BeanShell

Simple Java Scripting

version 3.0.0

Table of Contents

- Table of Contents
- Introduction
  - Scripting vs. Application Languages
  - Tearing Down the Barriers
  - History
  - Conclusion
- Quick Start
  - Download and Run BeanShell
  - The BeanShell GUI
  - Java Statements and Expressions
  - Useful BeanShell Commands
  - Scripted Methods
  - Implementing Interfaces
  - Scripted Objects
  - Calling BeanShell From Your Application
  - Conclusion
- Basic Syntax
  - Standard Java Syntax
  - Loosely Typed Java Syntax
  - Exception Handling
  - Basic Scoping of Variables
  - Variable Modifiers
  - Convenience Syntax
  - Auto Boxing and Unboxing
  - Importing Classes and Packages
  - Document Friendly Entities
- Scripted Methods
  - Scoping of Variables and Methods
  - Scope Modifier: 'super'
- Scripted Objects
  - The 'this' reference
- Scope Modifiers
  - 'this', 'super', and 'global'
  - Synchronized Methods Revisited
- Scripting Interfaces
  - Anonymous Inner-Class Style
  - 'this' references as Interface Types
  - Interface Types and Casting
  - "Dummy" Adapters and Incomplete Interfaces
  - Threads - Scripting Runnable
  - Limitations
- Scripting Classes
  - Declaring a Scripted Class
  - Implementing Interfaces
  - Static Blocks
  - Persistent Scriptable Classes (Experimental)
  - Current Limitations
- Lambda Expressions
  - Basic Forms
  - Capturing Variables
  - Exceptions
  - Limitations
- Special Variables and Values
  - Special Members of 'this' type References
  - Undefined Variables
  - Setting the Command Prompt
- BeanShell Commands
  - Commands Overview
- Adding BeanShell Commands
  - Hello World
  - Compiled Commands
  - User Defined Commands with invoke()
  - Commands Scope
  - Getting the Caller Context

    - setNameSpace()
    - Getting the Invocation Text
    - Working with Dirctories and Paths
    - Working With Class Identifiers
    - Working with Iterable Types
- Strict Java Mode
- Class Loading and Class Path Management
    - Changing the Class Path
    - Auto-Importing from the Classpath
    - Reloading Classes
    - Loading Classes Explicitly
    - Setting the Default ClassLoader
    - Class Loading in Java
    - Class Loading in BeanShell
- Modes of Operation
    - Standalone
    - Remote (removed in 3.0)
    - Interactive Use
    - The .bshrc Init File
- Embedding BeanShell in Your Application
    - The BeanShell Core Distribution
    - Calling BeanShell From Java
    - eval()
    - EvalError
    - source()
    - Multiple Interpreters vs. Multi-threading
    - Scripting via javax.script (JSR223)
    - Serializing Interpreters and Scripted Objects
- Remote Server Mode (removed in 3.0)
- BshServlet and Servlet Mode Scripting (removed in 3.0)
- The BeanShell Demo Applet
- BeanShell Desktop
    - Shell Windows
    - Editor Windows
    - The Class Browser
- BshDoc - Javadoc Style Documentation
    - BshDoc Comments
    - BshDoc XML Output
    - The bshcommands.xsl stylesheet
- The BeanShell Parser
    - Validating Scripts With bsh.Parser
    - Parsing and Performance
    - Parsing Scripts Procedurally
- Using JConsole
    - ConsoleInterface
- Reflective Style Access to Scripted Methods
    - eval()
    - invokeMethod()
    - Method Lookup
    - BshMethod
    - Uses
- Executable scripts under Unix
- BSF Bean Scripting Framework
    - Ant
- Learning More
    - Helping With the Project
- Credit and Acknowledgments
    - License and Terms of Use
- BeanShell Commands Documentation

Introduction

This document is about BeanShell. BeanShell is a small, free, embeddable Java source interpreter with object scripting language features, written in Java. BeanShell executes standard Java statements and expressions but also extends Java into the scripting domain with common scripting language conventions and syntax. BeanShell is a *natural* scripting language for Java.

Scripting vs. Application Languages

Traditionally, the primary difference between a scripting language and a compiled language has been in its type system: the way in which you define and use data elements. You might be thinking that there is a more obvious difference here - that of "interpreted" code vs. compiled code. But the compiler in and of itself does not fundamentally change the way you work with a language. Nor does interpreting a language necessarily make it more useful for what we think of as "scripting". It is the type system of a language that makes it possible for a compiler to analyze the structure of an application for correctness. Without types, compilation is reduced to just a grammar check and an optimization for speed. From the developer's perspective, it is also the type system that characterizes the way in which we interact with the code.

Types are good. Without strongly type languages it would be very hard to write large scale systems and make any assertions about their correctness before they are run. But working with types imposes a burden on the developer. Types are labels and labeling things can be tedious. It can be especially tedious during certain kinds of development or special applications where it is flexibility and not program structure that is paramount. There are times where simplicity and ease of use is a more important criterion.

This is not just rationalization to cover some underlying laziness. Productivity affects what people do and more importantly do *not* do in the real world, much more than you might think. There is a lot of important software that exists in the world today only because the cost/benefit ratio in some developer's mind reached a certain threshold.

Unit testing - one of the foundations of writing good code - is a prime example. Unit tests for well written code are, in general, vitally important as a collective but almost insignificant individually. It's a "tragedy of the commons" that leads individual developers to repeatedly weigh the importance of writing another unit test with working on "real code". Give developers have a tool that makes it easy to perform a test with a line or two of code they will probably use it. If, moreover, it is also a tool that they enjoy using during their development process - that saves the time, they will be even more inclined to use it.

Customizability through scripting also opens the door to applications that are more powerful than the sum of their parts. When users can extend, enhance, and add to their applications they use them in new and unexpected ways.

Scripting is powerful.

Tearing Down the Barriers

Traditionally scripting languages have traded in the power of types for simplicity. Most scripting languages distill the type system to just one or a handful of types such as strings, numbers, or simple lists. This is sufficient for many kinds of scripting.

Many scripting languages operate in a loose, unstructured land - a place dominated by text and course-grained tools. As such these scripting languages have evolved sophisticated mechanisms for working with these simple types (regular expressions, pipes, etc.). As a result there has developed a casm between the scripting languages and the application languages created by the collapse of the type system in-between. The scripting languages have remained a separate species, isolated and speaking a different dialect from their brothers the application languages.

BeanShell is a new kind of scripting language. BeanShell begins with the standard Java language and bridges it into the scripting domain in a natural way, but allowing the developer to relaxing types where appropriate. It is possible to write BeanShell scripts that look exactly like Java method code. But it's also possible to write scripts that look more like a traditional scripting language, while still maintaining the framework of the Java syntax.

BeanShell emulates typed variables and parameters when they are used. This allows you to "seed" your code with strong types where appropriate. You can "shore up" repeatedly used methods as you work on them, migrating them closer to Java. Eventually you may find that you want to compile these methods and maintain them in standard Java. With BeanShell this is easy. BeanShell does not impose a syntactic boundary between your scripts and Java.

But the bridge to Java extends much deeper than simple code similarity. BeanShell is one of a new breed of scripting languages made possible by Java's advanced reflection capabilities. Since BeanShell can run in the same Java virtual machine as your application, you can freely work with real, live, Java objects - passing them into and out of your scripts. Combined with BeanShell's ability to implement Java interfaces, you can achieve seamless and simple integration of scripting into your Java applications. BeanShell does not impose a type boundary between your scripts and Java.

History

What seems like an eternity ago, back in the summer of 1993, I was working at Southwestern Bell Technology Resources and I was infatuated with the Tcl/Tk scripting language. On the advice of someone at Sun I also began playing around a bit with the Oak language written by James Gosling. Little did I know that within just a few years Oak, which would become Java, would not only spark a revolution, but that I would be writing one of the first books on the new Java language (Exploring Java, O'Reilly & Associates) and creating Java's first scripting language, BeanShell, drawing inspiration from Tcl.

BeanShell's first public release was not until 1997, but I had been poking at it in one form or another for some time before that. BeanShell as a language became practical when Sun added reflection to the Java language in version 1.1. After that, and after

having seen its value in helping me create examples and snippets for the second edition of my book, I decided to try to polish it up and release it.

BeanShell has slowly, but steadily gained popularity since then. It has grown in fits and spurts as its contributor's time has allowed. But recently BeanShell has achieved a sort of critical mass. BeanShell is distributed with Emacs as part of the JDE and with Sun Microsystem's NetBeans / Forte for Java IDEs. BeanShell is also bundled by BEA with their Weblogic application server. We've had reports of BeanShell being used everywhere from the high energy physics laboratory CERN, to classrooms teaching programming to nine year olds. BeanShell is being used in everything from large financial applications all the way down to embedded systems floating in Buoys in the pacific ocean. I attribute this success to the power of the open source development model and owe many thanks to everyone who has contributed.

Conclusion

I believe that BeanShell is the simplest and most natural scripting language for Java because it is, foremost, Java. BeanShell draws on a rich history of scripting languages for its scripting syntax and uses it to very conservatively extend the Java language into this new domain. I hope that you have half as much fun using BeanShell as I have had working on it and I welcome all comments and suggestions.

Quick Start

Welcome to BeanShell. This is a crash course to get you going. We'll leave out many important options and details. Please see the rest of the user's guide for more information.

Download and Run BeanShell

Download the latest JAR file from the download page and start up BeanShell either in the graphical desktop mode or on the command line. BeanShell 3.0 releases are distributed through GitHub releases and Maven Central, as `io.github.beanshell:bsh`.

If you just want to start playing around you may be able to launch the BeanShell desktop by simply double clicking on the BeanShell JAR file. More generally however you'll want to add the jar to your classpath so that you can work with your own classes and applications easily.

To do this you can either drop the BeanShell JAR file into your Java extensions folder or add it to your classpath. (Important: If you put BeanShell in the extensions folder and wish to use it with BSF applications like Jakarta Ant you must install the bsf.jar in the same location).

```
To install as an extension place the bsh.jar file in your
$JAVA_HOME/jre/lib/ext folder.  (OSX users: place the bsh.jar in
/Library/Java/Extensions or ~/Library/Java/Extensions for individual users.)

Or add BeanShell to your classpath like this:

unix:     export CLASSPATH=$CLASSPATH:bsh-xx.jar
windows:  set classpath %classpath%;bsh-xx.jar
```

Tip:
You can modify the classpath from within BeanShell using the addClassPath() and setClassPath() commands.

You can then run BeanShell in either a GUI or command line mode:

```
    java bsh.Console       // run the graphical desktop
or
    java bsh.Interpreter   // run as text-only on the command line
or
    java bsh.Interpreter filename [ args ] // run script file
```

It's also possible to call BeanShell from within your own Java applications, to reach it in a remote server mode for debugging, to use it as a servlet, or even in an applet. See "BeanShell Modes of Operation" for more details.

The BeanShell GUI

The BeanShell GUI desktop is meant to allow some experimentation with the features of BeanShell. It is not intended to be a replacement for a full featured IDE. Please check out the jEdit editor for an example of a full featured development environment

based in part on BeanShell scripting capabilities.

Upon starting the BeanShell in GUI mode a console window will open. By right clicking on the desktop background you can open additional console windows and other tools such as a simple class browser.

Each console window runs a separate instance of the BeanShell interpreter. The graphical console supports basic command history, line editing, cut and paste, and even class and variable name completion. From the console you can open a simple editor window. In it you can write scripts and use the 'eval' option to evaluate the text in the attached console's workspace or a new workspace.

Java Statements and Expressions

BeanShell understands standard Java statements, expressions, and method declarations. Statements and expressions are all of the normal things that you'd say inside a Java method such as variable declarations and assignments, method calls, loops, and conditionals.

You can use these exactly as they would appear in Java, however in BeanShell you also have the option of working with "loosely typed" variables. That is, you can simply omit the types of variables that you use (both primitives and objects). BeanShell will only signal an error if you attempt to misuse the actual type of the variable.

Here are some examples:

```
foo = "Foo";
four = (2 + 2)*2/2;
print( foo + " = " + four );  // print() is a BeanShell command

// Do a loop
for (i=0; i<5; i++)
    print(i);

// Pop up a frame with a button in it
button = new JButton( "My Button" );
frame = new JFrame( "My Frame" );
frame.getContentPane().add( button, "Center" );
frame.pack();
frame.setVisible(true);
```

Useful BeanShell Commands

In the previous example we used a convenient "built-in" BeanShell command called print(), to display values. print() does pretty much the same thing as System.out.println() except that it insures that the output always goes to the command line. print() also displays some types of objects (such as arrays) more verbosely than Java would. Another related command is show(), which toggles on and off automatic display of the result of every line you type.

Here are a few other examples of BeanShell commands:

- **source(), run()** - Read a bsh script into this interpreter, or run it in a new interpreter
- **frame()** - Display a GUI component in a Frame or JFrame.
- **load(), save()** - Load or save serializable objects to a file.
- **cd(), cat(), dir(), pwd(), etc.** - Unix-like shell commands
- **exec()** - Run a native application
- **javap()** - Print the methods and fields of an object, similar to the output of the Java javap command.
- **setAccessibility()** - Turn on unrestricted access to private and protected components.

See the complete list of BeanShell Commands for more information.

> **Tip:**
> BeanShell commands are not really "built-in" but are simply BeanShell scripts that are automatically loaded from the classpath. You can add your own scripts to the classpath to extend the basic command set.

Scripted Methods

You can declare and use methods in BeanShell just as you would in a Java class.

```
int addTwoNumbers( int a, int b ) {
    return a + b;
}
```

```
sum = addTwoNumbers( 5, 7 );  // 12
```

Bsh methods may also allow dynamic (loose) argument and return types.

```
add( a, b ) {
    return a + b;
}

foo = add(1, 2);            // 3
foo = add("Oh", " baby");   // "Oh baby"
```

Implementing Interfaces

Note: implementing arbitrary interfaces requires BeanShell be run under a Java 1.3 or higher environment.

You can use the standard Java anonymous inner class syntax to implement an interface type with a script. For example:

```
ActionListener scriptedListener = new ActionListener() {
    actionPerformed( event ) { ... }
}
```

You don't have to script all of the methods of an interface. You can opt to script only those that you intend to call if you want to. The calling code will simply throw an exception if it tries to invoke a method that isn't defined. If you wish to override the behavior of a large number of methods - say to produce a "dummy" adapter for logging - you can implement a special method signature: invoke(name, args) in your scripted object. The invoke() method is called to handle any undefined method invocations:

```
ml = new MouseListener() {
    mousePressed( event ) { ... }
    // handle the rest
    invoke( name, args ) { print("Method: "+name+" invoked!");
}
```

Scripted Objects

In BeanShell, as in JavaScript and Perl, method "closures" allow you to create scripted objects. You can turn the results of a method call into an object reference by having the method return the special value **this**. You can then use the reference to refer to any variables set during the method call. Useful objects need methods of course, so in BeanShell scripted methods may also contain methods at any level. For example:

```
foo() {
    print("foo");
    x=5;

    bar() {
        print("bar");
    }

    return this;
}

myfoo = foo();    // prints "foo"
print( myfoo.x ); // prints "5"
myfoo.bar();      // prints "bar"
```

If this "closure" thing seems strange to don't worry. It's just an evolutionary step that languages acquired along the path to Objects. Please see the user's manual for a more thorough explanation.

Within your scripts, BeanShell scripted objects (i.e. any *'this'* type reference like myFoo in the previous example) can automatically implement any Java interface type. When Java code calls methods on the interface the corresponding scripted methods will be invoked to handle them. BeanShell will automatically "cast" your scripted object when you attempt to pass it as an argument to a method that takes an interface type. For passing script references outside of BeanShell, you can perform an explicit cast where necessary. Please see the user manual for full details.

Calling BeanShell From Your Application

You can evaluate text and run scripts from within your application by creating an instance of the BeanShell interpreter and using the eval() or source() commands. You may pass in variable references to objects you wish to use in scripts via the set() method and

retrieve results with the get() method.

```
import bsh.Interpreter;

Interpreter i = new Interpreter();  // Construct an interpreter
i.set("foo", 5);                    // Set variables
i.set("date", new Date() );

Date date = (Date)i.get("date");    // retrieve a variable

// Eval a statement and get the result
i.eval("bar = foo*10");
System.out.println( i.get("bar") );

// Source an external script file
i.source("somefile.bsh");
```

Tip:
In the above example the Interpreter's eval() method also returned the value of bar as the result of the evaluation.

Conclusion

We hope this brief introduction gets you started. Please see the full user manual for more details. Please consult the mailing list archives for more useful information. http://www.beanshell.org/

Basic Syntax

BeanShell is, foremost, a Java interpreter. So you probably already know most of what you need to start scripting with BeanShell. This section describes specifically what portion of the Java language BeanShell interprets and how BeanShell extends it or "loosens" it to be more scripting-language-like.

Standard Java Syntax

In a BeanShell script (and on the command line) you can type normal Java statements and expressions and display the results. Statements and expressions are the kinds of things you normally find inside of a Java method: variable assignments, method calls, math expressions, for-loops, etc.

Here are some examples:

```
/*
    Standard Java syntax
*/

// Use a hashtable
Hashtable hashtable = new Hashtable();
Date date = new Date();
hashtable.put( "today", date );

// Print the current clock value
print( System.currentTimeMillis() );

// Loop
for (int i=0; i<5; i++)
    print(i);

// Pop up a frame with a button in it
JButton button = new JButton( "My Button" );
JFrame frame = new JFrame( "My Frame" );
frame.getContentPane().add( button, "Center" );
frame.pack();
frame.setVisible(true);
```

You can also define your own methods and use them just as you would inside a Java class. We'll get to that in a moment.

Loosely Typed Java Syntax

In the examples above, all of our variables have declared types. e.g. "JButton button". Beanshell will enforce these types, as you

will see if you later try to assign something other than a JButton to the variable "button" (you will get an error message). However BeanShell also supports "loose" or dynamically typed variables. That is, you can refer to variables without declaring them first and without specifying any type. In this case BeanShell will do type checking where appropriate at runtime. So, for example, we could have left off the types in the above example and written all of the above as:

```
/*
    Loosely Typed Java syntax
*/

// Use a hashtable
hashtable = new Hashtable();
date = new Date();
hashtable.put( "today", date );

// Print the current clock value
print( System.currentTimeMillis() );

// Loop
for (i=0; i<5; i++)
    print(i);

// Pop up a frame with a button in it
button = new JButton( "My Button" );
frame = new JFrame( "My Frame" );
frame.getContentPane().add( button, "Center" );
frame.pack();
frame.setVisible(true);
```

This may not seem like it has saved us a great deal of work. But you will see the difference when you come to rely on scripting as part of your development and testing process; especially for in interactive use.

When a "loose" variable is used you are free to reassign it to another type of Java object later. Untyped BeanShell variables can also freely hold Java primitive values like **int** and **boolean**. Don't worry, BeanShell always knows the real types and only lets you use the values where appropriate. For primitive types this includes doing the correct numeric promotion that the real Java language would do when you use them in an expression.

Exception Handling

Exception handling using try/catch blocks works just as it does in Java. For example:

```
try {
    int i = 1/0;
} catch ( ArithmeticException e ) {
    print( e );
}
```

But you can loosely type your catch blocks if you wish:

```
try {
    ...
} catch ( e ) {
    print( "caught exception: "+e );
}
```

Basic Scoping of Variables

> **Note:**
> As of BeanShell version 1.3 the default scoping of loosely typed variables was changed to be more consistent with Java. BeanShell still supports an alternate scoping used in earlier versions. This mode can be enabled for legacy code by setting the system property "localscoping" to true. See appendix "Local Scoping".

Variable scoping in BeanShell behaves, wherever possible, just like that in Java. Ordinary Java, however, does not offer "loose" variables (variables that can be used without being declared first). So we must define their behavior within BeanShell. We'll see in the next section that untyped variables - variables that are not declared and not assigned a value elsewhere - default to the *local* scope. This means that, in general, if you assign a value to a variable without first declaring it, you are creating a new local variable in the current scope.

Blocks

Blocks are statements between curly braces {}. In BeanShell, as in Java, blocks define a level of scope for typed variables: typed variables declared within a block are local to the block. Other assignments within the block occur, as always, wherever the variable was defined.

Untyped variables in BeanShell, however, are not constrained by blocks. Instead they act as if they were declared at the outer (enclosing) scope's level. With this in mind, BeanShell code looks just like Java code. In BeanShell if you declare a typed variable within a block it is local to the block. But if you use an untyped variable (which looks just like an ordinary assignment in Java) it behaves as an assignment to the enclosing scope.

This will make sense with a few examples:

```
// Arbitrary code block
{
    y = 2;      // Untyped variable assigned
    int x = 1;  // Typed variable assigned
}
print( y ); // 2
print( x ); // Error! x is undefined.

// Same with any block statement: if, while, try/catch, etc.
if ( true ) {
    y = 2;      // Untyped variable assigned
    int x = 1;  // Typed variable assigned
}
print( y ); // 2
print( x ); // Error! x is undefined.
```

Variables declared in the for-init area of a for-loop follow the same rules as part of the block:

```
for( int i=0; i<10; i++ ) {  // typed for-init variable
    j=42;
}
print( i ); // Error! 'i' is undefined.
print( j ); // 42

for( z=0; z<10; z++ ) { }   // untyped for-init variable
print( z ); // 10
```

Variable Modifiers

The standard Java variable modifiers may be used on typed variables: private / protected / public, final, transient, volatile, static. Only 'final' is currently implemented. The others are currently ignored.

Modifiers may not be applied to untyped variables.

> **Note:**
> As of BeanShell 3.0 the parser also accepts Java annotations on class, method, field and parameter declarations, including qualified names, named pairs and nested annotations. They are parsed and discarded, so nothing is retained on a generated class and none of them are visible through reflection.

Convenience Syntax

In BeanShell you may access JavaBean properties as if they were fields:

```
button = new java.awt.Button();
button.label = "my button";  // Equivalent to: b.setLabel("my button");
print( button.label );       // Equivalent to print( b.getLabel() );
```

JavaBean properties are simply pairs of "setter" and "getter" methods that adhere to a naming convention. In the above example BeanShell located a "setter" method with the name "setLabel()" and used it to assign the string value. It then found the method named getLabel() to retrieve the value.

Boolean properties may optionally use the syntax "is" for their "getter". e.g.

```
Float f = new Float(42f);
print( f.infinite );  // Equivalent to print( f.isInfinite() ); // false
```

If there is any ambiguity with an actual Java field name of the object (e.g. label in the above example) then the actual field name takes precedence. If you wish to avoid any ambiguity BeanShell provides an additional, uniform syntax for accessing both Java Bean properties and Hashtable or Map entries. You may use the "{}" curly brace construct with a String identifier as a qualifier on any variable of the appropriate type:

```
b = new java.awt.Button();
b{"label"} = "my button";  // Equivalent to: b.setLabel("my button");

h = new Hashtable();
h{"foo"} = "bar";          // Equivalent to: h.put("foo", "bar");
```

Where the java.util.Collections API is available, Maps are also supported.

Enhanced 'for' Loop

BeanShell supports the Java 1.5 style enhanced for-loop for iterating over collections and array types. (Note that you do not have to be running Java 1.5 to use this feature).

```
List foo = getSomeList();

for ( untypedElement : foo )
        print( untypedElement );

for ( Object typedElement: foo )
        print( typedElement );

int [] array = new int [] { 1, 2, 3 };

for( i : array )
        print(i);

for( char c : "a string" )
        print( c );
```

Supported iterable types include all the obvious things.

- JDK 1.1+ - (no collections): Enumeration, arrays, Vector, String, StringBuffer
- JDK 1.2+ - (w/collections): Collections, Iterator

See also the BshIterator API which supports the ehanced for-loop and allows iteration over these types using the dynamically loaded BeanShell Collection manager.

Switch Statements

In BeanShell, the switch statement may be used not only with numeric types but with objects. For example, you may switch on Dates and Strings which are compared for equality with their equals() methods:

```
dateobj = new Date();

switch( dateobj )
{
        case newYears:
                break;
        case christmas:
                break;
        default:
}
```

Auto Boxing and Unboxing

"Boxing" and "Unboxing" are the terms used to describe automatically wrapping a primitive type in a wrapper class and unwrapping it as necessary. Boxing is a feature of Java (SDK1.5) and has been supported in BeanShell for many years.

BeanShell supports boxing and unboxing of primitive types. For example:

```
int i=5;
Integer iw = new Integer(5);
print( i * iw );  // 25

Vector v = new Vector();
```

```
v.put(1);
int x = v.getFirstElement();
```

> **Note:**
> As of BeanShell 3.0, arithmetic on **float** operands (+, -, *, /, %), including a float mixed with byte, short, char, int or long, follows Java's own numeric promotion: it is computed in float and returns a Float, instead of being widened to double as in earlier versions. This means a float result that overflows is now Infinity rather than a larger Double result (e.g. Float.MAX_VALUE * 2), and a compound assignment such as `long += float` rounds through float, exactly as compiled Java does. Arithmetic involving double, BigInteger or BigDecimal is unchanged.

Importing Classes and Packages

In BeanShell as in Java, you can either refer to classes by their fully qualified names, or you can **import** one or more classes from a Java package.

```
// Standard Java
import javax.xml.parsers.*;
import mypackage.MyClass;
```

In BeanShell import statements may appear anywhere, even inside a method, not just at the top of a file. In the event of a conflict, later imports take precedence over earlier ones.

A somewhat experimental feature is the "super import". With it you may automatically import the entire classpath, like so:

```
import *;
```

The first time you do this BeanShell will map out your entire classpath; so this is primarily intended for interactive use. Note that importing every class in your classpath can be time consuming. It can also result in a lot of ambiguities. Currently BeanShell will report an error when resolving an an ambiguous import from mapping the entire classpath. You may disambiguate it by importing the class you intend.

> **Tip:**
> The BeanShell which() command will use the classpath mapping capability to tell you where exactly in your classpath a specified class is located:
>
> ```
> bsh % which( java.lang.String );
> Jar: file:/usr/java/j2sdk1.4.0/jre/lib/rt.jar
> ```

See "Class Path Management" for information about modifying the BeanShell classpath at run-time with the addClassPath() or setClassPath() commands.

Also see "BeanShell Commands" for information about importing new BeanShell commands from the classpath.

Default Imports

By default, common Java core and extension packages are imported for you. They are, in the order in which they are imported:

- javax.swing.event
- javax.swing
- java.awt.event
- java.awt
- java.net
- java.util
- java.io
- java.lang

Two BeanShell package classes are also imported by default:

- bsh.EvalError
- bsh.Interpreter

Finally, we should mention that BeanShell commands may be imported from the classpath. The default commands are imported in the following way:

```
importCommands("/bsh/commands");
```

We will discuss how to import your own commands in a later section.

> **Tip:**
> The classes java.awt.List and java.util.List are both imported by default. Because java.util.List is imported later, as part of the java.util package, it takes precedence. To access java.awt.List simply import it in, or the java.awt package again your script. Later imports take precedence.

Static Imports

As of BeanShell 2.0 you can import the static methods and fields of a Java class into a BeanShell namespace, using the same syntax as Java 5:

```
import static java.lang.Math.*;
print( sqrt(4.0) ); // 2.0
```

You can also import the static members of an individual class name directly, without the wildcard:

```
import static java.lang.Math.sqrt;
print( sqrt(4.0) ); // 2.0
```

You can also import the methods and fields of a Java object *instance* into a BeanShell namespace, using the importObject() command. This works like a static import, but pulls in the instance methods and fields of a particular object rather than the static members of a class:

```
Map map = new HashMap();
importObject( map );
put("foo", "bar");
print( get("foo") ); // "bar"
```

Document Friendly Entities

BeanShell supports special overloaded text forms of all common operators to make it easier to embed BeanShell scripts inside other kinds of documents (e.g XML).

@gt	>
@lt	<
@lteq	<=
@gteq	>=
@or	||
@and	&&
@bitwise_and	&
@bitwise_or	|
@left_shift	<<
@right_shift	>>
@right_unsigned_shift	>>>
@and_assign	&=
@or_assign	|=
@left_shift_assign	<<=
@right_shift_assign	>>=
@right_unsigned_shift_assign	>>>=

Scripted Methods

You can define define methods in BeanShell, just as they would appear in Java:

```
int addTwoNumbers( int a, int b ) {
    return a + b;
}
```

And you can use them in your scripts just as you would any Java method or "built-in" BeanShell command:

```
sum = addTwoNumbers( 5, 7 );
```

Just as BeanShell variables may be dynamically typed, methods may have dynamic argument and return types. We could, for example, have declared our add() method above like so:

```
add( a, b ) {
    return a + b;
}
```

In this case, BeanShell would dynamically determine the types when the method is called and attempt to "do the right thing":

```
foo = add(1, 2);
print( foo ); // 3

foo = add("Oh", " baby");
print( foo ); // Oh baby
```

In the first case Java performed arithmetic addition on the integers 1 and 2. (By the way, if we had passed in numbers of other types BeanShell would have performed the appropriate numeric promotion and returned the correct Java primitive type.) In the second case BeanShell performed the usual string concatenation for String types and returned a String object. This example is a bit extreme, as there are no other overloaded operators like string concatenation in Java. But it serves to emphasize that BeanShell methods can work with loose types.

Methods with unspecified return types may return any type of object (as in the previous example). Alternatively they may also simply issue a "return;" without a value, in which case the effective type of the method is "void" (no type). In either case, the return statement is optional. If the method does not perform an explicit "return" statement and the return type is not explicitly set to void, the value of the last statement or expression in the method body becomes the return value (and must adhere to any declared return typing).

Method Modifiers and 'throws' Clauses

The standard Java modifiers may be applied to methods: private / protected / public, synchronized, final, native, abstract, and static.

The synchronized modifier is the only modifier currently implemented. The others are ignored. The 'throws' clause of methods is checked for valid class type names, but is not otherwise enforced.

Synchronized methods are synchronized on the object representing the method's common parent scope, so they behave like Java methods contained in a class. We will return to this topic after discussing scripted objects and "closures".

```
// foo() and bar() are synchronized as if they were in a common class
synchronized foo() { }
synchronized bar() { }
```

Scoping of Variables and Methods

As in Java, a method can refer to the values of variables and method names from the enclosing scope (in Java the "enclosing scope" would be a class). For example:

```
a = 1;
anotherMethod() { ... }

foo() {
    print( a );
        a = a+1;
```

```
        anotherMethod();
}

// invoke foo()
foo();      // prints 1
print( a ); // prints 2
```

Variables and methods are "inherited" from the parent scope in the usual way. In the example above there are just two levels of scope: the top or "global" scope and the scope of the method foo(). Later we'll talk about scripting objects in BeanShell and see that there can be arbitrary levels of scoping involved. But the rules will be the same.

As in Java, a typed variable is not visible outside the scope in which it is declared. So declaring a variable with a type is a way to limit its scope or make a *local* variable. In BeanShell using an untyped or "loosely" typed variable is also equivalent to declaring a local variable. That is, if you use a variable that has not been defined elsewhere, it defaults to the local scope:

```
a = 1;

foo() {
        a = a + 1; // a is defined in parent scope
        b = 3;     // undefined, defaults local scope
        int c = 4; // declared local scope
}

// invoke foo()
print( a ); // prints 2
print( b ); // ERROR! b undefined
print( c ); // ERROR! c undefined
```

In the above example the variable 'a' is declared in the global scope. When its value is read and assigned inside of foo() the global value of 'a' will be affected.

The variable 'b' is a usage of an untyped variable. Since 'b' has not been declared or assigned a value in any enclosing scope, it becomes a local variable 'b' in the scope of foo. The variable 'c' is explicitly declared (with a type) in the scope of foo() and is therefore, of course, local to foo().

Later we'll see that BeanShell allows arbitrary nesting of methods. If we were to declare another method inside of foo() it could see all of these variables (a, b, and c) as it is also in the scope of foo().

Scoping of Loosely Typed Variables

As in Java, declaring a variable with a type will always make it local. Even if the variable exists in the outer scope, it will be hidden by the local variable declaration. But what of loosely typed variables? As we've seen, untyped variable usage looks just like an ordinary Java assignment. What do we do if we want to make a local variable with the same name as a global one? One answer would be to resort to declaring the variable with a type. But if we wish to continue working with loosely typed variables in this case we have two options: We can explicitly declare a loosely typed variable with the BeanShell 'var' type. Or we can simply qualify our assignment with the 'this.' qualifier.

If you wish to, you can explicitly declare an untyped variable (making it local) using the special type 'var'. e.g.

```
foo() {
        var a = 1;
}
foo();
print( a ); // ERROR! a is undefined!
```

'var' is a magic type in BeanShell that represents a loose (untyped) variable. The default value of a variable declared with 'var' is null.

Alternately, you can use the scope modifier 'this' to explicitly qualify the variable assignment and make it local.

```
foo() {
        this.a = 1;
}
foo();
print( a ); // ERROR! a is undefined!
```

In this example we used the modifier 'this' to qualify an untyped variable's scope and make it local. We will explain 'this' and what it means in BeanShell scripted methods in the next section on Scripted Objects.

Scope Modifier: 'super'

Within a method, it is possible to explicitly qualify a variable or method reference with the identifier 'super' in order to refer to a variable or method defined in an enclosing scope (the scope in which the method is defined or "higher"). e.g.

```
int a = 42;

foo() {
    int a = 97;
    print( a );
    print( super.a );
}

foo();  // prints 97, 42
```

As in Java, the 'super' modifiers tells the scoping to begin its search for the variable or method in the parent scope. In the case above, the variable 'a' by default refers to the variable in the local scope. By qualifying 'a' with 'super' we can refer to the variable 'a' in the global scope (the "topmost" scope).

So, we've seen that 'super' can be used to refer to the method's parent context. We'll see in the next section how 'this' and 'super' are used in scripting Objects in BeanShell.

Scripted Objects

Many people who use BeanShell use it to write scripts that work with existing Java classes and APIs, or perform other kinds of dynamic activities for their own applications at run-time without the aid of a compiler. Often this means writing relatively unstructured code - for example, a sequence of method invocations or loops, all contained in a single script file or eval() statement. In the previous section we saw that BeanShell is also capable of scripting methods, just like Java. Creating methods and new BeanShell commands (which are just methods in their own files) is the natural progression of organizing your scripts into re-usable and maintainable components.

Beyond methods and structured programming lie, of course, objects and the full breadth of object oriented programming. In Java objects are the products of classes. While BeanShell is compatible with standard Java syntax for statements, expressions, and methods, you can't yet script new Java classes within BeanShell. Instead, BeanShell allows you to script objects as "method closures", similar to the way it is done in Perl 5.x, JavaScript, and other object-capable scripting languages. This style of scripting objects (which we'll describe momentarily) is simple and flows very naturally from the style of scripting methods. The syntax, as you'll see, is a straightforward extension of the standard Java concept of referring to an object with a 'this' reference.

> **Note:**
> In standard Java, a method inside of an object (an instance method) may refer to the enclosing object using the special variable 'this'. For example:
>
> ```
> // MyClass.java
> MyClass {
>     Object getObject() {
>         return this; // return a reference to our object
>     }
> }
> ```
>
> In the example above, the getObject() method of MyClass returns a reference to its own object instance (an instance of the MyClass object) using 'this'.

The 'this' reference

As in most languages, an executing method in BeanShell has its own "local" scope that holds argument (parameter) variables and locally declared variables. For example, in the following code segment any variables that we might use within the foo() method will normally only be visible within the scope of foo() and for the lifetime of one particular foo() method invocation:

```
// Define the foo() method:
foo() {
    int bar = 42;
    print( bar );
}

// Invoke the foo() method:
foo();  // prints 42
```

```
print( bar ); // Error, bar is undefined here
```

In the above, the bar variable is local to foo() and therefore not available outside of the method invocation - it is thrown away when the method exits, just like a standard Java local variable.

Now comes the twist - In BeanShell you have the option to "hang on" to the scope of a method invocation after exiting the method by referring to the special 'this' reference. As in Java, 'this' refers to the current object context. The only difference is that in this case the context is associated with the method and not a class instance.

By saving the 'this' reference after the method returns, you can continue to refer to variables defined within the method, using the standard Java "." notation:

```
foo() {
    int bar = 42;
    return this;
}

fooObject = foo();
print( fooObject.bar ); // prints 42!
```

In the above, the value returned by the foo() method (the 'this' reference) can be thought of as an instance of a "foo" object. Each foo() method invocation effectively creates a new object; foo() is now not just a method, but a kind of object constructor.

In the above case our foo object is not so much an object, but really more of a structure. It contains variables (bar) but no "behavior". The next twist that we'll introduce is that BeanShell methods are also allowed to contain other methods:

```
foo() {
    bar() {
        ...
    }
}
```

Scripted methods may define any number of nested methods in this way, to an arbitrary depth. The methods are "local" to the method invocation.

Statements and expressions within the enclosing BeanShell method can call their "local" methods just like any other method. (Locally declared methods override outer-more methods like local variables hide instance variables in Java.) The enclosed methods are not directly visible outside of their enclosing method. However, as you might expect, we can invoke them as we would on a Java object, through an appropriate object reference:

```
foo() {
    int a = 42;
    bar() {
        print("The bar is open!");
    }

    bar();
    return this;
}

// Construct the foo object
fooObject = foo();     // prints "the bar is open!"
// Print a variable of the foo object
print ( fooObject.a )  // 42
// Invoke a method on the foo object
fooObject.bar();       // prints "the bar is open!"
```

Methods declared inside block structures within methods behave just as if they were declared directly in the method. i.e. there are no block-local methods. For example:

```
foo() {

        bar() { }

        if ( true ) {
                bar2() { }
        }

        return this;
}
```

In the above example the methods bar() and bar2() are both defined within foo().

In the next section we'll return to the topic of variable scoping and go into more depth about how to work with scripted methods and objects.

Scope Modifiers

Now that we've seen how methods can be nested and treated as objects, we can revisit the topic of variable scope and scope modifiers.

'this', 'super', and 'global'

In the "Scripted Methods" section we described the use of 'super' to refer to a method's parent scope (the scope in which the method is defined). And in the previous section we talked about super's brother 'this', which refers to the current method's scope, allowing us to think of a method scope as an object. Now we can see how these concepts are related. Any method scope, and indeed the 'global' scope, can be thought as an object context. A scripted object can be thought of as encapsulated in a parent scope that determines its "environment" - its inherited variables and methods.

The references 'this', 'super', and 'global' are really the same kind of reference - references to BeanShell method contexts, which can be used as scripted objects. From here on We'll refer to 'this', 'super', 'global', and any other reference to a scripted object context in general as a *'this' type reference*.

> **Note:**
> If you print a 'this' type reference you'll see what it refers to:
>
> ```
> BeanShell 1.3 - by Pat Niemeyer (pat@pat.net)
> bsh % print( this );
> 'this' reference (XThis) to Bsh object: global
> bsh % foo() { print(this); print(super); }
> bsh % foo();
> 'this' reference (XThis) to Bsh object: foo
> 'this' reference (XThis) to Bsh object: global
> ```

The above note shows that the foo() method's 'this' reference is local (named 'foo') and that it's parent is the global scope; the same scope in which foo is defined.

'global'

The scope modifier 'global' allows you to always refer to the top-most scope. In the previous note you can see that the top level script context is called "global" and that it appears again as the 'super' of our foo() method. The global context is always the top scope of the script. It is the global namespace of the current interpreter. Referring to 'super' from the top scope simply returns the same 'global' again.

```
global.foo = 42;
```

Global variables are not special in any way. Their visibility derives simply from the fact that they are in the topmost scope. However, for those who do not like the idea of qualifying anything with "global". You can always use a more object oriented approach like the following.

```
// Create a top level object to hold some state
dataholder = object();

foo() {
    ...
    bar() {
        dataholder.value = 42;
    }

    bar();
    print( dataholder.value );
}
```

In the above example we used a global object to hold some state, rather than putting the 'value' variable directly in the global scope.

```

```

Tip:
In the above example we used the BeanShell object() command to create an "empty" BeanShell scripted object context in which to hold some data. The object() command is just a standard empty method named object() that returns 'this'. The variable 'dataholder' above is a 'this' type reference and has all of the properties of any other BeanShell object scope.

Synchronized Methods Revisited

Now that we have covered the meaning of 'this' and 'super' with respect to BeanShell methods we can define the meaning of the 'synchronized' modifier for BeanShell methods. Synchronized BeanShell methods behave as if they were in a common class by synchronizing on their common 'super' reference object. For example, in the four cases in the following example, synchronization occurs on the same Java object. That object is the 'this' type reference of the global scope (a Beanshell object of type bsh.This):

```
print( this ); // 'this' reference (XThis) to Bsh object: global

// The following cases all synchronize on the same lock
synchronized ( this ) { }     // synchronized block
synchronized int foo () { }   // synchronized method foo()
synchronized int bar () { }   // synchronized method bar()
int gee() {
        synchronized( super ) { }  // synchronized blockinside gee()
}
```

Scripting Interfaces

One of the most powerful features of BeanShell is the ability to script Java interfaces. This feature allows you to write scripts that serve as event handlers, listeners, and components of other Java APIs. It also makes calling scripted components from within your applications easier because they can be made to look just like any other Java object.

> **Note:**
> The techniques on this page use a 'this' reference to stand in for an interface type. Since BeanShell 2.0 you can also declare real Java classes and interfaces in your scripts with the ordinary `class`, `interface`, `extends` and `implements` keywords. See "Scripting Classes" which follows this section.

Anonymous Inner-Class Style

One way to get a scripted component to implement a Java interface is by using the standard Java anonymous inner class syntax to construct a scripted object implementing the interface type. For example:

```
buttonHandler = new ActionListener() {
    actionPerformed( event ) {
        print(event);
    }
};

button = new JButton();
button.addActionListener( buttonHandler );
frame(button);
```

In the above example we have created an object that implements the `ActionListener` interface and assigned it to a variable called buttonHandler. The buttonHandler object contains the scripted method actionPerformed(), which will be called to handle invocations of that method on the interface.

Note that in the example we registered our scripted ActionListener with a JButton using its addActionListener() method. The JButton is, of course, a standard Swing component written in Java. It has no knowledge that when it invokes the buttonHandler's actionPerformed() method it will actually be causing the BeanShell interpreter to run a script to evaluate the outcome.

To generalize beyond this example a bit - Scripted interfaces work by looking for scripted methods to implement the methods of the interface. A Java method invocation on a script that implements an interface causes BeanShell to look for a corresponding scripted method with a matching signature (name and argument types). BeanShell then invokes the method, passing along the arguments and passing back any return value. When BeanShell runs in the same Java VM as the rest of the code, you can freely pass "live" Java objects as arguments and return values, working with them dynamically in your scripts; the integration can be seamless.

See also the dragText example.

'this' references as Interface Types

The anonymous inner class style syntax which we just discussed allows you to explicitly create an object of a specified interface type, just as you would in Java. But BeanShell is more flexible than that. In fact, within your BeanShell scripts, any 'this' type script reference can automatically implement any interface type, as needed. This means that you can simply use a 'this' reference to your script or a scripted object anywhere that you would use the interface type. BeanShell will automatically "cast" it to the correct type and perform the method delegation for you.

For example, we could script an event handler for our button even more simply using just a global method, like this:

```
actionPerformed( event ) {
    print( event );
}

button = new JButton("Foo!");
button.addActionListener( this );
frame( button );
```

Here, instead of making a scripted object to hold our actionPerformed() method we have simply placed the method in the current context (the global scope) and told BeanShell to look there for the method.

Just as before, when `ActionEvents` are fired by the button, your actionPerformed() method will be invoked. The BeanShell 'this' reference to our script implements the interface and directs method invocations to the appropriately named method, if it exists.

> **Note:**
> If you want to have some fun, try entering the previous example interactively in a shell or on the command line. You'll see that you can then redefine actionPerformed() as often as you like by simply entering the method again. Each button press will find the current version in your shell. In a sense, you are working inside a dynamic Java object that you are creating and modifying as you type. Neat, huh? Be the Bean!

Of course, you don't have to define all of your interface methods globally. You can create references in any scope, as we discussed in "Scripting Objects". For example, the following code creates a scripted message button object which displays a message when its pushed. The scripted object holds its own actionPerformed() method, along with a variable to hold the Frame used for the GUI:

```
messageButton( message ) {
    JButton button = new JButton("Press Me");
    button.addActionListener( this );
    JFrame frame = frame( button );

    actionPerformed( e ) {
        print( message );
        frame.setVisible(false);
    }
}

messageButton("Hey you!");
messageButton("Another message...");
```

The above example creates two buttons, with separate messages. Each button prints its message when pushed and then dismisses itself. The buttons are created by separate calls to the messageButton() method, so each will have its own method context, separate local variables, and a separate instance of the ActionListener interface handler. Each registers itself (its own method context) as the ActionListener for its button, using its own 'this' reference.

In this example all of the "action" is contained in messageButton() method context. It serves as a scripted object that implements the interface and also holds some state, the frame variable, which is used to dismiss the GUI. More generally however, as we saw in the "Scripting Objects" section, we could have returned the 'this' reference to the caller, allowing it to work with our messageButton object in other ways.

Interface Types and Casting

It is legal, but not usually necessary to perform an explicit cast of a BeanShell scripted object to an interface type. For example:

```
actionPerformed( event ) {
    print( event );
}

button.addActionListener(
    (ActionListener)this ); // added cast
```

In the above, the cast to ActionListener would have been done automatically by BeanShell when it tried to match the 'this' type argument to the signature of the addActionListener() method.

Doing the cast explicitly has the same effect, but takes a different route internally. With the cast, BeanShell creates the necessary adapter that implements the ActionListener interface first, at the time of the cast, and then later finds that the method is a perfect match.

What's the difference? Well, there are times where performing an explicit cast to control when the type is created may be important. Specifically, when you are passing references out of your script, to Java classes that don't immediately use them as their intended type. In our earlier discussion we said that automatic casting happens "within your BeanShell scripts". And in our examples so far BeanShell has always had the opportunity to arrange for the scripted object to become the correct type, before passing it on. But it is possible for you to pass a 'this' reference to a method that, for example, takes the type 'Object', in which case BeanShell would have no way of knowning what it was destined for later. You might do this, for example, if you were placing your scripted objects into a collection (Map or List) of some kind. In that case, you can control the process by performing an explicit cast to the desired type before the reference leaves your script.

Another case where you may have to perform a cast is where you are using BeanShell in an embedded application and returning a scripted object as the result of an eval() or a get() variable from the Interpreter class. There again is a case where BeanShell has no way of knowing the intended type within the script. By performing an explicit cast you can create the type before the reference leaves your script.

We'll discuss embedded applications of BeanShell in the "Embedding BeanShell" section a bit later, along with the Interpreter getInterface() method, which is another way of accomplishing this type of cast from outside a script.

"Dummy" Adapters and Incomplete Interfaces

It is common in Java to see "dummy" adapters created for interfaces that have more than one method. The job of a dummy adapter is to implement all of the methods of the interface with stubs (empty bodies), allowing the developer to extend the adapter and override just the methods of interest.

We hinted in our earlier discussion that BeanShell could handle scripted interfaces that implement only the subset of methods that are actually used and that is indeed the case. You are free in BeanShell to script only the interface methods that you expect to be called. The penalty for leaving out a method that is actually invoked is a special run-time exception: java.lang.reflect.UndeclaredThrowableException, which the caller will receive.

The UndeclaredThrowableException is an artifact of Java Proxy API that makes dynamic interfaces possible. It says that an interface threw a checked exception type that was not prescribed by the method signature. This is a situation that cannot normally happen in compiled Java. So the Java reflection API handles it by wrapping the checked exception in this special unchecked (RuntimeException) type in order to throw it. You can get the underlying error using the exception's getCause() method, which will, in this case, reveal the BeanShell EvalError exception, reporting that the scripted method of the correct signature was not found.

The invoke() Meta-Method

BeanShell provides a very simple short-hand mechanism for scripting interfaces with large numbers of methods. You can implement the special method *invoke(name, args)* in any scripted context. The invoke() method will be called to handle the invocation of any method of the interface that is not defined. For example:

```
mouseHandler = new MouseListener() {
    mousePressed( event ) {
        print("mouse button pressed");
    }

    invoke( method, args ) {
        print("Undefined method of MouseListener interface invoked:"
            + name +", with args: "+args
        );
    }
};
```

In the above example we have neglected to implement four of the five methods of the MouseListener interface. They will be handled by the invoke() method, which will simply print the name of the method and its arguments. However since mousePressed() is defined it will be called for the interface.

Here is a slightly more realistic example of where this comes in handy. Let's use the invoke() method to print the names of methods called via the ContentHandler interface of the Java SAX API, while parsing an XML document.

```
import javax.xml.parsers.*;
```

```
import org.xml.sax.InputSource;

factory = SAXParserFactory.newInstance();
saxParser = factory.newSAXParser();
parser = saxParser.getXMLReader();
parser.setContentHandler( this );

invoke( name, args ) {
    print( name );
}

parser.parse( new InputSource(bsh.args[0]) );
```

By running this script with the XML file as an argument, we can see which of the dozen or so methods of the SAX API are being exercised by the structure of the document, without having to write a stub for each of them.

Tip:
You can use the invoke(name, args) meta-method directly in your own scope or in the global scope as well, in which case you can handle arbitrary "unknown" method invocations yourself, perhaps to implement your own "virtual" commands. Try typing this on the command line:

```
invoke(name,args) { print("Command: "+name+" invoked!"); }
noSuchMethod(); // prints "Command: noSuchMethod() invoked!"
```

Threads - Scripting Runnable

BeanShell 'this' type references can implement the standard java.lang.Runnable interface. So you can declare a "run()" method in your bsh objects and make it the target of a Thread:

```
foo() {
    run() {
        // do work...
    }
    return this;
}

foo = foo();
// Start two threads on foo.run()
new Thread( foo ).start();
new Thread( foo ).start();
```

BeanShell is thread-safe internally, so as long as your scripts do not explicitly do anything ordinarily non-thread safe (e.g. access shared variables or objects) you can write multi-threaded scripts.

Note:
You can use the bg() "background" command to run an external script in a separate thread. See bg().

Limitations

When running under JDK 1.3 or greater BeanShell can script any kind of Java interface. However when running under JDK 1.2 (or JDK1.1 + Swing) only the core AWT and Swing interfaces are available. To support those legacy cases a special extension of the 'this' reference implementation (the bsh.This class) is loaded which implements these interfaces along with Runnable, statically.

Scripting Classes

Since BeanShell 2.0, in addition to the 'this' reference style of scripting objects and interfaces described in the previous section, you can also declare real Java classes and interfaces in your scripts using the ordinary Java `class` and `interface` keywords. A scripted class may extend another class (scripted or compiled) and implement any number of interfaces, just as it would in compiled Java. Instances of scripted classes are real instances of that class - they pass `instanceof` checks for the class and all of its superclasses and interfaces - which was not the case in earlier versions of BeanShell.

Declaring a Scripted Class

Constructors, instance and static fields, and instance and static methods all work as you would expect from Java. Method and

constructor bodies are, of course, still loosely typed BeanShell code.

```
class Foo
{
    static int a = 5;
    int b;

    Foo() { }             // Foo needs a default constructor for Bar()
    Foo( int c ) { b = c; }

    void method() {
        print( "a is " + a );
    }

    static void smethod() {
        print( "static method, a is " + a );
    }
}

class Bar extends Foo
{
    method() {             // override, loosely typed
        print( "Bar.method() overrides Foo.method()" );
        super.method();     // call the overridden method
    }
}

Bar bar = new Bar();
print( bar instanceof Bar );  // true
print( bar instanceof Foo );  // true
bar.method();
Bar.smethod();                 // static members are inherited
```

As in Java, a subclass whose superclass does not have a no-argument constructor must define a constructor that calls one of the superclass's constructors explicitly.

Implementing Interfaces

You may also declare interfaces directly in a script, and implement them with the standard `implements` clause:

```
interface Named
{
    String getName();
}

class Person implements Named
{
    String name;
    Person( String name ) { this.name = name; }
    String getName() { return name; }
}

Named n = new Person( "Pat" );
print( n.getName() );
```

This works equally well with ordinary Java interfaces, including AWT and Swing listener types, giving you an alternative to the anonymous 'this' reference style described in "Scripting Interfaces" whenever you want a real, named, reusable class instead of an ad-hoc object.

> **Note:**
> The 'this' reference style of scripting objects and interfaces, described in the previous section, is still fully supported and remains the simplest way to script a one-off handler or adapter. Reach for a scripted `class` when you want inheritance, real `instanceof` behavior, or a type you intend to construct more than once.

Static Blocks

Static initializer blocks in the class body are supported:

```
class Foo
{
    static int a;
```

```
    static {
        a = 42;
    }
}
```

Persistent Scriptable Classes (Experimental)

Ordinarily a scripted class is generated dynamically, in memory, the first time it is referenced. Using the system property `-DsaveClasses=<savedir>` you can instead instruct BeanShell to write out a persistent `.class` file for each class definition it finds in a script. In this mode BeanShell does not execute the script; it only generates and stores the class files.

A class "Foo.class" generated this way expects to find its associated scripted class definition in a file "Foo.bsh" in the same location as the class file, so each stored class must have a corresponding script. For example, given:

```
// File: Foo.bsh
class Foo {
    Foo() { print("I'm being constructed!"); }
}
```

you can generate Foo.class in the current directory with:

```
java -DsaveClasses=. bsh.Interpreter Foo.bsh
```

The resulting Foo class may then be used like any ordinary Java class and invoked from ordinary Java code, e.g. `new Foo()`. When it is loaded it automatically starts a BeanShell interpreter to run its corresponding Foo.bsh script, so the script is still free to contain additional loose code beyond the class definition itself; that code runs when the class is initialized.

> **Note:**
> This feature is experimental. Each saved class currently initializes its own interpreter instance when loaded.

Current Limitations

A few corners of scripted classes are still incomplete:

- In order to run a class file's main method with arguments from the command line, the class containing your main method must be the last item in the file.
- Non-static inner classes are implemented using a non-standard mechanism (mix-ins), so mixing interpreted classes with their pre-compiled inner classes does not work properly in all cases.
- Inner classes are not properly "imported" in some cases: e.g. class A contains one or more inner classes, class B extends A, and B attempts to refer to those inner classes by their unqualified names.

Lambda Expressions

Since BeanShell 3.0 you can write lambda expressions - `x -> ...`, `(a, b) -> ...`, `(Type a) -> ...` - and use them as a functional-interface value in Java code, usually without an explicit cast. Method references (`Foo::bar`) are not supported.

Basic Forms

A lambda body may be a single expression or a block, and its parameter list may be empty, a single untyped identifier, or a parenthesized list of untyped or typed parameters:

```
Function expr  = x -> x + 1;
Function block = x -> { return x + 1; };

Supplier zeroArg   = () -> 5;
Function oneArg    = x -> x * 2;
BiFunction twoArgs = (a, b) -> a + b;
Function typedArg  = (Integer x) -> x + 10;

expr.apply(1);   // 2
twoArgs.apply(2, 3); // 5
```

As in the examples above, a lambda can usually be assigned directly to a variable typed with the target functional interface, with no

cast needed. This works with ordinary JDK functional interfaces too, so you can pass a lambda straight into APIs like `Collection.forEach()`, `Collections.sort()`, or a `Stream` pipeline:

```
sum = 0;
Arrays.asList(1, 2, 3).forEach( x -> { sum += x; } );

sorted = new ArrayList(Arrays.asList(3, 1, 2));
Collections.sort( sorted, (a, b) -> a - b );

mapped = Arrays.asList(1, 2, 3).stream()
    .map( x -> x * 2 )
    .collect( Collectors.toList() );
```

A lambda's value is opaque - it doesn't commit to a particular interface type - until it is converted to one, by cast, assignment, or overload resolution. Until then it can sit in an untyped or `Object` variable just like any other BeanShell value.

Capturing Variables

A lambda captures its declaring scope *by reference*, not by snapshotting each variable's value the way Java requires captured locals to be effectively final. This means a later change to a captured variable - a global, a method local, or a scripted object's field - is visible the next time the lambda runs:

```
counter = 1;
Supplier getCounter = () -> counter;
counter = 2;
getCounter.get(); // 2
```

> **Note:**
> Because BeanShell does not refresh a for-each loop variable per iteration the way Java does, a lambda created inside a loop captures the *same* variable across iterations, not a fresh one each time:
>
> ```
> for ( String s : list )
>     list2.add( () -> s );  // every lambda here returns list's LAST element
> ```
>
> If you need each lambda to see a different value, assign it to a fresh local inside the loop body first.

Exceptions

An unchecked exception, or a checked exception the target interface's method actually declares, propagates out of a lambda call as itself. Any other checked exception - or a BeanShell script error inside the lambda body - comes out wrapped as `bsh.RuntimeEvalError` instead.

Limitations

A handful of target interface shapes are rejected when a lambda is converted to them, rather than allowed to fail confusingly later:

- Sealed interfaces (JLS 9.8).
- An interface whose single abstract method declares a return or parameter type that isn't accessible to the generated wrapper class (for example, public only inside a JPMS module not exported to unnamed modules).
- A scripted interface that declares or inherits a scripted *default* method - a scripted default can't yet call back into the lambda's own method. An inherited compiled Java default (such as `Predicate.negate()`) is unaffected.
- A raw generic target such as a bare `Supplier` can't be fully type-checked at conversion time, so a Java caller expecting `Supplier<String>` may still see a `ClassCastException` at the point of use.

A lambda converted to a `Serializable` interface serializes normally, but deserializing it throws rather than running - the same rule that already applies to serialized 'this' references - because deserialization has to freshly initialize the named interface.

Special Variables and Values

In addition to the scope modifiers: 'this', 'super', 'global', BeanShell supports a number of pre-defined system variables, "magic" values, and methods.

Special Values

- **$_** - The value of the last expression evaluated. The strange construct for this is drawn from Perl, but the idea exists in many scripting languages. It is useful for getting back the last result when you are working interactively.
- **$_e** - The last uncaught exception object thrown. This is useful in interactive use for retrieving the last exception to inspect it for details.
- **bsh** - The BeanShell root system object, containing system information and variables.
- **bsh.args** - An array of Strings passed as command line arguments to the BeanShell interpreter.
- **bsh.shared** - A special static space which is shared across all interpreter instances. Normally each bsh.Interpreter instance is entirely independent; having its own unique global namespace and settings. bsh.shared is implemented as a static namespace in the bsh.Interpreter class. It was added primarily to support communication among instances for the GUI desktop.
- **bsh.console** - If BeanShell is running in its GUI desktop mode, this variable holds a reference to the current interpreter's console, if it has one.
- **bsh.appletcontext** - If BeanShell is running inside an Applet, the current applet context, if one exists.
- **bsh.cwd** - A String representing the current working directory of the BeanShell interpreter. This is used or manipulated by the cd(), dir(), pwd(), and pathToFile() commands.
- **bsh.show** - A boolean value used by the show() command. It indicates whether results are always printed, for interactive use.
- **bsh.interactive** - A boolean indicating whether this interpreter running in an interactive mode
- **bsh.evalOnly** - A boolean indicating whether this interpreter has an input stream or whether is it only serving as an engine for eval() operations (e.g. for embedded use).

> **Note:**
> The choice of "bsh" for the root system object name was somewhat unfortunate because it conflicts with the current package name for BeanShell (also bsh). This means that if you wish to work with BeanShell classes explicitly from BeanShell scripts (e.g. bsh.Interpreter) you must first import them, e.g.:
>
> ```
> import bsh.Interpreter;
> i=new Interpreter();
> ```

Special Members of 'this' type References

'this' type references have several "magic" members:

- **this.variables** - An array of Strings listing the variables defined in the current method context (namespace).
- **this.methods** - An array of Strings listing the methods defined the current method context (namespace).
- **this.interpreter** - A bsh.Interpreter reference to the currently executing BeanShell Interpreter object.
- **this.namespace** - A bsh.NameSpace reference to the BeanShell NameSpace object of the current method context. See "Advanced Topics".
- **this.caller** - A bsh.This reference to the calling BeanShell method context. See "Variables and Scope Modifiers".
- **this.callstack** - An array of bsh.NameSpace references representing the "call stack" up to the current method context. See "Advanced Topics".

These magic references are primarily used by BeanShell commands.

Undefined Variables

You can test to see if a variable is defined using the special value **void**. For example:

```
if ( foobar == void )
    // undefined
```

You can return a variable to the undefined state using the unset() command:

```
a == void;  // true
a=5;
unset("a"); // note the quotes
a == void;  // true
```

Setting the Command Prompt

Users may set the command line prompt string for use in interactive mode by setting the value of the variable bsh.prompt or by defining the scripted method (or command) getBshPrompt().

If the command or method getBshPrompt() is defined it will be called to get a string to display as the user prompt. For example, one could define the following method to place the current working directory into their command prompt:

```
        getBshPrompt() { return bsh.cwd + " % "; }
```

The default getBshPrompt() command returns the value of the variable bsh.prompt if it is defined or the string "bsh % " if not. If the getBshPrompt() method or command does not exist, throws an exception, or does not return a String, a default prompt of "bsh % " will be used.

BeanShell Commands

BeanShell commands appear to the user as pre-defined methods such as print(), load(), or save(). BeanShell Commands can be implemented as scripted methods or compiled Java classes which are dynamically loaded on demand from the classpath. We'll talk about adding your own commands in the next section "Adding BeanShell Commands".

Tip:
You can easily override any BeanShell command simply by defining the method yourself in your script. For example:

```
print( arg ) {
    System.out.println( "You printed: " + arg );
}
```

If you define the method in the global scope it will apply everywhere. If you define it local to a scripted object it will only apply in that object context.

Commands Overview

This is a high level overview of the BeanShell command set. You can find full documentation for all BeanShell commands in the "BeanShell Commands Documentation" section of this manual. See also the "BshDoc" section which covers javadoc style documentation of BeanShell script files.

Interpreter Modes

The following commands affect general modes of operation of the interpreter.

exit()	Exit the interpreter. (Also Control-D).
show()	Turn on "show" mode which prints the result of every evaluation that is not of void type.
setAccessibility()	Turn on access to private and protected members of Java classes.
server()	Launch the remote access mode, allowing remote access to the interpreter from a web browser or telnet client.
debug()	Turns on debug mode. Note: this is very verbose, unstructured output and is primarily of interest to developers.
setStrictJava()	Turn on "strict Java" mode which enforces Java compatibility by dissallowing loose types and undeclared variables.

Output

The following commands are used for output:

print(), error()	Print output to standard out or standard error. print() always goes to the console, whereas System.out may or may not be captured by a GUI console or servlet.
frame()	Display the AWT or Swing component in a Frame

Source and Evaluation

The following commands are used for evaluation or to run external scripts or applications:

eval()	Evaluate a string as if it were typed in the current scope.
source(), sourceRelative()	Read an external script file into the interpreter and evaluate it in the current scope
	Run an external file in a subordinate interpreter or in a background thread in a subordinate

run(), bg()	interpreter.
exec()	Run a native executable in the host OS

Utilities

The following commands are useful utilities:

javap()	Print the methods and fields of an object, similar to the output of javap
which()	Like the Unix 'which' command for executables. Map the classpath and determine the location of the specified class.
load(), save()	load a serializable object from a file or save one to a file. Special handling is provided for certain objects.
object()	Create an "empty" object context to hold variables; analogous to a Map.

Variables and Scope

The following commands affect the current scope:

clear()	Clear all variables, methods and imports from the current scope.
unset()	Remove a variable from the current scope. (Return it to the "undefined" state).
setNameSpace()	Set the current namespace to a specified scope. Effectively bind the current scope to a new parent scope.

Classpath

The following commands manipulate or access the classpath:

addClassPath(), setClassPath(), getClassPath()	Modify the BeanShell classpath.
reloadClasses()	Reload a class or group of classes.
getClass()	Load a class explicitly taking into account the BeanShell classpath.
getResource()	Get a resource from the classpath.

Files and Directories

The following commands work with files, directories, and the working directory:

cd(), pwd(), dir(), rm(), mv(), cat()	Unix Style file commands.
pathToFile()	Translate a relative path to an absolute path taking into account the BeanShell current working directory.

Desktop and Class Browser

The following commands work with GUI tools:

classBrowser(), browseClass()	Open a class browser window or browse a specific class or object.
desktop()	Launch the BeanShell GUI desktop.
setNameCompletion()	Turn on or off name completion in the GUI console.

Note:
The dir() command is written in Java; primarily as a demonstration of how to do this when desired.

Adding BeanShell Commands

BeanShell Commands are scripted methods or compiled Java classes which are dynamically loaded from the classpath to implement a method. All of the standard commands we discuss in this manual live in the BeanShell JAR file under the path /bsh/commands.

Adding to the set of "prefab" commands supplied with BeanShell is as easy as writing any other BeanShell methods. You simply have to place your script into a file named with the same name as the command and place the file in the classpath. You may then "import" the commands with the importCommands() method.

Command files can be placed anywhere in the BeanShell classpath. You can use even use the addClassPath() or setClassPath() commands to add new command directories or JARs containing commands to your script at any time.

Hello World

For example, let's make a helloWorld() command:

```
// File: helloWorld.bsh
helloWorld() {
    print("Hello World!");
}
```

Place the command file helloWorld.bsh in a directory or JAR in your classpath and import it with the importCommands() command. You can either set the classpath externally for Java or inside of BeanShell with the addClassPath() command. For example, suppose we have placed the file in the path: /home/pat/mycommands/helloWorld.bsh. We could then do:

```
addClassPath("/home/pat");  // If it's not already in our classpath
importCommands("/mycommands");
```

We can now use helloWorld() just like any other BeanShell command.

```
helloWorld(); // prints "Hello World!"
```

importCommands() will accept either a "resource path" style path name or a Java package name. Either one is simply converted to a resource path or Java package name as required to load scripts or compiled BeanShell command classes. A relative path (e.g. "mycommands") is turned into an absolute path by prepending "/". You may import "loose" commands (like unpackaged classes) at the top of the classpath by importing "/".

If for example you have placed BeanShell commands along with your other classes in a Java package called com.xyz.utils in your classpath, you can import those commands with:

```
// equivalent
importCommands("com.xyz.utils");
importCommands("/com/xyz/utils");
```

Imported commands are scoped just like imported classes. So if you import commands in a method or object they are local to that scope.

Overloaded Commands

BeanShell command scripts can contain any number of overloaded forms of the command method, e.g.:

```
// File: helloWorld.bsh
helloWorld() {
        print("Hello World!");
}
helloWorld( String msg ) {
        print("Hello World: "+msg);
}
```

BeanShell will select the appropriate method based on the usual rules for methods selection.

Compiled Commands

You can also implement BeanShell commands as compiled classes instead of scripts if you wish. Your class name must simply be the name of the command (matching case as well) and it must implement one or more static invoke() methods who's signatures match a pattern. The first two arguments of the invoke() method must be the bsh.Interpreter and bsh.CallStack objects that provide context to all BeanShell scripts. Then any number (possibly zero) of arguments, which are the arguments of the command may follow. BeanShell will select the appropriate method based on the usual rules for methods selection.

The dir() command is an example of a BeanShell command that is implemented in Java. Let's look at a snippet from it to see how it implements a pair of invoke() methods for the dir() and dir(path) commands.

```
/**
        Implement dir() command.
*/
public static void invoke( Interpreter env, CallStack callstack )
{
        String dir = ".";
        invoke( env, callstack, dir );
}

/**
        Implement dir( String directory ) command.
*/
public static void invoke(
        Interpreter env, CallStack callstack, String dir )
{
...
}
```

User Defined Commands with invoke()

It is useful to note that the invoke() meta-method which we described in the section "Scripting Interfaces" can be used directly in scope as well as through an object reference and one could use this to load arbitrary commands or implement arbitrary behavior for commands (undefined method calls). For example:

```
invoke( String methodName, Object [] arguments ) {
        print("You invoked the method: "+ methodName );
}

// invoke() will be called to handle noSuchMethod()
noSuchMethod("foo");
```

invoke() is called to handle any method invocations for undefined methods within its scope. In this case we have declared it at the global scope.

Commands Scope

Scripted BeanShell commands are loaded when no existing method matches the command name. When a command script is loaded it is sourced (evaluated) in the 'global' scope of the interpreter. This means that once the command is loaded the methods declared in the command script are then defined in the interpreter's global scope and subsequent calls to the command are simply handled by the those methods as any other scripted method.

> **Note:**
> Note that this means that currently scripted commands may only be loaded once and then they are effectively cached.

Getting the Caller Context

A useful feature of BeanShell for command writers is the 'this.caller' reference, which allows you to create side effects (set or modify variables) in the method caller's scope. For example:

```
fooSetter() {
    this.caller.foo=42;
}
```

The above command has the effect that after running it the variable 'foo' will be set in the caller's scope. e.g.:

```
fooSetter();
```

```
print( foo ); // 42
```

It may appear that we could simply have used the 'super' modifier to accomplish this and in this case it would have worked. However it would not have been correct in general because the 'super' of fooSetter() always points to the same location - the scope in which it was defined. We would like fooSetter() to set the variable in whatever scope it was called from.

To reiterate: The 'super' of a method is always the context in which the method was defined. But the caller may be any context in which the method is used. In the following example, the parent context of foo() and the caller context of foo() are the same:

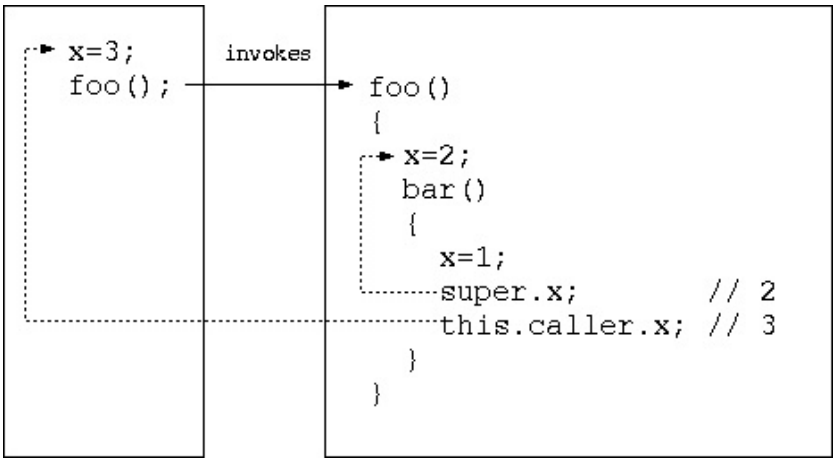
```
foo() { ... }
foo();
```

But this is not always the case, as for bar() in the following example:

```
foo() {
    bar() { ... }
    ...
}

// somewhere
fooObject.bar();
```

The special "magic" field reference: 'this.caller' makes it possible to reach the context of whomever called bar(). The 'this.caller' reference always refers to the calling context of the current method context.



The diagram above shows the foo() and bar() scopes, along with the caller's scope access via 'this.caller'.

This is very useful in writing BeanShell commands. BeanShell command methods are always loaded into the global scope. If you refer to 'super' from your command you will simply get 'global'. Often it is desirable to write commands that explicitly have side effects in the caller's scope. The ability to do so makes it possible to write new kinds of commands that have the appearance of being "built-in" to the language.

A good example of this is the eval() BeanShell command. eval() evaluates a string as if it were typed in the current context. To do this, it sends the string to an instance of the BeanShell interpreter. But when it does so it tells the interpreter to evaluate the string in a specific namespace: the namespace of the caller; using this.caller.

```
    eval("a=5");
    print( a ); // 5
```

The eval() command is implemented simply as:

```
eval( String text ) {
        this.interpreter.eval( text, this.caller.namespace );
}
```

As a novelty, you can follow the call chain further back if you want to by chaining the '.caller' reference, like so:

```
    this.caller.caller...;
```

Or, more generally, another magic reference 'this.callstack' returns an array of bsh.NameSpace objects representing the full call "stack". This is an advanced topic for developers that we'll discuss in another location.

setNameSpace()

In the previous discussion we used the this.caller reference to allow us to write commands that have side effects in the caller's context. This is a powerful tool. But what happens when one command calls another command that intends to do this? That would leave the side effects in the first command's context, not it's original caller. Fortunately this doesn't come up all that often. But there is a general way to solve this problem. That is to use the powerful setNameSpace() method to "step into" the caller's context. After that we may set variables and call methods exactly as if we were in the caller's context (because we are). If all commands did this there would be no need to use the this.caller reference explicitly (indeed, we may make it idiomatic for all commands to do this in the future).

```
myCommand() {
        // "Step into" the caller's namespace.
        setNameSpace( this.caller.namespace );

        // work as if we were in the caller's namespace.
}
```

You can try out the setNameSpace() command with arbitrary object scope's as well. For example:

```
object = object();

// save our namespace
savedNameSpace = this.namespace;

// step into object's namespace
setNameSpace( object.namespace );

// Work in the object's scope
a=1;
b=2;

// step back
setNameSpace( savedNameSpace );

print( object.a ); // 1
print( object.b ); // 2

print( a ); // ERROR! undefined
```

Getting the Invocation Text

You can get specific information about the invocation of a method using namespace.getInvocationLine() and namespace.getInvocationText(). The most important use for this is in support of the ability to write an assert() method for unit tests that automatically prints the assertion text.

```
assert( boolean condition )
{
    if ( condition )
        print( "Test Passed..." );
    else {
        print(
            "Test FAILED: "
            +"Line: "+ this.namespace.getInvocationLine()
            +" : "+this.namespace.getInvocationText()
            +" : while evaluating file: "+getSourceFileInfo()
        );
        super.test_failed = true;
    }
}
```

Working with Dirctories and Paths

BeanShell supports the notion of a *current working directory* for commands that work with files. The cd() command can be used to change the working directory and pwd() can be used to display the current value. The BeanShell current working directory is stored in the variable bsh.cwd.

All commands that work with files respect the working directory, including the following:

- dir()
- source()

- run(),
- cat()
- load()
- save()
- mv()
- rm()
- addClassPath()

pathToFile()

As a convenience for writing your own scripts and commands you can use the pathToFile() command to translate a relative file path to an absolute one relative to the current working directory. Absolute paths are unmodified.

```
absfilename = pathToFile( filename );
```

Path Names and Slashes

When working with path names you can generally just use forward slashes in BeanShell. Java localizes forward slashes to the appropriate value under Windows environments. If you must use backslashes remember to escape them by doubling them:

```
dir("c:/Windows"); // ok
dir("c:\\Windows"); // ok
```

Working With Class Identifiers

You may have noticed that certain BeanShell commands such as javap(), which(), and browseClass() which take a class as an argument can accept any type of argument, including a plain Java class identifier. For example, all of the following are legal:

```
javap( Date.class ); // use a class type directly
javap( new Date() ); // uses class of object
javap( "java.util.Date" ); // Uses string name of class
javap( java.util.Date );  // Use plain class identifier
```

In the last case above we used the plain Java class identifier java.util.Date. In Beanshell this resolves to a bsh.ClassIdentifier reference. You can get the class represented by a ClassIdentifier using the Name.identifierToClass() method. Here is an example of how to work with all of the above, converting the argument to a class type:

```
        import bsh.ClassIdentifier;

    if ( o instanceof ClassIdentifier )
        clas = this.namespace.identifierToClass(o);
    if ( o instanceof String)
        clas = this.namespace.getClass((String)o);
    else if ( o instanceof Class )
        clas = o;
    else
        clas = o.getClass();
```

Working with Iterable Types

In conjunction with the enhanced for-loop added in BeanShell version 1.3 a unified API was added to provide support for iteration over composite types. The bsh.BshIterator interface provides the standard hasNext() and next() methods of the java.util.Iterator interface, but is available in all versions of Java and can be created for all composite types including arrays.

The BeanShell CollectionManager is used to get a BshIterator for an interable object or array. It is a dynamically loaded extension, so it provides support for the java.util.Collections API when available, but does not break compatability for Java 1.1 applications. You can use this in the implementation of BeanShell commands to iterate over Enumeration, arrays, Vector, String, StringBuffer and (when the java.util.collections API is present) Collections and Iterator.

```
cm = CollectionManager.getCollectionManager();
if ( cm.isBshIterable( myObject ) )
{
        BshIterator iterator = cm.getBshIterator( myObject );
        while ( iterator.hasNext() )
                i = iterator.next();
}
```

Strict Java Mode

Note: Strict Java Mode is new and currently breaks some BeanShell tools and APIs when activated. The GUI desktop and most BeanShell commands will not work with strict Java mode enabled. Please see notes at the end of this page

If you are a Java teacher or a student learning the Java language and you would like to avoid any potential confusion relating to BeanShell's use of loose variable types, you can turn on Strict Java Mode. Strict Java Mode is enabled with the the setStrictJava() command. When strict Java mode is enabled BeanShell will require typed variable declarations, method arguments and return types. For example:

```
setStrictJava(true);

int a = 5;

foo=42; // Error! Undeclared variable 'foo'.

bar() { .. } // Error! No declared return type.
```

Class Loading and Class Path Management

BeanShell is capable of some very fine grained and sophisticated class reloading and modifications to the class path. BeanShell can even map the entire class path to allow for automatic importing of classes.

Changing the Class Path

addClassPath(URL | path)

Add the specified directory or archive to the classpath. Archives may be located by URL, allowing them to be loaded over the network.

Examples:

```
addClassPath( "/home/pat/java/classes" );
addClassPath( "/home/pat/java/mystuff.jar" );
addClassPath( new URL("http://myserver/~pat/somebeans.jar") );
```

Note that if you add class path that overlaps with the existing Java user classpath then the new path will effectively reload the classes in that area.

If you add a relative path to the classpath it is evaluated to an absolute path; it does not "move with you".

```
cd("/tmp");
addClassPath("."); // /tmp
```

setClassPath(URL [])

Change the entire classpath to the specified array of directories and/or archives.

This command has some important side effects. It effectively causes all classes to be reloaded (including any in the Java user class path at startup). Please see "Class Reloading" below for further details.

Note: setClassPath() cannot currently be used to make the classpath smaller than the Java user path at startup.

Auto-Importing from the Classpath

As an alternative to explicitly importing class names you may use the following statement to trigger automatic importing:

```
import *;
```

There may be a significant delay while the class path is mapped. This is why auto-importing is not turned on by default. When run

interactively, Bsh will report the areas that it is mapping.

It is only necessary to issue the auto-import command once. Thereafter changes in the classpath via the addClassPath() and setClassPath() commands will remap as necessary.

Note: As of BeanShell 1.1alpha new class files added to the classpath (from outside of BeanShell) after mapping will not be seen in imports.

Reloading Classes

BeanShell provides an easy to use mechanism for reloading classes from the classpath. It is possible in BeanShell to reload arbitrary subsets of classes down to a single class file. However There are subtle issues to be understood with respect to what it means to reload a class in the Java environment. Please see the discussion of class loading detail below. But in a nutshell, it is important that classes which work together be reloaded together at the same time, unless you know what you are doing.

reloadClasses([package name])

The most course level of class reloading is accomplished by issuing the reloadClasses() command with no arguments.

```
reloadClasses();
```

This will effectively reload all classes in the current classpath (including any changes you have made through addClassPath()).

Note: that reloading the full path is actually a light weight operation that simply replaces the class loader - normal style class loading is done as classes are subsequently referenced.

Be aware that any object instances which you have previously created may not function with new objects created by the new class loader. Please see the discussion of class loading details below.

You can also reload all of the classes in a specified package:

```
reloadClasses("mypackage.*");
```

This will reload only the classes in the specified package. The classes will be reloaded even if they are located in different places in the classpath (e.g. if you have some of the package in one directory and some in another).

As a special case for reloading unpackaged classes the following commands are equivalent:

```
reloadClasses(".*")
reloadClasses("<unpackaged>")
```

You can also reload just an individual class file:

```
reloadClasses("mypackage.MyClass")
```

Note: As of alpha1.1 classes contained in archives (jar files) cannot be reloaded. i.e. jar files cannot be swapped.

Mapping the path

Unlike the reloadClases() command which reloads the entire class path, when you issue a command to reload a package or individual class name BeanShell must map some portions of the classpath to find the location of those class files. This operation can be time consuming, but it is only done once. If running in interactive mode feedback will be given on the progress of the mapping.

Loading Classes Explicitly

In order to perform an explicit class lookup by name while taking into account any BeanShell class path modification you must use a replacement for the standard Class.forName() method.

The getClass() command will load a class by name, using the BeanShell classpath. Alternately, you can consult the class manager explicitly:

```
name="foo.bar.MyClass";
c = getClass( name );
c = BshClassManager.classForName( name );  // equivalent
```

Setting the Default ClassLoader

The bsh.Interpeter setClassLoader() and bsh.BshClassManager.setClassLoader() methods can be used to set an external class loader which is consulted for all basic class loading in BeanShell.

BeanShell will use the specified class loader at the same point where it would otherwise use the plain Class.forName(). If no explicit classpath management is done from the script (addClassPath(), setClassPath(), reloadClasses()) then BeanShell will only use the supplied classloader. If additional classpath management is done then BeanShell will perform that in addition to the supplied external classloader. However BeanShell is not currently able to reload classes supplied through the external classloader.

Class Loading in Java

A fundamental Java security proposition is that classes may only be loaded through a class loader once and that classes loaded through different class loaders live in different name spaces. By different name spaces I mean that they are not considered to be of the same type, even if they came from the very same class file.

You can think of this in the following way: When you load classes through a new class loader imagine that every class name is prefixed with the identifier "FromClassLoaderXXX" and that all internal references to other classes loaded through that class loader are similarly rewritten. Now if you attempt to pass a reference to a class instance loaded through another class loader to one of your newly loaded objects, it will not recognize it as the same type of class.

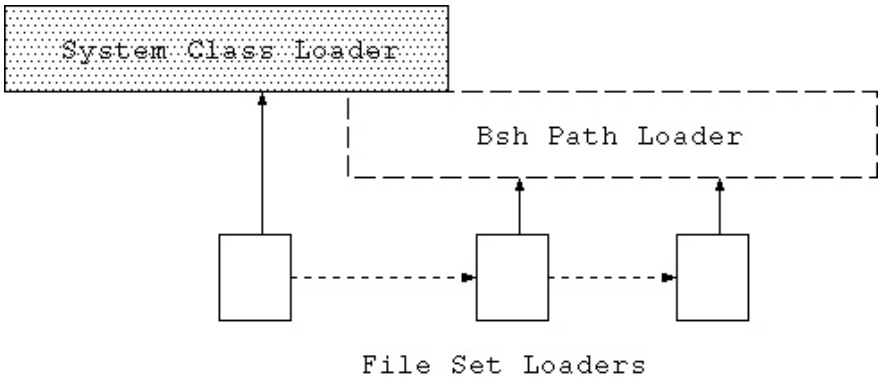
BeanShell works with objects dynamically through the reflection API, so your scripts will not have a problem recognizing reloaded class objects. However any objects which have you already created might not like them.

Class Loading in BeanShell

The following is a discussion of the BeanShell class loader architecture, which allows both course class path extension and fine grained individual class reloading.

Thriftiness - Abiding by the BeanShell thriftiness proposition: no class loading code is exercised unless directed by a command. BeanShell begins with no class loader and only adds class loading in layers as necessary to achieve desired effects.

The following diagram illustrates the two layer class loading scheme:



A "base" class loader is used to handle course changes to the classpath including added path. Unless directed by setClassPath() the base loader will only add path and will not cover existing Java user class path. This prevents unnecessary class space changes for the existing classes.

Packages of classes and individual classes are mapped in sets by class loaders capable of handling discrete files. A mapping of reloaded classes is maintained. The discrete file class loaders will also use this mapping to resolve names outside there space, so when any individual class is reloaded it will see all previously reloaded classes as well.

The BshClassManager knows about all class loader changes and broadcasts notification of changes to registered listeners. BeanShell namespaces use this mechanism to dereference cached type information, however they do not remove existing object instances.

Type caching is extremely important to BeanShell performance. So changing the classloader, which necessitates clearing all type caches, should be considered an expensive operation.

Modes of Operation

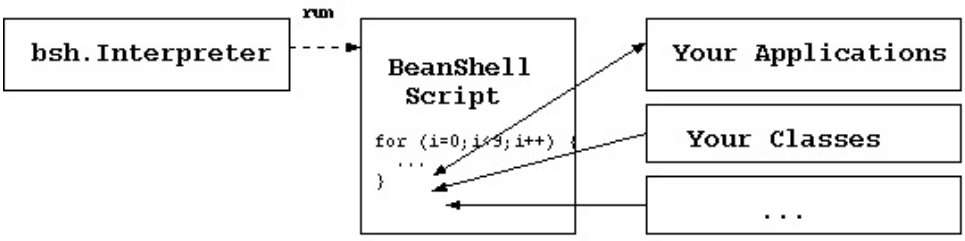
There are currently five basic modes of operation for running BeanShell:

- Standalone scripts
- Embedded in your application
- Remote server mode
- Servlet mode
- Applet mode

We'll outline these in this and the coming sections.

BeanShell is also integrated into a number of other tools and development environments including the Emacs JDE and the NetBeans/Forte for Java IDE. Please see the web site for articles and information about using BeanShell within these third party tools.

Standalone



You can use BeanShell to run scripts from the command line or enter statements interactively by starting the bsh.Interpreter class. (See "Quickstart" for instructions on adding BeanShell to your classpath.)

```
java bsh.Interpreter [ filename ] [ arg ] [ ... ]  // Run a script file
```

There are a few options which can be passed to the Interpreter using Java system properties:

- **outfile** - Send all output to the specified file by redirecting System.out and System.err
- **debug** - Turn on debugging output by setting to true. *Note: this mode is very verbose and unstructured. It is not intended for general use.*
- **trace** - Setting trace to true turns on method tracing. This mode prints each line before it is executed. *Note that this currently prints only top level lines as they are parsed and executed by the interpreter. Trace skips over method executions (including bsh commands) etc. This mode is incomplete. It should be considered experimental.*

Remote

> **Removed in BeanShell 3.0:**
> The bsh.Remote launcher, along with the servlet and remote server engines it talked to, has been removed. This section describes 2.x behavior only; it does not apply to BeanShell 3.x. See "Remote Server Mode" and "BshServlet and Servlet Mode Scripting".

In BeanShell 2.x and earlier, the bsh.Remote launcher was the equivalent of bsh.Interpreter, but ran the specified file in a remote BeanShell engine - either a servlet mode BeanShell engine (BshServlet) or a native server mode remote BeanShell instance.

Interactive Use

One of the most popular uses for BeanShell is, of course, as a "shell" for interactive experimentation and debugging. BeanShell can be run in a GUI desktop mode that offers a number conveniences like command line history, cut & paste, and tools for interactive use such as a simple classbrowser. We'll talk about the GUI in "The BeanShell Desktop" later.

> **Tip:**
> The BeanShell GUI is comprised mostly of a set of BeanShell scripts supplied in the JAR file and launched by the BeanShell desktop() command.

However BeanShell can also be run interactively in plain text on the command line.

```
java bsh.Interpreter   // Run interactively on the command line
```

This is useful for quick "one liners"; however it does not offer creature comforts such as command line history and editing. We

should note that some shells, such as the Windows environment, do command line history and editing automatically - providing these features for BeanShell.

Tip: You can exit from an interactive shell by typing Control-D.

The return statement is ignored in interactive mode (it does not exit the shell).

The .bshrc Init File

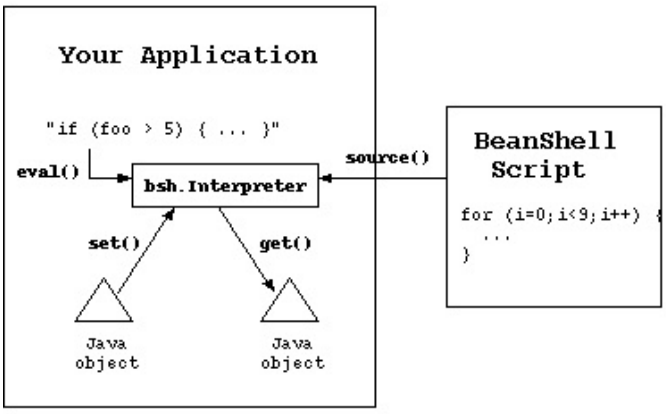
When run interactively, BeanShell looks for a startup file called ".bshrc" in the user's home directory. If the file is found it is sourced into the interactive shell. You can use this to perform setup for your interactive use. For example, to add additional default imports or to toggle on the show() command if you prefer that.

The location of the .bshrc file is determined by the Java system property "user.home", which has different locations under different operating systems. The following table lists common locations:

.bshrc File Location:

Unix	$HOME/.bshrc
Win95/98 single user	C:\Windows\.bshrc
Win98 Multiuser	C:\Windows\Profiles\<username>\.bshrc
NT/2K	C:\Winnt\Profiles\<username>\.bshrc

Embedding BeanShell in Your Application



BeanShell was designed not only as a standalone scripting language - to run scripts from files or on the command line - but to be easily embeddable in and extensible by your applications. When we talk about embedding BeanShell in your application we mean simply that you can use the BeanShell Interpreter in your own classes, to evaluate scripts and work with objects dynamically.

There are a number of reasons you might use BeanShell in this way. Here are a few:

Highly Customizable Applications

You can use BeanShell to make your applications highly customizable by users without requiring them to compile Java classes or even to know all of the Java syntax. During development you can use BeanShell to "factor out" volatile or environmentally dependent algorithms from your application and leave them in the scripting domain while they are under the most intense change. Later it is easy to move them back into compiled Java if you wish because BeanShell syntax is Java syntax.

Macros and Evaluation

BeanShell can be used as a generic language for "macros" or other complex tasks that must be described by a user of your application. For example, the popular JEdit Java editor uses BeanShell to allow users to implement macros for key bindings. This gives user power to customize the behavior of the editor, using as much (or as little) of the full power of Java as desired.

Java with loose variables is a very simple and appealing language; especially because there is already so much Java out there. Even

a non-programmer will be familiar with the name "Java" and more inclined to want to work with it than an arbitrary new language.

BeanShell can also be used to perform dynamic evaluation of complex expressions such as mathematics or computations entered by the user. Why write an arithmetic expression parser when you can let your user enter equations using intermediate variables, operators, and other constructs. If strict control is desired, you can generate the script yourself using your own rules, and still leave the evaluation to BeanShell.

Simplify Complex Configuration Files

Many applications use simple Java properties files or XML for the majority of their runtime configuration. It is very common in the development of a large applications for configuration files like this to become increasingly complex. It can begin in a number of seemingly harmless ways - with the desire to make "cross references" inside the config files (XML supports this nicely). Then comes the desire to do something like variable substitution - which introduces some new syntax such as "${variable}" and usually a second pass in the parsing stage. Usually, at some point, integration with Java forces the introduction of class names into the mix. The configuration files soon want to start assigning parameters for object construction. Ultimately what you'll discover is that you are creating your own scripting language - and one that is probably not as easy to read as plain old Java.

BeanShell solves the problem of complex configuration files by allowing users to work not only with simple properties style values (loose variable assignment) but also to have the full power of Java to construct objects, arrays, perform loops and conditionals, etc. And as we'll see, BeanShell scripts can work seamlessly with objects from the application, **without** the need to turn them into strings to cross the script boundary.

The BeanShell Core Distribution

Beanshell is fairly small, even in its most general distribution. The full JAR with all utilities weighs in at about 250K. But BeanShell is also distributed in a componentized fashion, allowing you to choose to add only the utilities and other pieces that you need. The core distribution includes only the BeanShell interpreter and is currently about 130K. *We expect this size to drop in the future with improvements in the parser generator.* Any significant new features will always be provided in the form of add-on modules, so that the core language can remain small.

More and more people are using BeanShell for embedded applications in small devices. We have reports of BeanShell running everywhere from palm-tops to autonomous buoys in the Pacific ocean!

> **Note:**
> BeanShell 3.0 is on Maven Central as `io.github.beanshell:bsh`. The 3.0.0b1 JAR on GitHub releases was built as `org.beanshell:bsh`; on Central it is under `io.github.beanshell` too.

Calling BeanShell From Java

Invoking BeanShell from Java is easy. The first step is to create in instance of the bsh.Interpreter class. Then you can use it to evaluate strings of code, source external scripts. You can pass your data in to the Interpreter as ordinary BeanShell variables, using the Interpreter set() and get() methods.

In "QuickStart" we showed a few examples:

```
import bsh.Interpreter;

Interpreter i = new Interpreter();  // Construct an interpreter
i.set("foo", 5);                    // Set variables
i.set("date", new Date() );

Date date = (Date)i.get("date");    // retrieve a variable

// Eval a statement and get the result
i.eval("bar = foo*10");
System.out.println( i.get("bar") );

// Source an external script file
i.source("somefile.bsh");
```

The default constructor for the Interpreter assumes that it is going to be used for simple evaluation. Other constructors allow you to set up the Interpreter to work with interactive sources including streams and the GUI console.

> **Tip:**
> As of BeanShell 3.0 you can call `interpreter.setOutputFile(path)` to redirect just that one interpreter's output and error streams to a file, without touching System.out/System.err for the rest of the JVM. The older static Interpreter.redirectOutputToFile() still exists but, despite its name, always redirects the JVM-wide system streams; it is

deprecated in favor of the new instance method.

eval()

The Interprete eval() method accepts a script as a string and interprets it, optionally returning a result. The string can contain any normal BeanShell script text with any number of Java statements. The Interpreter maintains state over any number of eval() calls, so you can interpret statements individually or all together.

> **Note:**
> It is not necessary to add a trailing ";" semi-colon at the end of the evaluated string. BeanShell always adds one at the end of the string.

The result of the evaluation of the last statement or expression in the evaluated string is returned as the value of the eval(). Primitive types (e.g int, char, boolean) are returned wrapped in their primitive wrappers (e.g. Integer, Character, Boolean). If an evaluation of a statement or expression yields a "void" value; such as would be the case for something like a for-loop or a void type method invocation, eval() returns null.

```
Object result = i.eval( "long time = 42; new Date( time )" ); // Date
Object result = i.eval("2*2");  // Integer
```

You can also evaluate text from a java.io.Reader stream using eval():

```
reader = new FileReader("myscript.bsh");
i.eval( reader );
```

EvalError

The bsh.EvalError exception is the general exception type for an error in evaluating a BeanShell script. Subclasses of EvalError - ParseException and TargetError - indicate the specific conditions where a textual parsing error was encountered or where the script itself caused an exception to be generated.

```
try {
    i.eval( script );
} catch ( EvalError e ) {
    // Error evaluating script
}
```

You can get the error message, line number and source file of the error from the EvalError with the following methods:

String getErrorText() {
int getErrorLineNumber() {
String getErrorSourceFile() {

ParseException

ParseException extends EvalError and indicates that the exception was caused by a syntactic error in reading the script. The error message will indicate the cause.

TargetError

TargetError extends EvalError and indicates that the exception was not related to the evaluation of the script, but caused the by script itself. For example, the script may have explicitly thrown an exception or it may have caused an application level exception such as a NullPointer exception or an ArithmeticException.

The TargetError contains the "cause" exception. You can retrieve it with the getTarget() method.

```
try {
    i.eval( script );
} catch ( TargetError e ) {
    // The script threw an exception
    Throwable t = e.getTarget();
    print( "Script threw exception: " + t );
```

```
} catch ( ParseException e ) {
    // Parsing error
} catch ( EvalError e ) {
    // General Error evaluating script
}
```

source()

The Interpreter source() method can be used to read a script from an external file:

```
i.source("myfile.bsh");
```

The Interpreter source() method may throw FileNotFoundException and IOException in addition to EvalError. Aside from that source() is simply and eval() from a file.

set(), get(), and unset()

As we've seen in the examples thus far, set() and get() can be used to pass objects into the BeanShell interpreter as variables and retrieve the value of variables, respectively.

It should be noted that get() and set() are capable of evaluation of arbitrarily complex or compound variable and field expression. For example:

```
import bsh.Interpreter;
i=new Interpreter();

i.eval("myobject=object()" );
i.set("myobject.bar", 5);

i.eval("ar=new int[5]");
i.set("ar[0]", 5);

i.get("ar[0]");
```

The get() and set() methods have all of the evaluation capabilities of eval() except that they will resolve only one variable target or value and they will expect the expression to be of the appropriate resulting type.

The deprecated setVariable() and getVariable() methods are no longer used because the did not allow for complex evaluation of variable names

You can use the unset() method to return a variable to the undefined state.

Getting Interfaces from Interpreter

We've talked about the usefulness of writing scripts that implement Java interfaces. By wrapping a script in an interface you can make it transparent to the rest of your Java code. As we described in the "Interfaces" section earlier, within the BeanShell interpreter scripted objects automatically implement any interface necessary when they are passed as arguments to methods requiring them. However if you are going to pass a reference outside of BeanShell you may have to perform an explicit cast inside the script, to get it to manufacture the correct type.

The following example scripts a global actionPerformed() method and returns a reference to itself as an ActionListener type:

```
// script the method globally
i.eval( "actionPerformed( e ) { print( e ); }");

// Get a reference to the script object (implementing the interface)
ActionListener scriptedHandler =
    (ActionListener)i.eval("return (ActionListener)this");

// Use the scripted event handler normally...
new JButton.addActionListener( scriptedHandler );
```

Here we have performed the explicit cast in the script as we returned the reference. (And of course we could have used the standard Java anonymous inner class style syntax as well.)

An alternative would have been to have used the Interpreter getInterface() method, which asks explicitly for the global scope to be cast to a specific type and returned. The following example fetches a reference to the interpreter global namespace and cast it to the specified type of interface type.

```
Interpreter interpreter = new Interpreter();
// define a method called run()
interpreter.eval("run() { ... }");

// Fetch a reference to the interpreter as a Runnable
Runnable runnable =
    (Runnable)interpreter.getInterface( Runnable.class );
```

The interface generated is an adapter (as are all interpreted interfaces). It does not interfere with other uses of the global scope or other references to it. We should note also that the interpreter does *not* require that any or all of the methods of the interface be defined at the time the interface is generated. However if you attempt to invoke one that is not defined you will get a runtime exception.

Multiple Interpreters vs. Multi-threading

A common design question is whether to use a single BeanShell interpreter or multiple interpreter instances in your application.

The Interpreter class is, in general, thread safe and allows you to work with threads, within the normal bounds of the Java language. BeanShell does not change the normal application level threading issues of multiple threads from accessing the same variables: you still have to synchronize access using some mechanism if necessary. However it is legal to perform multiple simultaneous evaluations. You can also write multi-threaded scripts within the language, as we discussed briefly in "Scripting Interfaces".

Since working with multiple threads introduces issues of synchronization and application structure, you may wish to simply create multiple Interpreter instances. BeanShell Interpreter instances were designed to be very light weight. Construction time is usually negligible and in simple tests, we have found that it is possible to maintain hundreds (or even thousands) of instances.

There are other options in-between options as well. It is possible to retrieve BeanShell scripted objects from the interpreter and "re-bind" them again to the interpreter. We'll talk about that in the next section. You can also get and set the root level bsh.NameSpace object for the entire Interpreter. The NameSpace is roughly equivalent to a BeanShell method context. Each method context has an associated NameSpace object.

> **Tip:**
> You can clear all variables, methods, and imports from a scope using the clear() command.

Note: at the time of this writing the synchronized language keyword is not implemented. This will be corrected in an upcoming release.

See also "The BeanShell Parser" for more about performance issues.

Scripting via javax.script (JSR223)

As of BeanShell 2.0 you can also invoke BeanShell through the standard Java scripting API (JSR223, the `javax.script` package), instead of using the bsh.Interpreter class directly. BeanShell registers itself as a script engine under the names "beanshell", "bsh" and "java", and for the file extensions ".bsh" and ".java", so the standard ScriptEngineManager will find it automatically as long as the BeanShell JAR is on your classpath:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("beanshell");

engine.put("foo", 5);
engine.eval("bar = foo * 10;");
System.out.println( engine.get("bar") );
```

BeanShell keeps all of its variable, type and other interpreter state for a given ScriptContext in a BeanShell NameSpace embedded in the context, so the engine is fully pluggable with respect to the spec while still preserving full interpreter state between eval() calls.

> **Note:**
> If you would rather work directly against the interpreter (for example to use eval()'s return value or the get()/set() shortcuts described above) use the bsh.Interpreter class directly as described earlier in this section. The JSR223 engine is mainly useful when your application is already written against the generic javax.script API and you want to be able to swap scripting languages.

Serializing Interpreters and Scripted Objects

The BeanShell Interpreter is serializable, assuming of course that all objects referenced by variables in the global scope are also serializable. So you can save the entire static state of the interpreter by serializing it and storing it. Note that serializing the Intepreter does not "freeze" execution of BeanShell scripts in any sense other than saving the current state of the variables. In general if you serialize an Interpreter while it is executing code the results will be undetermined. De-serializing an interpreter does not automatically restart method executions; it simply restores state.

> **Note:**
> There is serious Java bug that affects BeanShell serializability in Java versions prior to 1.3. When using these versions of Java the primitive type class identifiers cannot be de-serialized. See the FAQ for a workaround.

It is also possible to serialize individual BeanShell scripted objects ('this' type references and interfaces to scripts). The same rules apply. One thing to note is that by default serializing a scripted object context will also serialize all of that object's parent contexts up to the global scope - effectively serializing the whole interpreter.

To detach a scripted object from its parent namespace you can use the namespace prune() method:

```
// From BeanShell
object.namespace.prune();

// From Java
object.getNameSpace().prune();
```

To bind a BeanShell scripted object back into a particular method scope you can use the bind() command:

```
// From BeanShell
bind( object, this.namespace );

// From Java
bsh.This.bind( object, namespace, interpreter );
```

The bind() operation requires not only the namespace (method scope) into which to bind the object, but an interpreter reference as well. The interpreter reference is the "declaring interpreter" of the object and is used for cases where there is no active interpreter - e.g. where an external method call from compiled Java enters the object.

The BeanShell save() command which serializes objects recognize when you are trying to save a BeanShell scripted object (a bsh.This reference) type and automatically prune()s it from the parent namespace, so that saving the object doesn't drag along the whole interpreter along for the ride. Similarly, load() binds the object to the current scope.

Remote Server Mode

> **Removed in BeanShell 3.0:**
> The built-in remote server (the server() command, bsh.util.Httpd, bsh.util.Sessiond) and the remote console applets have been removed. None of it was authenticated: server() opened an unauthenticated BeanShell session on the network. This page describes 2.x behavior only; it does not apply to BeanShell 3.x.

Remote server mode let you access a BeanShell Interpreter inside of a remote VM in BeanShell 2.x and earlier. With remote server mode activated you could telnet into the running application and type commands at the BeanShell shell prompt, or use a web browser to bring up a remote GUI console.

BshServlet and Servlet Mode Scripting

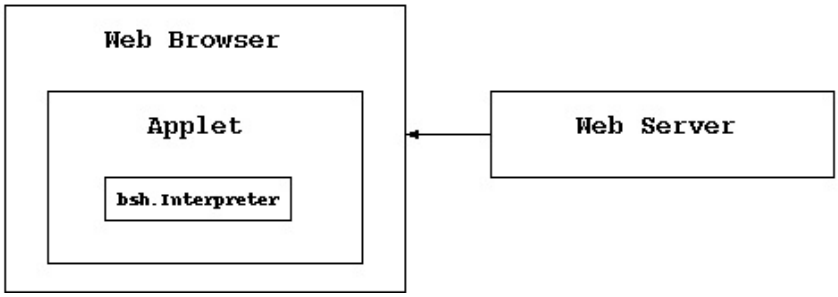
> **Removed in BeanShell 3.0:**
> The bsh.servlet package, BshServlet, and the bsh.Remote launcher have been removed. The servlet evaluated a script passed in as a request parameter with no authentication. This page describes 2.x behavior only; it does not apply to BeanShell 3.x.

BshServlet was a simple servlet that could be used to evaluate BeanShell scripts inside of an application server or servlet container in BeanShell 2.x and earlier. BshServlet accepted BeanShell scripts via the POST method, evaluated them capturing output

(optionally including standard out and standard error) and returned the results.

The BeanShell Demo Applet

Removed: Java applet demos are not available in current browsers. The description below is retained for historical reference. See the BeanShell examples instead.



The BeanShell Applet is primarily for demonstration and educational purposes. It allows you to experiment with BeanShell live, directly in your web browser. You can try the applet live at the following locations:

Swing JConsole Applet

A Swing enabled JConsole usable with the Java plug-in (or other swing capable browser:

BeanShell examples

AWT Console Applet

A minimal (not very good) AWT based console that should work in any browser.

BeanShell examples

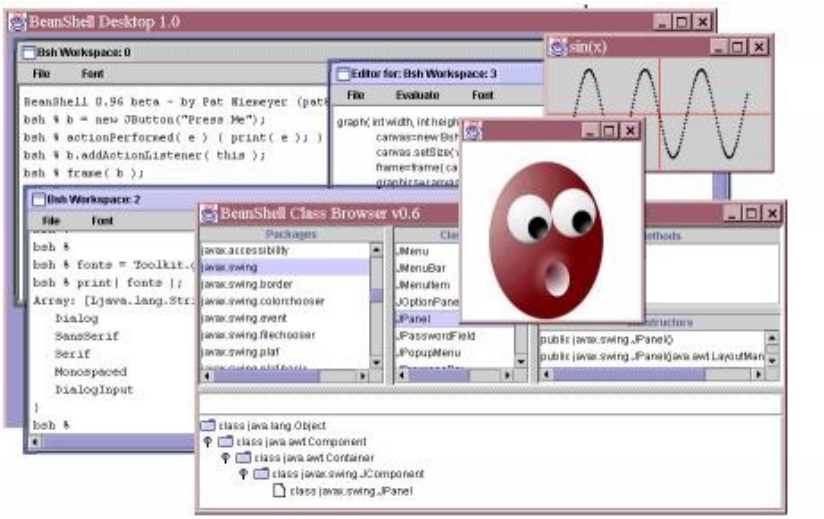
Signed JConsole Applet

There are many additional security restrictions on Applets and this limits what you can do with BeanShell in this mode. For unrestricted access try the signed version of the applet here. It requires the Java 1.4 plug-in to function.

A Swing enabled JConsole as a signed applet with the Java plug-in (or other swing capable browser). The signed applet will allow you unrestricted access to your environment through scripting.

BeanShell examples

BeanShell Desktop



The BeanShell Desktop is a simple GUI environment that provides multiple bsh shell windows (MDI), a simple text editor, and a

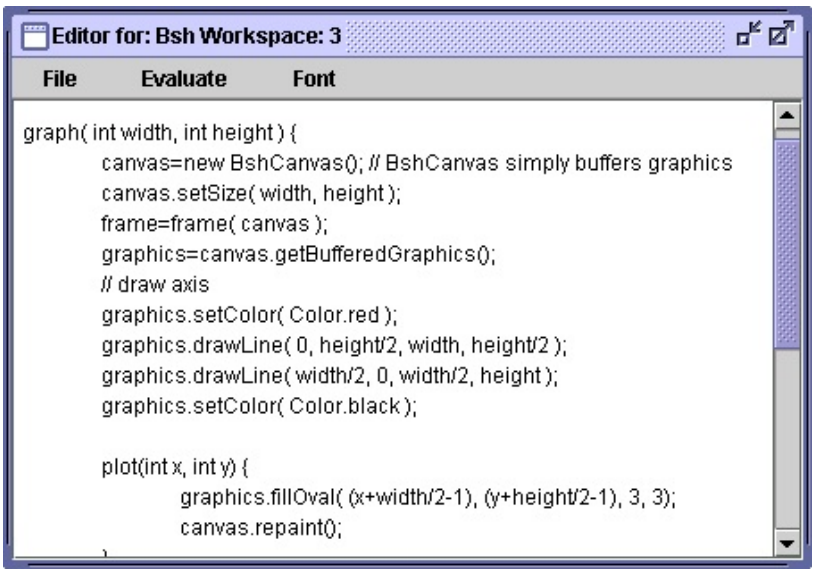
simple class browser. The desktop is mostly implemented by BeanShell scripts, launched by the desktop() command.

Shell Windows

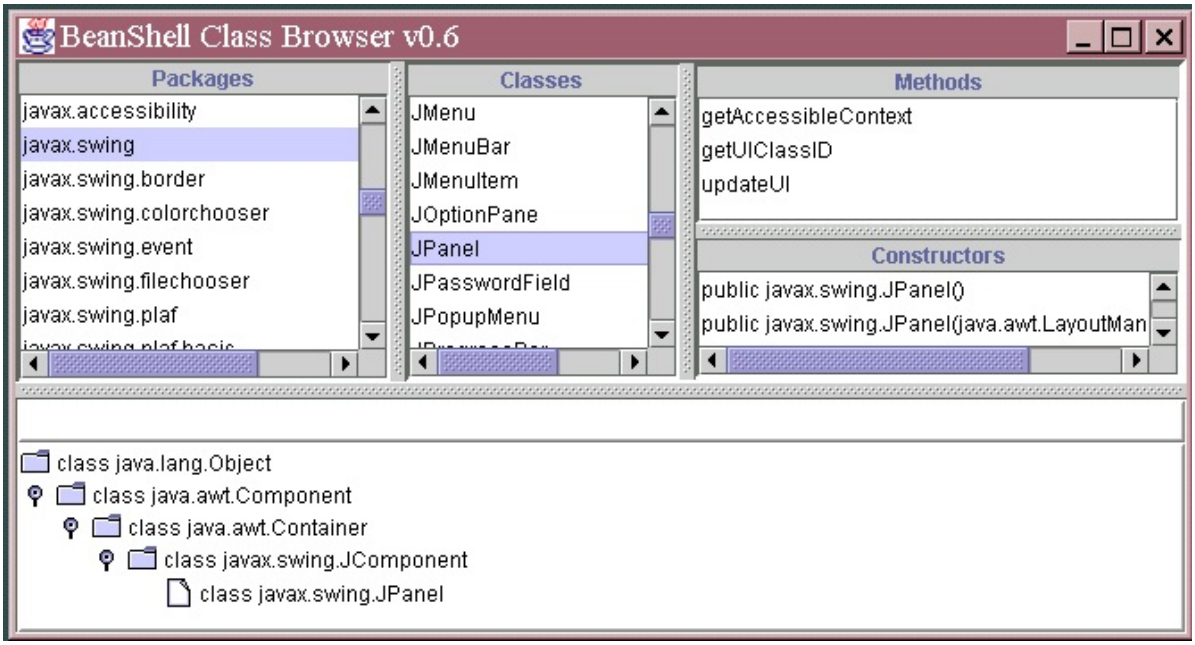


The bsh console windows provide simple command line editing, history, cut & paste, and variable and class name completion.

Editor Windows



The Class Browser



BshDoc - Javadoc Style Documentation

BshDoc requires JDK1.4 and BeanShell 1.2b6 or greater to run

BshDoc is a BeanShell script that supports supports javadoc style documentation of BeanShell scripts. BshDoc parses one or more BeanShell script files for method information and javadoc style formal comments. Its output is an XML description of the files, containing all of the method signature and comment information. An XSL stylesheet, bshcommands.xsl, supplied with the user manual source, can be used to render the XML to a nicely indexed HTML document describing BeanShell commands.

The bshdoc.bsh script is currently distributed with the source distribution. An example of the styled output of this command is the "BeanShell Commands Documentation" section of this user manual. That section is automatically generated as part of the build process by running bshdoc.bsh on bsh/commands/*.bsh. See the source distribution Ant build file for an example of how to do this and the user manual Ant build file for an example of using a stylesheet to build your documents.

BshDoc Comments

BshDoc comments look just like JavaDoc comments and may include HTML markup and javadoc style @tags. If you wish to use the associates XSL stylesheet, you should use well formed XHTML for you documentation. (Always close tags, etc.). e.g.

```
/**
    This is a javadoc style comment.
    <pre>
        <b>Here is some HTML markup.</b>
        <p/>
        Here is some more.
    </pre>

    @author Pat Niemeyer
*/
```

Javadoc style @tags are parsed by bshdoc for inclusion in the XML output. Currently they are only recognized at the start of a line and they terminate the comment. (i.e. they must come at the end of the comment).

BshDoc identifies two kinds of Javadoc style comments: File Comments and Method Comments. Method comments are comments that appear immediately before a method declaration with no statements intervening. File comments are comments that appear as the first statement of a script **and** are not method comments. If a comment appears as the first statement in a script and is also immediately followed by a method declaration it is considered a method comment.

BshDoc XML Output

To use BshDoc, run the bshdoc.bsh script on one or more BeanShell script files:

```
java bsh.Interpreter bshdoc.bsh myfile.bsh [ myfile2.bsh ] [ ... ] > output.xml
```

The output goes to standard out. It looks something like this:

```
<!-- This file was auto-generated by the bshdoc.bsh script -->
<BshDoc>
  <File>
    <Name>foo</Name>
    <Method>
      <Name>doFoo</Name>
      <Sig>doFoo ( int x )</Sig>
      <Comment>
        <Text>&lt;![CDATA[ doFoo() method comment. ]]&gt;</Text>
        <Tags>
        </Tags>
      </Comment>
    </Method>
    <Comment>
        <Text>&lt;![CDATA[ foo file comment. ]]&gt;</Text>
        <Tags>
        </Tags>
    </Comment>
  </File>
</BshDoc>
```

The bshcommands.xsl stylesheet

The bshcommands.xsl stylesheet can be used to render the output of bshdoc.bsh to an indexed HTML page describing BeanShell commands.

The bshcommands.xsl stylesheet is intended for scripts that serve as BeanShell commands. These are script files containing one or more overloaded methods which have the same name as the filename containing them. The BshDoc script produces a complete description of any BeanShell script file. However the supplied bshcommands.xsl stylesheet does not necessarily use all of this information. Specifically, it does not present all individual method comments. Instead it tries to identify the comments pertaining to the command, based upon the file name. It (the XSL stylesheet) applies some logic to choose either the single File Comment if it exists or the Method Comment of the first method matching the filename. Another stylesheet could (and will) be easily created for more general BeanShell file documentation. Please check the web site for updates.

Method signatures displayed for methods can be overridden for the bshcommands.xsl stylesheet by explicitly supplying them in special javadoc @method tags within a Method Comment. For example you might do this to provide a more verbose description for loosely typed arguments to a BeanShell command. The bshcommands.xsl stylesheet will use the @method tag signatures in lieu of autogenerated ones when they are present. So you can also use this tag to determine exactly which methods from a file are listed if you wish. e.g.

```
/**
    BshDoc for the foo() command.
    Explicitly supply the signature to be displayed for the foo() method.

    @method foo( int | Integer ) and other text...
*/
foo( arg ) { ... }
```

Tip:
BshDoc uses the bsh.Parser API to parse the BeanShell script files without actually running them. bshdoc.bsh is not very complex. Take a look at it to learn how to use the parser API.

The BeanShell Parser

This BeanShell parser class bsh.Parser is used internally by the BeanShell Interpreter. It is responsible for the lexical parsing of the input text, the application of the grammar structure, and the building of an internal representation of the BeanShell script file called an "abstract syntax tree" (AST).

The Parser just analyzes the language syntax. It knows only how to parse the structure of the language - it does not interpret names, or execute methods or commands. You can use the Parser directly if you have a need to analyze the structure of BeanShell scripts or Java methods and statements in general.

Validating Scripts With bsh.Parser

You can use the Parser class from the command line to do basic structural validation of BeanShell files without actually executing them. e.g.

```
java bsh.Parser [ -p ] file [ file ] [ ... ]
```

The -p option causes some of the abstract syntax to be printed.

The parser will detect any syntax errors in the script and print an error. Note again that names, imports, and string evaluations are analyzed only for syntax - not content or meaning.

Parsing and Performance

It is useful to have a high level understanding how BeanShell works with scripts to understand performance issues.

The first time a script is read or sourced into an interpreter, BeanShell uses the parser to parse the script internally to an AST. The AST consists of Java object representations of all of the language structures and objects. The AST consists of Java classes, but is *not* the same as compiled Java code. When the script is "executed" BeanShell steps through each element of the AST and tells it to perform whatever it does (e.g. a variable assignment, for-loop, etc.). This execution of the ASTs is generally much faster than the original parsing of the text of the method. It is really only limited by the speed of the application calls that it is making, the speed of the Java reflection API, and the efficiency of the implementation of the structures in BeanShell.

When parsing "line by line" through a BeanShell script the ASTs are routinely executed and then thrown away. However the case of a BeanShell method declaration is different. A BeanShell method is parsed only once: when it is declared in the script. It is then stored in the namespace like any variable. Successive invocations of the method execute the ASTs again, but do not re-parse the original text.

This means that successive calls to the same scripted method are as fast as possible - much faster than re-parsing the script each time. You can use this to your advantage when running the same script many times simply by wrapping your code in the form of a BeanShell scripted method and executing the method repeatedly, rather than sourcing the script repeatedly. For example:

```
// From Java
import bsh.Interpreter;
i=new Interpreter();

// Declare method or source from file
i.eval("foo( args ) { ... }");

i.eval("foo(args)"); // repeatedly invoke the method
i.eval("foo(args)");
...
```

In the above example we defined a method called foo() which holds our script. Then we executed the method repeatedly. The foo() method was parsed only once: when its declaration was evaluated. Subsequent invocations simply execute the AST.

Parsing Scripts Procedurally

If you are willing to learn about the BeanShell abstract syntax tree classes you can use the Parser to parse a BeanShell script into its ASTs like this:

```
in=new FileReader("somefile.bsh");
Parser parser = new Parser(in);
while( !(eof=parser.Line()) ) {
    SimpleNode node = parser.popNode();
    // Use the node, etc. (See the bsh.BSH* classes)
    ...
}
```

To learn more about the abstract syntax tree please download the source distribution and consult the source documentation.

Tip:
The BshDoc bshdoc.bsh script uses the parser to extract method signatures and comments from a BeanShell file. Check it out for a more realistic example.

Note: Many components of the AST classes are not public at this time. Use setAccessibility(true) to access them.

Using JConsole



The bsh.util.JConsole is a light weight graphical shell console window, with simple command editing and history capabilities. BeanShell uses the JConsole for the GUI desktop mode.

> **Tip:**
> As of BeanShell 3.0 you can preload command history with `JConsole.addHistory(String)`, so an embedding application can seed lines that the user recalls with the up arrow before they type anything.

You can use the JConsole to provide an interactive BeanShell prompt in your own applications. You are free to use the JConsole for your own purposes outside of BeanShell as well! It is a fairly generic shell window easily attached to any kind of streams or through the simple console interface.

JConsole is a Swing component. Embed it in your application as you would any other swing component. For example:

```
JConsole console = new JConsole();
myPanel.add(console);
```

You can connect an Interpreter to the console by specifying it in the Interpreter constructor, like so:

```
Interpreter interpreter = new Interpreter( console );
new Thread( interpreter ).start(); // start a thread to call the run() method
```

Or you can connect the JConsole to the Interpreter directly with Interpreter setConsole().

For external use, JConsole can supply a PrintWriter through its getOut() method and has a full suite of direct print() methods.

> **Tip:**
> When interacting with any Swing component from outside the Java event handling thread, use the Swing thread safety facilities: SwingUtilities.invokeNow() and invokeLater().

ConsoleInterface

JConsole implements the bsh.ConsoleInterface interface, which defines how the Interpreter interacts with a console object. To the interpreter a console is simply a set of I/O streams with some optimized print methods:

Reader getIn();
PrintStream getOut();
PrintStream getErr();
void println(String s);
void print(String s);
void error(String s);

Any object that implements this interface can be attached to the Interpreter as a GUI console.

The bsh.util.GUIConsoleInterface extends the ConsoleInterface and adds methods for printing a string with a color attribute, supplying wait feedback (the wait cursor) and name completion support. JConsole implements this interface and it is used indirectly via BeanShell commands when it is detected.

AWTConsole

The bsh.util.AWTConsole is a legacy implementation of the GUI Console using AWT instead of Swing. This console does work, but it is not as slick or pretty as the JConsole. The primary reason it is still here is to support remote access from generic web browsers using only Java 1.1.

Reflective Style Access to Scripted Methods

The following examples show how to work with BeanShell methods dynamically from within scripts, using the equivalent of reflective style access in Java. This is an advanced topic primarily of interest to developers who wish to do tight integration of BeanShell scripts with their application environment.

eval()

The simplest form of reflective style access to scripts is through the eval() command. With eval() you can evaluate any text just as if it had appeared in the current scope. For example:

```
eval("a=5;");
print( a ); // 5
```

So, if you know the signature (argument types) of a method you wish to work with you can simply construct a method call as a string and evaluate it with eval() as in the following:

```
// Declare methods foo() and bar( int, String )
foo() { ... }
bar( int arg1, String arg2 ) { ... }

// Invoke a no-args method foo() by its name using eval()
name="foo";
// invoke foo() using eval()
eval( name+"()");

// Invoke two arg method bar(arg1,arg2) by name using eval()
name="bar";
arg1=5;
arg2="stringy";
eval( name+"(arg1,arg2)");
```

You can get the names of all of the methods defined in the current scope using the 'this.methods' magic reference, which returns an array of Strings:

```
// Print the methods defined in this namespace
print( this.methods );
```

We'll talk about more powerful forms of method lookup in a moment.

invokeMethod()

You can explicitly invoke a method by name with arguments through a 'this' type reference using the invokeMethod() method:

```
this.invokeMethod( "bar", new Object [] { new Integer(5), "stringy" } );
```

Arguments are passed as an array of objects. Primitive types must be wrapped in their appropriate wrappers. BeanShell will select among overloaded methods using the standard Java method resolution rules. (JLS 15.11.2).

Method Lookup

The previous section showed how to invoke a method by name when we know the argument types. Of course, in general we'd like to be able to find out what methods are defined in the current script or to look up a method by its signature.

You can get "handles" to all of the methods defined in a context using the namespace getMethods() method. getMethods() returns an array of bsh.BshMethod objects, which are wrappers for the internally parsed representation of BeanShell scripted methods:

```
foo() { ... }
foo( int a ) { ... }
bar( int arg1, String arg2 ) { ... }

print ( this.namespace.getMethods() );

// Array: [Lbsh.BshMethod;@291aff {
//   Bsh Method: bar
//   Bsh Method: foo
//   Bsh Method: foo
// }
```

We'll talk about what you can do with a BshMethod in a moment.

Alternately, you can use the namespace getMethod() method to search for a specific method signature. The method signature is a set of argument types represented by an array of Classes:

```
name="bar";
signature = new Class [] { Integer.TYPE, String.class };

// Look up a method named bar with arg types int and String
bshMethod = this.namespace.getMethod( name, signature );

print("Found method: "+bshMethod);
```

Tip:
The Java reflection API uses special class values to represent primitive types such as int, char, an boolean. These types are static fields in the respective primitive wrapper classes. e.g. Integer.TYPE, Character.TYPE, Boolean.TYPE.

In the above snippet we located the bar() method by its signature. If there had been overloaded forms of bar() getMethod() would have located the most specific one according to the standard Java method resolution rules (JLS 15.11.2). The result of the lookup is a bsh.BshMethod object, as before.

BshMethod

You can inspect a BshMethod object to determine its method name and argument types:

```
name = bshMethod.getName();
Class [] types = bshMethod.getArgumentTypes();
Class returnType = bshMethod.getReturnType();
```

To invoke the BshMethod, call its invoke() method, passing an array of arguments, an interpreter reference, and a "callstack" reference.

```
// invoke the method with arg
bshMethod.invoke( new Object [] { new Integer(1), "blah!" },
    this.interpreter, this.callstack );
```

For the interpreter and callstack references you can simply pass along the current context's values via 'this.interpreter' and 'this.callstack', as we did above. The arguments array may be null or empty for no arguments.

Uses

Why would anyone want to do this? Well, perhaps you are sourcing a script created by a user and want to automatically begin using methods that they have defined. Perhaps the user is allowed to define methods to take control of various aspects of your application. With the tools we've described in this section you can list the methods they have defined and invoke them dynamically.

Executable scripts under Unix

You can use BeanShell for writing scripts as you would any other shell under many Unixs:

```
#!/usr/java/bin/java bsh.Interpreter

print("foo");
```

However some flavors of Unix are more picky about what they will allow as a shell program. For those you can use the following hack to make your BeanShell scripts executable.

```
#!/bin/sh
# The following hack allows java to reside anywhere in the PATH.
//bin/true; exec java bsh.Interpreter "$0" "$@"

print("foo");
```

The above trick presumes that /bin/true exists on your system and that //bin is the same as /bin. The // causes BeanShell to ignore the line.

The above has been tested on Solaris. It does not seem to work under Cygwin.

OSX

For OSX the path is a bit different:

```
#!/Library/Java/home/bin/java bsh.Interpreter

print("foo");
```

On OSX /usr/bin/java is itself a shell script, which unfortunately won't work out-of-the-box.

BSF Bean Scripting Framework

BSF is the Apache "Bean Scripting Framework". It is generic framework that allows many scripting languages to be plugged into an application. It shields the application from knowledge of how to invoke the scripting languages and their APIs, via adapter "engines".

BeanShell supports the BSF API by providing the necessary adapter. This means that BeanShell can be used as a scripting language for any BSF 2.3 capable application simply by dropping the bsh JAR file into the classpath.

Prior to version 2.3, BSF was maintained by IBM. To get BeanShell to work with older versions of BSF you must use the older bsh-bsf-1.2x.jar file which includes the adapter class for the previous ibm packaged BSF API. You must also explicitly register the BeanShell adapter with older versions of BSF. Here is an example of how to do that:

```
import com.ibm.bsf.*;

// register beanshell with the BSF framework
String [] extensions = { "bsh" };
BSFManager.registerScriptingEngine(
    "beanshell", "bsh.util.BeanShellBSFEngine", extensions );
```

See http://jakarta.apache.org/bsf/ and http://oss.software.ibm.com/developerworks/projects/bsf for more information about BSF.

Ant

This section needs to be updated. I'm not sure what versionf of Ant use the new and which use the old BSF API.

Ant 1.5+ has explicit support for BeanShell as a BSF scripting language. The BeanShell JAR file includes the necessary BSF adapter. You must simply specify language="beanshell" in your script tags.

Installation

To use BeanShell within Ant you must do two things:

1. Add the BSF bsf.jar file to ANT_HOME/lib or the classpath.

2. Add the BeanShell bsh.jar file to ANT_HOME/lib or the classpath.

You can then run scripts from a file, or in-line like so:

```
<project name="testbsh" default="runscript" basedir=".">
    <target name="runscript">

        <!-- Run script from a file -->
        <script language="beanshell" src="myscript.bsh"/>

        <!-- Run script in-line -->
        <script language="beanshell">&lt;![CDATA[
            for(int i=0; i<10; i++ )
                print( "i="+i );
        ]]&gt;</script>

    </target>
</project>
```

Learning More

BeanShell is a simple tool but one with rapidly evolving capabilities. To learn more about BeanShell you are highly encouraged to download the source and build it (using the Ant build file). Even if you don't consider yourself a developer, you can learn a lot from the source distribution by looking at the implementation of the standard BeanShell commands and the test suite.

Almost all of the built-in BeanShell commands are simply scripts stored in the BeanShell JAR file under the path "bsh/commands". A good way to familiarize yourself with more of BeanShell is to take a look at those commands. Simply unpack bsh/commands/*.bsh from the JAR file. The BeanShell test suite consists of many BeanShell scripts that exercise all parts of the language.

In addition to the mailing list and mailing list archives, an important source of information is the "recent changes" file supplied with the source distribution and online at: http://www.beanshell.org/Changes.html. This file is one of the few documents that is always up to date with the latest release (smile).

Helping With the Project

BeanShell is an open source project which relies on people like you to get things done. If you are excited about BeanShell there is undoubtedly some way for you to help. If you are a developer, there is always work (sometimes boring, sometimes not) to be done. If you are not a developer you may still be able to help by writing new tests for the test suite, or working on the documentation, web site, tutorials or examples.

Here are some things that we can always use help with:

- **Tests for the test suite** - We need more tests! BeanShell relies heavily on its test suite to guarantee that changes don't break subtle aspects of the language. Often tests are added for specific bug cases (Developers: please add a test for any bug you fix!). But it would be best if tests were generalized to cover all of the "corner cases" too.
- **Bug fixes** - Check the bugs list at the sourceforge site and dig into the code. Some bug fixes are easy, some are deep. Feel free to contact me (pat@pat.net) directly if you want help getting started on an issue.
- **Docs** - We can always use articles, documentation and examples.
- **Integration and third party tools** - Have you integrated BeanShell into another tool or environment? Let us know and we'll link to your site.
- **Feedback from the World** - Despite BeanShell's relative popularity you would be amazed at how little information we have about who is using the tool and how. If you are using it or you know people using it please let us know!

Credit and Acknowledgments

Many people have contributed substantially to BeanShell over the years. I will attempt to start crediting those individual here. Please do not be offended if your name is missing. This list will grow as I have time to work backwards through my email and recover names.

- Thanks to Daniel Leuck for his long time support and many contributions to the project.

Me

Finally, I will put in a plug for myself: Pat Niemeyer (pat@pat.net)



If you like BeanShell check out my book: Learning Java, O'Reilly & Associates, 2nd edition.

Winner of the Best Java Introductory Book - JavaOne 2001. Learning Java (previously titled Exploring Java) is available in nine languages world-wide. It is a comprehensive overview of the Java language and APIs including a brief introduction to BeanShell as well!

License and Terms of Use

You may freely use and reproduce this document in its entirety as long as you preserve this license information and a pointer to the original web site: http://www.beanshell.org. You may integrate parts of this document into your own documentation as long as you provide this same information at an appropriate place in your document.

This document is copyright Pat Niemeyer, 2002. Sections contributed by other authors are copyrighted to those individuals and subject to the terms of use described above.

BeanShell Commands Documentation

The following documentation was generated automatically by 'BshDoc' from Javadoc style comments in the BeanShell command script files. See "BshDoc" for more information.

addClassPath	void addClassPath(string | URL)
bg	Thread bg(String filename)
bind	bind (bsh .This ths , bsh .NameSpace namespace)
browseClass	void browseClass(String | Object | Class)
cat	cat (String filename) cat (URL url) cat (InputStream ins) cat (Reader reader)
cd	void cd (String pathname)
classBrowser	classBrowser ( )
clear	clear ( )
cp	cp (String fromFile , String toFile)
debug	debug ( )
desktop	void desktop()
dirname	String dirname (String pathname)
editor	editor ( )
error	void error (item)
eval	Object eval (String expression)
exec	exec (String arg)
exit	exit ( )
extend	This extend(This object)
frame	Frame | JFrame | JInternalFrame frame(Component component)
getBshPrompt	String getBshPrompt ( )
getClass	Class getClass (String name)
getClassPath	URL [ ] getClassPath ( )
getResource	URL getResource (String path)

getSourceFileInfo	getSourceFileInfo ( )
importCommands	void importCommands(resource path | package name)
importObject	void importObject(Object object)
javap	void javap(String | Object | Class | ClassIdentifier)
load	Object load (String filename)
makeWorkspace	makeWorkspace (String name)
mv	mv (String fromFile , String toFile)
object	This object()
pathToFile	File pathToFile (String filename)
print	void print (arg)
printBanner	printBanner ( )
pwd	pwd ( )
reloadClasses	void reloadClasses([package name])
rm	boolean rm (String pathname)
run	run (String filename , Object runArgument) run (String filename)
save	void save (Object obj , String filename)
server	void server (int port)
setAccessibility	setAccessibility (boolean b)
setClassPath	void setClassPath(URL [])
setFont	Font setFont (Component comp , int ptsize)
setNameCompletion	void setNameCompletion (boolean bool)
setNameSpace	setNameSpace (ns)
setStrictJava	void setStrictJava (boolean val)
show	show ( )
source	Object source (String filename) Object source (URL url)
sourceRelative	sourceRelative (String file)
super	This super(String scopename)
unset	void unset (String name)
which	which(classIdentifier | string | class)
workspaceEditor	workspaceEditor(bsh.Interpreter parent, String name)

addClassPath
void addClassPath(string | URL)

Add the specified directory or JAR file to the class path. e.g.

```
addClassPath( "/home/pat/java/classes" );
addClassPath( "/home/pat/java/mystuff.jar" );
addClassPath( new URL("http://myserver/~pat/somebeans.jar") );
```

See Class Path Management

bg
Thread bg(String filename)

Source a command in its own thread in the caller's namespace

This is like run() except that it runs the command in its own thread. Returns the Thread object control.

bind
bind (bsh .This ths , bsh .NameSpace namespace)

Bind a bsh object into a particular namespace and interpreter

browseClass
void browseClass(String | Object | Class)

Open the class browser to view the specified class. If the argument is a string it is considered to be a class name. If the argument is an object, the class of the object is used. If the arg is a class, the class is used.

Note: To browse the String class you can't supply a String. You'd have to do: browseClass(String.class);

cat
cat (String filename)
cat (URL url)
cat (InputStream ins)
cat (Reader reader)

Print the contents of filename, url, or stream (like Unix cat)

cd
void cd (String pathname)

Change working directory for dir(), etc. commands (like Unix cd)

classBrowser
classBrowser ( )

Open the class browser.

clear
clear ( )

Clear all variables, methods, and imports from this namespace. If this namespace is the root, it will be reset to the default imports. See NameSpace.clear();

cp
cp (String fromFile , String toFile)

Copy a file (like Unix cp).

debug
debug ( )

Toggle on and off debug mode. Debug output is verbose and generally useful only for developers.

desktop
void desktop()

Start the BeanShell GUI desktop.

dirname
String dirname (String pathname)

Return directory portion of path based on the system default file separator. Note: you should probably use pathToFile() to localize the path relative to BeanShell's working directory and then file.getAbsolutePath() to get back to a system localized string.

Example: to change to the directory that contains the script we're currently executing:

```
// Change to the directory containing this script
path=pathToFile( getSourceFileInfo() ).getAbsolutePath();
cd( dirname( path ) );
```

editor
editor ( )

Open a GUI editor from the command line or in the GUI desktop mode. When run from the command line the editor is a simple standalone frame. When run inside the GUI desktop it is a workspace editor. See workspaceEditor()

error
void error (item)

Print the item as an error. In the GUI console the text will show up in (something like) red, else it will be printed to standard error.

eval
Object eval (String expression)

Evaluate the string in the current interpreter (see source()). Returns the result of the evaluation or null.

Evaluate a string as if it were written directly in the current scope, with side effects in the current scope.

e.g.

```
a=5;
eval("b=a*2");
print(b); // 10
```

eval() acts just like invoked text except that any exceptions generated by the code are captured in a bsh.EvalError. This includes ParseException for syntactic errors and TargetError for exceptions thrown by the evaluated code.

e.g.

```
try {
    eval("foo>>><>M>JK$LJLK$");
} catch ( EvalError e ) {
    // ParseException caught here
}

try {
    eval("(Integer)true");  // illegal cast
} catch ( EvalError e ) {
    // TargetException caught here
    print( e.getTarget() )  // prints ClassCastException
}
```

If you want eval() to throw target exceptions directly, without wrapping them, you can simply redefine own eval like so:

```
myEval( String expression ) {
    try {
        return eval( expression );
    } catch ( TargetError e ) {
        throw e.getTarget();
    }
}
```

Here is a cute example of how to use eval to implement a dynamic cast. i.e. to cast a script to an arbitrary type by name at run-time where the type is not known when you are writing the script. In this case the type is in the variable interfaceType.

```
reference = eval( "("+interfaceType+")this" );
```

Returns the value of the expression.

Throws bsh.EvalError on error

exec
exec (String arg)

Start an external application using the Java Runtime exec() method. Display any output to the standard BeanShell output using print().

exit
exit ( )

Conditionally exit the virtual machine. Call System.exit(0) unless bsh.system.shutdownOnExit == false.

extend
This extend(This object)

Return a new object that is a child of the specified object. **Note: this command will likely change along with a better inheritance mechanism for bsh in a future release.**

extend() is like the object() command, which creates a new bsh scripted object, except that the namespace of the new object is a child of the parent object.

For example:

```
foo=object();
bar=extend(foo);

is equivalent to:

foo() {
    bar() {
        return this;
    }
}

foo=foo();
bar=foo.bar();

and also:

oo=object();
ar=object();
ar.namespace.bind( foo.namespace );
```

The last example above is exactly what the extend() command does. In each case the bar object inherits variables from foo in the usual way.

frame
Frame | JFrame | JInternalFrame frame(Component component)

Show component in a frame, centered and packed, handling disposal with the close button.

Display the component, centered and packed, in a Frame, JFrame, or JInternalFrame. Returns the frame. If the GUI desktop is running then a JInternaFrame will be used and automatically added to the desktop. Otherwise if Swing is available a top level JFrame will be created. Otherwise a plain AWT Frame will be created.

getBshPrompt
String getBshPrompt ( )

Get the value to display for the bsh interactive prompt. This command checks for the variable bsh.prompt and uses it if set. else returns "bsh % "

Remember that you can override bsh commands simply by defining the method in your namespace. e.g. the following method displays the current working directory in your prompt:

```
String getBshPrompt() {
        return bsh.cwd + " % ";
}
```

getClass
Class getClass (String name)

Get a class through the current namespace utilizing the current imports, extended classloader, etc.

This is equivalent to the standard Class.forName() method for class loading, however it takes advantage of the BeanShell class manager so that added classpath will be taken into account. You can also use Class.forName(), however if you have modified the classpath or reloaded classes from within your script the modifications will only appear if you use the getClass() command.

getClassPath
URL [ ] getClassPath ( )

Get the current classpath including all user path, extended path, and the bootstrap JAR file if possible.

getResource
URL getResource (String path)

Get a resource from the BeanShell classpath. This method takes into account modification to the BeanShell class path via addClassPath() and setClassPath();

getSourceFileInfo
getSourceFileInfo ( )

Return the name of the file or source from which the current interpreter is reading. Note that if you use this within a method, the result will not be the file from which the method was sourced, but will be the file that the caller of the method is reading. Methods are sourced once but can be called many times... Each time the interpreter may be associated with a different file and it is that calling interpreter that you are asking for information.

Note: although it may seems like this command would always return the getSourceFileInfo.bsh file, it does not since it is being executed after sourcing by the caller's interpreter. If one wanted to know the file from which a bsh method was sourced one would have to either capture that info when the file was sourced (by saving the state of the getSourceFileInfo() in a variable outside of the method or more generally we could add the info to the BshMethod class so that bsh methods remember from what source they were created...

importCommands
void importCommands(resource path | package name)

Import scripted or compiled BeanShell commands in the following package in the classpath. You may use either "/" path or "." package notation. e.g.

```
// equivalent
importCommands("/bsh/commands")
importCommands("bsh.commands")
```

```
When searching for a command each path will be checked for first, a file
named 'command'.bsh and second a class file named 'command'.class.
```

```
You may add to the BeanShell classpath using the addClassPath() or
setClassPath() commands and then import them as usual.
```

```
addClassPath("mycommands.jar");
importCommands("/mypackage/commands");
```

```
If a relative path style specifier is used then it is made into an absolute
path by prepending "/". Later imports take precedence over earlier ones.
```

```
Imported commands are scoped just like imported clases.
```

importObject
void importObject(Object object)

Import an instance object into this namespace (analogous to static class imports). You can import the methods and fields of a Java object instance into a BeanShell namespace. e.g.

```
Map map = new HashMap();
importObject( map );
put("foo", "bar");
print( get("foo") ); // "bar"
```

javap
void javap(String | Object | Class | ClassIdentifier)

Print the public fields and methods of the specified class (output similar to the JDK javap command).

If the argument is a string it is considered to be a class name. If the argument is an object, the class of the object is used. If the arg is a class, the class is used. If the argument is a class identifier, the class identified by the class identifier will be used. e.g. If the argument is the empty string an error will be printed.

```
// equivalent
javap( java.util.Date ); // class identifier
javap( java.util.Date.class ); // class
javap( "java.util.Date" ); // String name of class
javap( new java.util.Date() ); // instance of class
```

load
Object load (String filename)

Load a serialized Java object from filename. Returns the object.

makeWorkspace
makeWorkspace (String name)

Open a new workspace (JConsole) in the GUI desktop.

mv
mv (String fromFile , String toFile)

Rename a file (like Unix mv).

object
This object()

Return an "empty" BeanShell object context which can be used to hold data items. e.g.

```
myStuff = object();
myStuff.foo = 42;
myStuff.bar = "blah";
```

pathToFile
File pathToFile (String filename)

Create a File object corresponding to the specified file path name, taking into account the bsh current working directory (bsh.cwd)

print
void print (arg)

Print the string value of the argument, which may be of any type. If beanshell is running interactively, the output will always go to the command line, otherwise it will go to System.out.

Most often the printed value of an object will simply be the Java toString() of the object. However if the argument is an array the contents of the array will be (recursively) listed in a verbose way.

Note that you are always free to use System.out.println() instead of print().

printBanner
printBanner ( )

Print the BeanShell banner (version and author line) - GUI or non GUI.

pwd
pwd ( )

Print the BeanShell working directory. This is the cwd obeyed by all the unix-like bsh commands.

reloadClasses
void reloadClasses([package name])

Reload the specified class, package name, or all classes if no name is given. e.g.

```
reloadClasses();
reloadClasses("mypackage.*");
reloadClasses(".*")  // reload unpackaged classes
reloadClasses("mypackage.MyClass")
```

See "Class Path Management"

rm
boolean rm (String pathname)

Remove a file (like Unix rm).

run
run (String filename , Object runArgument)
run (String filename)

Run a command in its own in its own private global namespace, with its own class manager and interpeter context. (kind of like unix "chroot" for a namespace). The root bsh system object is extended (with the extend() command) and made visible here, so that general system info (e.g. current working directory) is effectively inherited. Because the root bsh object is extended it is effectively read only / copy on write... e.g. you can change directories in the child context, do imports, change the classpath, etc. and it will not affect the calling context.

run() is like source() except that it runs the command in a new, subordinate and prune()'d namespace. So it's like "running" a command instead of "sourcing" it. run() teturns the object context in which the command was run.

Returns the object context so that you can gather results.

Parameter runArgument an argument passed to the child context under the name runArgument. e.g. you might pass in the calling This context from which to draw variables, etc.

save
void save (Object obj , String filename)

Save a serializable Java object to filename.

server
void server (int port)

Create a remote BeanShell listener service attached to the current interpreter, listening on the specified port.

setAccessibility
setAccessibility (boolean b)

Setting accessibility on enables to private and other non-public fields and method.

setClassPath
void setClassPath(URL [])

Change the classpath to the specified array of directories and/or archives.

See "Class Path Management" for details.

setFont
Font setFont (Component comp , int ptsize)

Change the point size of the font on the specified component, to ptsize. This is just a convenience for playing with GUI components.

setNameCompletion
void setNameCompletion (boolean bool)

Allow users to turn off name completion.

Turn name completion in the GUI console on or off. Name competion is on by default. Explicitly setting it to true however can be used to prompt bsh to read the classpath and provide immediate feedback. (Otherwise this may happen behind the scenes the first time name completion is attempted). Setting it to false will disable name completion.

setNameSpace
setNameSpace (ns)

Set the namespace (context) of the current scope.

The following example illustrates swapping the current namespace.

```
fooState = object();
barState = object();

print(this.namespace);
setNameSpace(fooState.namespace);
print(this.namespace);
a=5;
setNameSpace(barState.namespace);
print(this.namespace);
a=6;

setNameSpace(fooState.namespace);
print(this.namespace);
print(a);  // 5

setNameSpace(barState.namespace);
print(this.namespace);
print(a); // 6
```

You could use this to creates the effect of a static namespace for a method by explicitly setting the namespace upon entry.

setStrictJava
void setStrictJava (boolean val)

Enable or disable "Strict Java Mode". When strict Java mode is enabled BeanShell will:

1. Require typed variable declarations, method arguments and return types.
2. Modify the scoping of variables to look for the variable declaration first in the parent namespace, as in a java method inside a java class. e.g. if you can write a method called incrementFoo() that will do the expected thing without referring to "super.foo".

   See "Strict Java Mode" for more details.

   Note: Currently most standard BeanShell commands will not work in Strict Java mode simply because they have not been written with full types, etc.

show
show ( )

Toggle on or off displaying the results of expressions (off by default). When show mode is on bsh will print() the value returned by each expression you type on the command line.

source
Object source (String filename)
Object source (URL url)

Read filename into the interpreter and evaluate it in the current namespace. Like the Bourne Shell "." command. This command acts exactly like the eval() command but reads from a file or URL source.

sourceRelative
sourceRelative (String file)

Source a file relative to the callering script's directory.

e.g. scripts A running in dir A sources script B in dir B. Script B can use this command to load additional scripts (data, etc.) relative to its own location (dir B) without having to explicitly know its "home" directory (B).

Note: this only works for files currently.

super
This super(String scopename)

Return a BeanShell 'this' reference to the enclosing scope (method scope) of the specified name. e.g.

```
foo() {
    x=1;
    bar() {
        x=2;
        gee() {
            x=3;
            print( x ); // 3
            print( super.x ); // 2
            print( super("foo").x ); // 1
        }
    }
}
```

This is an experimental command that is not intended to be of general use.

unset
void unset (String name)

"Undefine" the variable specifed by 'name' (So that it tests == void).

Note: there will be a better way to do this in the future. This is currently equivalent to doing namespace.setVariable(name, null);

which
which(classIdentifier | string | class)

Use classpath mapping to determine the source of the specified class file. (Like the Unix which command for executables).

This command maps the entire classpath and prints all of the occurrences of the class. If you just want to find the first occurrence in the classpath (the one that will be used by Java) you can also get it by printing the URL of the resource. e.g.:

```
print( getResource("/com/foo/MyClass.class") );
        // Same as...
// System.out.println(
//    getClass().getResourceAsStream("/com/foo/MyClass.class" ) );
```

Note: This is all a lie! This command is broken and only reports the currently first occurence! To be fixed!

workspaceEditor
workspaceEditor(bsh.Interpreter parent, String name)

Make a new workspaceEditor in the GUI.
