APIs of Java. By virtue of these interpreters the learner can explore primitive values, variables, expressions, assignments, and control-flow statements before even knowing about classes and methods. The web-based interpreter has been integrated into an online tutorial for learning Java programming from basic principles.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*interpreters, compilers*

Keywords

Java, interpreter

1. INTRODUCTION

Right from the start Sun was marketing Java as a simple, object-oriented programming language [3]. But object-orientation hinders to learn Java step-by-step from basic principles, because right from the beginning the learner has to define at least one public class with a method with signature `public static void main( java.lang.String[])`.

So the teacher has two choices here: trying to explain most of the concepts involved (classes, methods, types, arrays,...) or just provide the surrounding program text and let the learner add code to the body of the method `main`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2003 Kilkenny City, Ireland
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Instead we suggest to use Java interpreters that allow to expose the learner just to those concepts of the language that he can understand at that time and enables the learner to experiment with these concepts. Thus the learner can start with primitive values, expressions, then proceed to statements and finally define methods. Not until these basic concepts are understood we continue with classes and inheritance.

From a technical point of view the approach presented in this paper enables the interpreter to access all Java APIs, to plug-in a Java compiler of ones choice and to support multiple users over the same server.

1.1 *JOSH*

JOSH is a stand-alone Java interpreter which enables the learner to evaluate expressions, to execute simple statements, to declare variables and to define methods without a need to define classes.

In the following we use the term *simple code fragment* to denote single expressions and statements, or declarations of methods, variables and classes. The term *code fragment* is a sequence of one or more simple code fragments. As soon as the user has entered a code fragment *JOSH* starts evaluating or executing it. More precisely when the user presses the RETURN key, *JOSH* checks whether the input so far is a code fragment, a prefix of a code fragment or otherwise. In the first case the fragment is executed, in the second case *JOSH* waits for additional user input to be appended to the current input to complete the fragment. In all other cases the current input can not be completed to form a fragment. As a consequence there a syntax error is reported and the input buffer is emptied.

1.2 *JOSH-online*

When using *JOSH* in class we encountered the several problems. One was that the JDK had to be installed and the CLASSPATH and several paths in the configuration file had to be set correctly. To facilitate access to *JOSH* such that everyone with an Java-enabled web browser can use it, we developed *JOSH-online* and integrated it into an interactive, web-based tutorial on programming in Java.

2. RELATED WORK

Recently several interpreters for Java have been developed including MiniJava [6], DynamicJava [4] and BeanShell [5]. While MiniJava is not publically 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 [7] is a commercial Java interpreter also written in Java that presumably works in a similar way. While Dy-

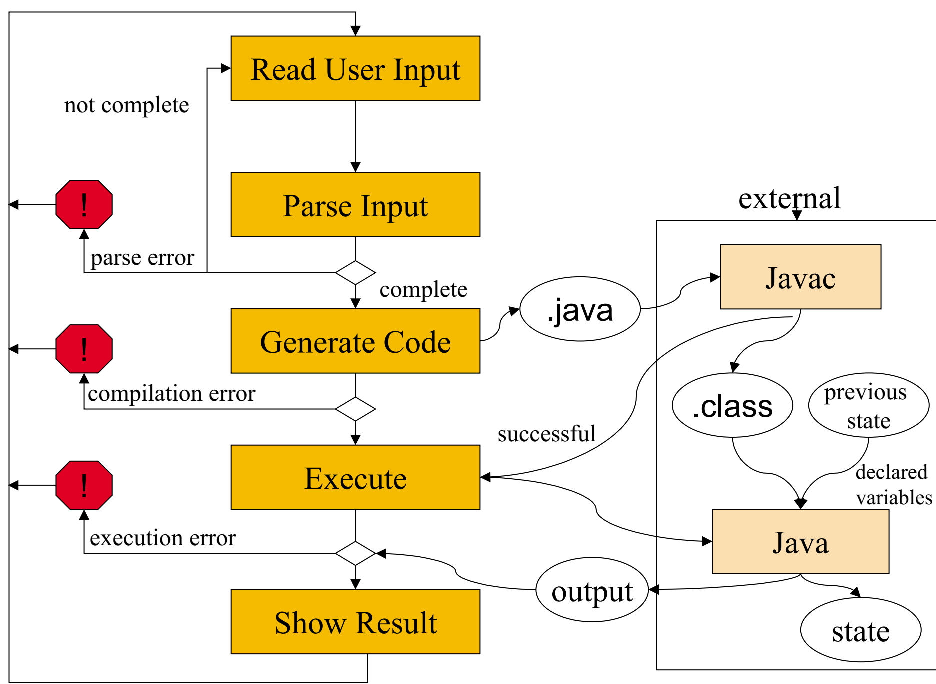

Figure 1: Processing of code fragments in *JOSH*

namic Java supports class and method declarations, BeanShell only supports method declarations and JIN supports neither of both.

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.

3. IMPLEMENTATION OF THE STAND-ALONE INTERPRETER

JOSH inserts complete code fragments are inserted 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 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 `InterpreterAux`m 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`m 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.

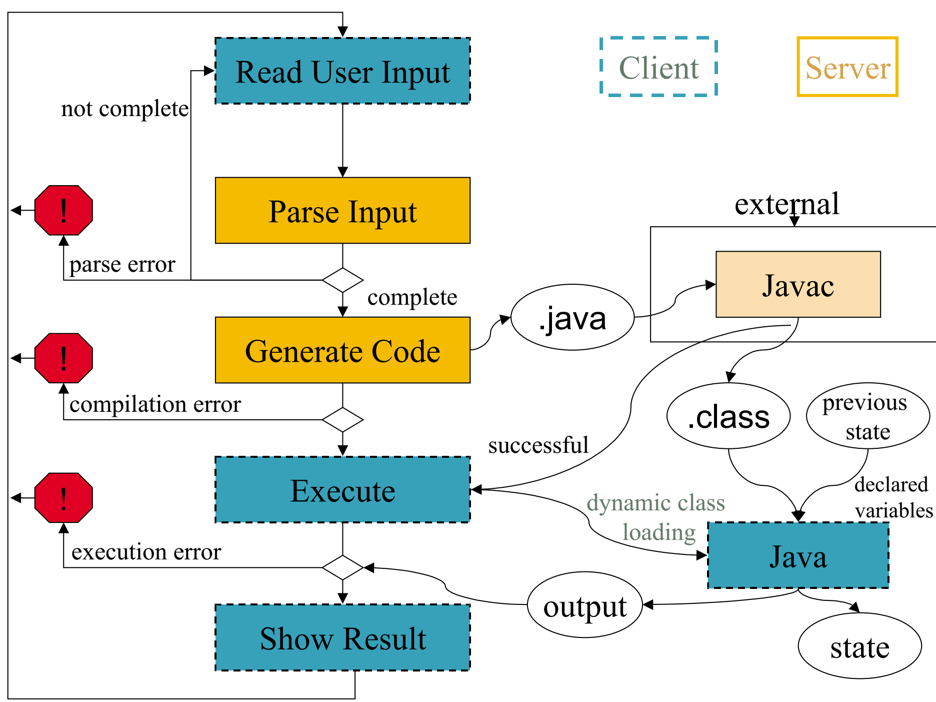

Figure 2: Processing of code fragments in *JOSH-online*

4. IMPLEMENTATION OF THE DISTRIBUTED INTERPRETER

To make *JOSH* available online there is no immediate way to turn it into an applet, because it would require that the applet would have access to the Java compiler on 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 that 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 at a button the code fragment is sent to the server, the server parses the fragment. If it is complete the fragment is compiled in a similar way 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 serializing the previous state and storing it in a file, the state is stored in a class variable and kept in the clients memory. Thus instead of calling the method `undump()` an instance of the parent class is accessed e.g.:

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

Note, that this method does not allow to recover from all kind of runtime errors as 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.

5. THE ONLINE JAVA TUTORIAL

JOSH-online was integrated into an online tutorial on programming in Java. The tutorial follows the natural way from basic concepts like primitive data types and variables to method and class abstraction. For expressions *JOSH* not only returns the value, but also their types. We found this very useful for helping the students to understand the type system and type conversion in Java. Every unit of the tutorial ends with a set of examples and exercises that can be interactively done in the text book. The most interesting aspect of the tutorial is the communication between the text book and the interpreter applet. Every source code example in the text book can be immediately tested by clicking at a button that appears next to it. The source code is then automatically inserted into the text area of the applet and the learner can edit and execute it. Some of the exercises rely on previous ones, ie. that variables have the previously assigned value or defined methods or classes are still known. Throughout the tutorial the learner is encouraged to come up with own examples and test them interactively in the interpreter. To this end the tutorial provides helpful suggestions to explore the features of the Java language.

The text of the tutorial was developed for two courses that the first author gave at the Universities of Duisburg and Saarbrücken. In these courses the students were using the stand-alone interpreter.

6. CONCLUSION

JOSH and *JOSH-online* are basically front-ends to whatever Java compiler you want to use. Compared to DynamicJava and thus DrJava we found that *JOSH* executes code fragments more than 100 times faster[1]. For typical code fragments the external compilation takes less than 1 second on modern hardware and thus the response time is acceptable for interactively work with the interpreter. Also

[1]For example the loop `for(int i=0;i<10000000;i++);` took 3 seconds in *JOSH* and 297 seconds in DrJava.

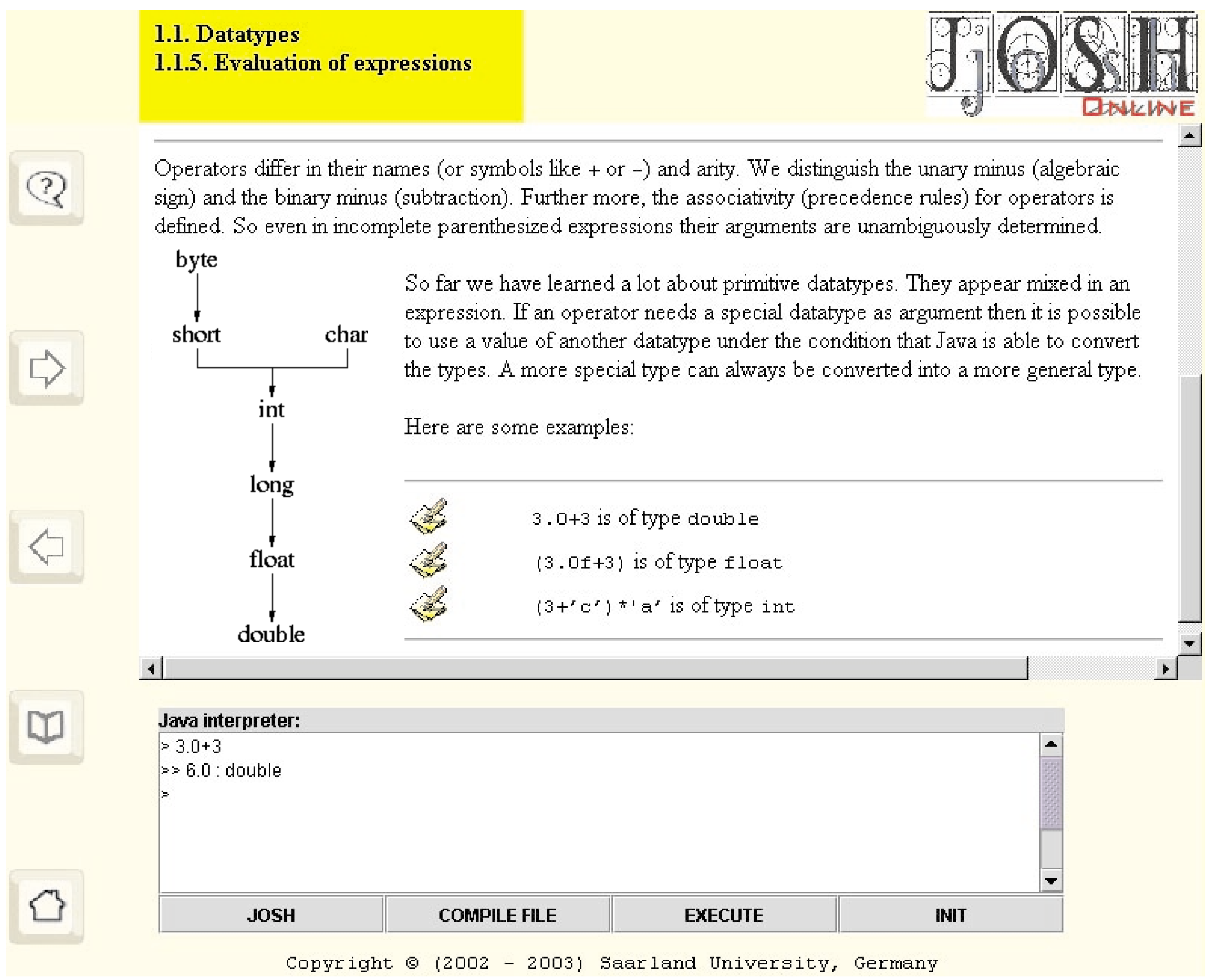

Figure 3: Screenshot of the tutorial with the *JOSH-online* applet

JOSH has no restrictions with respect to the APIs that can be used and it is even possible to use the Java Debugger in combination with *JOSH*. The *JOSH* approach can be summarized as follows: First compile code fragments externally, then execute them externally or by dynamic class loading while preserving state.

We presented two approaches to preserve state: one based on serialization (*JOSH*), while the other keeps copies in the clients memory (*JOSH-online*). We expect that both the stand-alone and the distributed method presented in this paper are also useful in other contexts to extend or re-compile parts of a running system and selectively preserve state.

7. REFERENCES

[1] E. Allen, R. Cartwright, and B. Stoler. DrJava. `http://drjava.sourceforge.net/`.

[2] E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *Proceedings of ACM SIGCSE 2002*. ACM SIGCSE Bulletin 34(1), 2002.

[3] J. Gosling and H. McGilton. The Java Language Environment: a White Paper, 1995.

[4] S. Hillion. DynamicJava. `http://koala.ilog.fr/djava/`.

[5] P. Niemeyer. BeanShell. `http://www.beanshell.org/`.

[6] E. Roberts. An overview of MiniJava. In *Proceedings of ACM SIGCSE 2001*. ACM SIGCSE Bulletin 33(1), 2001.

[7] L. Vanhelsuwé. JIN. `http://www.lv2.clara.co.uk/products/Jin.html`.