Jacl: A Tcl Implementation in Java

Ioi K. Lam, Brian Smith
Department of Computer Science
Cornell University
Ithaca, NY

```java
import cornell.Tcl.*

class PrintCmd implements Command {
    Object CmdProc(Interp interp, CmdArgs ca) throws EvalException {
        if (ca.argc != 2) {
            throw new EvalException("wrong # args: should be \"" + ca.argv(0)
                    + " string\"");
        }
        System.out.println(ca.argv(1));
        return "";
    }
}

....
    // Create a new "print" command.
    interp.CreateCommand("print", new PrintCmd());
....
```

Example (4.1)

```java
class Index1Cmd implements Command {
    Object CmdProc(Interp interp, CmdArgs ca) throws EvalException {
        if (ca.argc != 2) {
            throw new EvalException("wrong # args: should be \"" + ca.argv(0)
                    + " vector\"");
        }
        if (!ca.object(1) instanceof Vector) {
            throw new EvalException("expected Vector but got \"" + ca.argv(1)
                    + "\"");
        }
        Vector vector = (Vector)(ca.object(1));
        if (vector.elementCount < 1) {
            throw new EvalException("Vector must not be empty");
        }
        return vector.elementAt(0);
    }
}

....
    // Create a new "index1" command.
    interp.CreateCommand("index1", new Index1Cmd());
....
```

Example (4.2)

Bibliography

[1] John Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Massachusetts, 1994

[2] Ken Arnold, James Gosling, *The Java Programming Language*, Addison-Wesley, Massachusetts, 1996

[3] Jacob Levy , *A Tcl/Tk Netscape Plugin*, Proc. of the 1996 USENIX Tcl Workshop, Monterey, 1996.

[4] Scott Stanton and Ken Corey, TclJava: Toward Portable Extensions, Proc. of the USENIX 1996 Tcl/Tk Workshop, Monterey, 1996.

[5] Michael Morrison, et al., *Java Unleashed*, Sams.net Publishing, Indianapolis, 1997

[6] Brian Lewis, *An On-the-fly Bytecode Compiler for Tcl*. Proc. of the USENIX 1996 Tcl/Tk Workshop, Monterey, 1996.

[7] John Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, http://www.sunlabs.com/people/john.ousterhout/scripting.html, 1997.

[8] Sun Microsystems, Inc., *Java™ Core Reflection, API and Specification*, http://java.sun.com/products/jdk/1.1/docs/guide/reflection/, 1997.

such as `lappend`, is applied to that variable.

3.2. Exception Handling

Another difference between Tcl 7.6 and Jacl is how they handle error conditions. Tcl 7.6 uses return code such as `TCL_OK` and `TCL_ERROR` to indicate the success or failure of script execution. The Tcl 7.6 C source code spends considerable efforts in checking the return code of functions. In contrast, Jacl uses the Java exception mechanism to handle runtime errors. Thus, the Jacl source code is less cumbersome than the Tcl 7.6 C source code. For example, inside the Tcl parser, where errors can happen in many sections of the code, the Jacl implementation uses about 30% fewer lines of code than the Tcl 7.6 implementation written in C. Figure 3.1 compares the coding style between Jacl and Tcl 7.6

4. Writing Jacl Extensions

A Jacl extension is generally a collection of new Tcl commands. A Tcl command is a class that implements the `Command` interface. The command can be added to a Jacl interpreter by passing an instance of its class to the `CreateCommand` method. Example 4.1 shows how a `print` command can be defined

One interesting feature of Example 4.1 is the way arguments are passed to `CmdProc`, the command procedure. Because the arguments passed to a command may be Java objects of any type, it is no longer sufficient to pass the arguments as `(int argc, char ** argv)` in Tcl 7.6. Instead, Jacl passes the arguments in a `CmdArgs` object. The following code shows the interface of the `CmdArgs` class:

```
public class CmdArgs {
    public int argc;
    public String argv(int index);
    public int intArg(int index);
    public double doubleArg(int index);
    ....
    public Object object(int index);
}
```

A command can use the converter methods, such as `argv`, `intArg` and `doubleArg`, to convert the arguments into the required types. The command can also use the Java `instanceof` operator to directly infer type information of the arguments. In example 4.2, the `index1` command verifies that it receives an non-empty Vector object as its first argument before returning the first element of this Vector.

5. Status and Future Directions

As of this writing, the Tcl parser, expression evaluator and most basic Tcl commands have been implemented in Jacl. It also supports a subset of the Tk commands for creating graphical interfaces. Jacl is already being used to create simple applets to run inside browsers. It can also be used to control Java applications and applets with raw- and custom scripting. A beta release is expected to be available in the third or fourth quarter of this year.

Many more features have been planned for Jacl, including built-in debugging, supports for multithreading, and a byte-code compiler. To find out more about the new developments of Jacl, please visit the Jacl home page at http://www.cs.cornell.edu/ home/ioi/Jacl.

Acknowledgment

I would like to thank Thomas Breuel and Anil Nair for providing valuable inputs during the early design stage of Jacl. Thomas sent me the basic design of the `cornell.Tcl.Command` interface, which I put into Jacl without much change. Scott Stanton and Jacob Levy were instrumental in the design of the raw scripting API.

Java code	C equivalent

<pre>...
int i = interp.GetInt(string);
  // exception is thrown if string
  // doesn't contain a valid integer
...</pre>
	<pre>...
	int i;
	if (Tcl_GetInt(interp, string, &i) != TCL_OK) {
	        return TCL_ERROR;
	}
	...</pre>

Figure (3.1)

2.4 Custom Scripting

In custom scripting, Jacl scripts access Java objects through a scripting API provided by a Jacl extension (see section 4 for a discussion on writing Jacl extensions.) The `button` command in section 2.2 is an example of a custom scripting API for accessing java.awt.Button objects. One can access Java objects through raw scripting or custom scripting. Figure 2.3 shows the differences between raw- and custom scripting and compares the scripting code with Java code.

As shown in figure 2.3, both raw- and custom scripting provides interactive access to Java classes. Custom scripting has the advantage of supporting a more convenient syntax but it requires the writing of Jacl extensions. Therefore, raw scripting is generally used to gain "quick and dirty" access to Java objects. When it is necessary to have better scripting support for Java objects, Jacl extensions can be written to provide a custom scripting API.

3. Implementation of the Jacl Interpreter

The Jacl interpreter is based on the Tcl 7.6 interpreter. Most of the parsing routines for Tcl scripts and expressions are translations of the Tcl 7.6 C source code into Java code. Therefore, the Jacl interpreter is compatible with the Tcl 7.6 interpreter. In fact, the Tcl 7.6 test suite is used to ensure that Jacl parses and executes scripts in exactly the same manner as Tcl 7.6.

There are two major enhancements in Jacl with respect to Tcl 7.6: object support and exception handling. These enhancements improve efficiency and simplify the implementation of the Jacl interpreter and extensions.

3.1 Object Support

In Tcl 7.6, all objects are represented by strings. In Jacl, however, an object can be represented by any Java object. For example, in the following code:

```
set a 1234
incr a
```

After the first line, the variable `a` will contain a string "1234". At the second line, the `incr` command will coerce the string into an integer and then increment its value by one. After this operation, the variable `a` will contain a integer with the value 1235.

Moreover, lists in Jacl are implemented as copy-on-write Vector objects to improve both access time and storage efficiency. In the following code

```
set list1 [list 1 2 ... n]
set c [lindex $list 3]
set list2 $list1
...
...
lappend list2 abc
```

the `lindex` operation takes constant time, compared to the O(n) time in Tcl 7.6. Also, after the `set list2 $list1` command, the two variables `list1` and `list2` will refer to the same object. The contents of the list will be copied into the `list2` variable only when a destructive operation,

Coding Method	Program Listing	Interactivity	Simple Syntax	Extension Required

Java Code	`Button b = new Button("Hello");`
	`Color c = new Color(255, 255, 0);`
	`b.setForeground(c)`
	`add(b);`
		No	No	--

Raw Scripting	`set b [new Button Hello]`
	`set c [new Color 255 255 0]`
	`$b setForeground $c`
	`$applet add $b`
		Yes	No	No

Custom Scripting	`button .b -text Hello -fg #ffff00` `pack .b`	Yes	Yes	Yes

Figure (2.3)

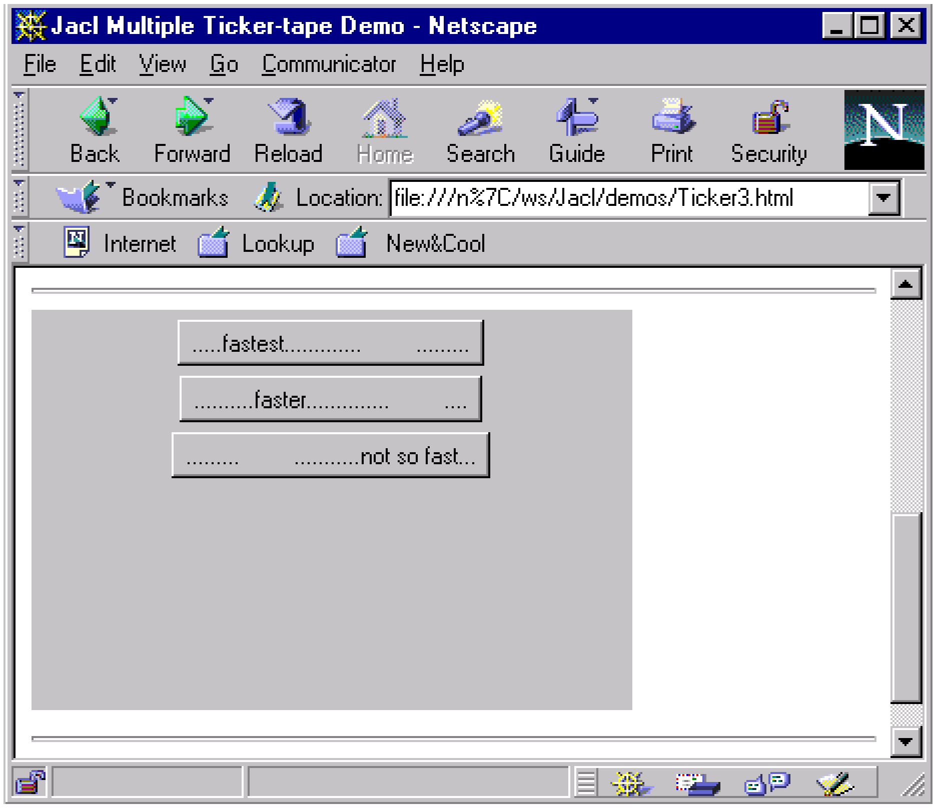

Figure (2.2)

When the methods and fields of the Java objects are invoked, Jacl will coerce the parameters when necessary. For example, in the following code segment, the parameters passed to the `setSize` method of the frame object may be represented as strings in the script. Jacl will convert them into integers before invoking the `setSize` method:

```
set frame [new java.awt.Frame]
$frame setSize 100 200
```

Jacl uses a set of heuristics to disambiguate the invocation of overloaded methods. For example, if we have a Java class with an overloaded method `foo` that can take either an integer or a string parameter:

```
class A {
    void foo(int i);
    void foo(String s);
}
```

and we manipulate this class with the following script:

```
set obj [new A]
$obj foo 1
$obj foo abcd
```

The first call to `foo` will invoke the integer version because the parameter looks like an integer. In contrast, the second call will invoke the string version because it is not possible to convert `abcd` to an integer.

In the cases where the disambiguation heuristics are insufficient, one can use *method signatures* to choose which version of an overloaded method should be called. A method signature specifies the name and argument types of a method. For example, the following code forces the string version of the `foo` method to be called even though the argument looks like an integer:

```
set obj [new A]
$obj {foo String} 1
```

```
button .b1
button .b2
button .b3

.b1 config -text "   ..............fastest............      "
.b2 config -text "   ..............faster.............      "
.b3 config -text "   ...........not so fast...........      "

pack .b1 .b2 .b3

proc scroll {btn time} {
    set str [$btn cget -text]
    set str [string range $str 1 end][string index $str 0]
    $btn config -text $str
    after $time scroll $btn $time
}

scroll .b1 100
scroll .b2 200
scroll .b3 500
```

Example (2.1)

2.1 Using Jacl in an Applet

The Java classes that implement Jacl are in the cornell.* hierarchy. The conell.applet.Shell class can be used to execute Jacl scripts inside applets. The following HTML code shows how to embed an Jacl-enabled applet inside an HTML page:

```
<applet width=300 height=100>
  code=cornell.applet.Shell.class>
  <param NAME="jacl.script"
         VALUE="buttons.tcl">
</applet>
```

When the cornell.applet.Shell class starts up, it will create a Jacl interpreter to execute the script file specified by the jacl.script parameter.

2.2 Using Tcl and Tk Commands

Jacl supports all the basic Tcl commands (e.g., `string` and `puts`, as well as the control constructs such as `if` and `for`.) It also supports a subset of the Tk commands for building graphical interfaces. Example 2.1 shows a script that performs a simple animation by scrolling text across three buttons at different speed. This script should look familiar to experienced Tcl/Tk programmers because its syntax is exactly the same as traditional Tcl/Tk programs. Figure 2.2 shows how the applet appears inside Netscape.

2.3 Accessing Java Classes with Raw Scripting

There are two ways for Jacl scripts to access Java classes: *Raw Scripting* and *Custom Scripting*. Raw scripting uses the Java Reflection API [8] to directly create Java classes and invoke their methods and fields. The following example shows how an applet can use raw scripting to manipulate a java.lang.Date object:

```
set date [new java.lang.Date]
button .day -text [$date getDay]
```

The `new` command is used to create an instance of a Java class with the given name (in this case, java.lang.Date.) The `new` command returns an *object command*, which can be used to invoke the methods of the object. In the above example, the `getDay` method of the object is called to query the current day of the week on the system.

The object command supports two special options, `get` and `set`, to query and modify the fields of the Java object. In the following example, we create an object of the java.util.Vector class, add several elements and the query the `elementCount` field to determine the number of elements in the vector object:

```
set vector [new java.lang.Vector]
$vector addElement “string1”
$vector addElement “string2”
set num [$vector get elementCount]
```

Jacl: A Tcl Implementation in Java

Ioi K. Lam, Brian Smith
Department of Computer Science
4130 Upson Hall
Cornell University
Ithaca, NY 14853-7501

{ioi, bsmith}@cs.cornell.edu

Abstract

Jacl, Java Command Language, is a version of the Tcl [1] scripting language for the Java [2] environment. Jacl is designed to be a universal scripting language for Java: the Jacl interpreter is written completely in Java and can run on any Java Virtual Machine. Jacl can be used to create Web content or to control Java applications.

This paper explains the need for Jacl as a scripting language for Java and discusses the implications of Jacl for both the Java and Tcl programming communities. It then describes how to use Jacl. It also explains the implementation of the Jacl interpreter and how to write Tcl extensions in Java.

1. Motivation

One on-going question in the Tcl community is, how can Tcl exploit the popularity of Java and the World Wide Web. There are two projects that try to bring Tcl into the world of Java and WWW. The Tcl Plugin [3] allows the execution of Tcl scripts inside Web browsers. However, the Tcl Plugin runs only inside certain browsers (Navigator and Explorer), requires the user to install software on local machines and does not communicate well with Java. Tcl-Java [4] allows the evaluation of Tcl code in Java applications, but it requires native methods and thus cannot run inside most browsers.

A Tcl implementation in Java will facilitate the creation of portable Tcl extensions [4]. Tcl is a portable scripting language. However, although Tcl provides some support for writing portable extensions, maintaining Tcl extensions written in C for multiple platforms is still a difficult task, especially if network or graphics programming is involved. Currently Tcl runs on more platforms than Java. However, due to the large number of commercial Java developers, Java will probably catch up in the near future and run on more platforms. If Tcl implementations can be written in Java, the Tcl community can leave the portability issues to JavaSoft and other Java implementers and concentrate on developing the Tcl core interpreter and extensions.

On the other hand, Java needs a scripting language as powerful as Tcl. Java is a structured programming language and is not a good scripting or command language [7]. Currently, scripting languages that can be used on Java platforms, such as Javascript and VBScript, are proprietary, non-portable and restrictive. Javascript and VBScript run only on the browsers that support them. Their scripting engines are system-dependent and cannot run on arbitrary Java Virtual Machines. These languages are good for scripting HTML pages, but they lack the features that would allow their deployment at any larger scale. For example, Javascript cannot define new classes; Java applets cannot directly pass events to VBScript [5, pp. 843]. Moreover, these scripting languages are not embeddable and thus cannot be used to control Java applic ations.

Jacl is a comprehensive solution to the problem of Tcl and Java integration. Since the Jacl interpreter and extensions are written completely in Java, they can run inside any JVM, making Tcl an embeddable, universal scripting language for Java. By using the Jacl interpreter, Java programmers can use Tcl to control simple Web pages, complex networked Java applications, and anything in between.

2. Using Jacl to Script Java Applications and Applets

Java applications and applets and very similar to each other. The following section concentrates on applets only but the discussion holds true for Java applications as well.