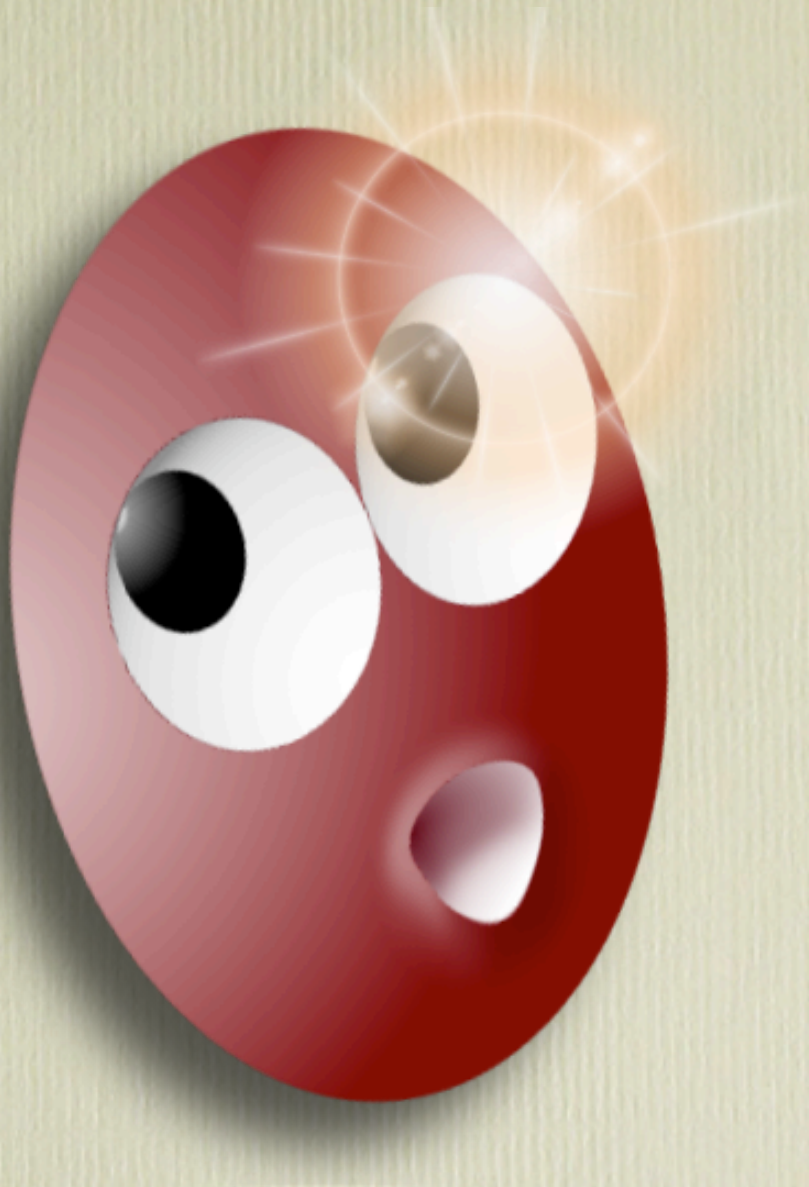
BeanShell BOF



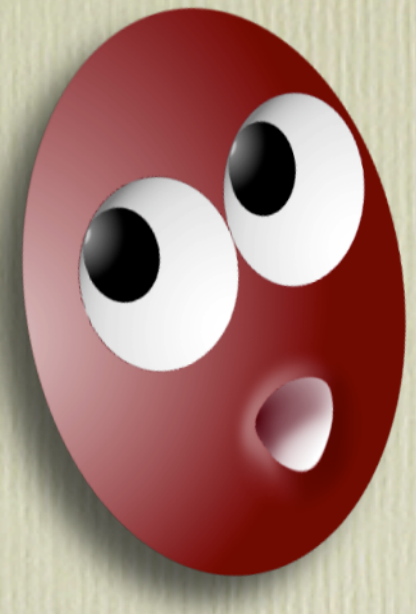
Topics

- JSR-274 Update.
- What's New in BeanShell 2.0. (A Quick Recap)
- New APIs for Scripting Java
- Where Are We Going... ?
- Developer Resources and our New Home (Dan).
- Q&A

Why go the JSR route?

- What does BeanShell have to gain?
- What does Java have to gain?

What's New in 2.0?



- Performance
- Error Reporting
- New Language Features
- Java 1.4 Compatibility
- Full Scripted Classes

Performance Improvements

- JavaCC 3.0 based Parser faster and 30% smaller. Many grammar optimizations.
- Caching of method resolution for performance. (50% speed improvement in some cases).

Better Error Reporting

- Expected niceties such as line numbers and invocation text on error messages.
- Script stack traces
  (e.g. method a() called method b(), etc.)

Applet Friendly (Again)

- BeanShell core features do not trip Applet security.
- Advantage of the existing reflection based implementation.

Misc. New features...

- Mix-ins
- Properties style auto-allocation of variables

Mix-ins

- Instance Object imports (Mix-ins) with **`importObject()`**

```
Map map = new HashMap();
importObject( map );

put("foo", "bar");
print( get("foo") ); // "bar"
```

Properties Style Auto-Allocation of Variables

```
// foo is initially undefined
foo.bar.gee = 42;

print( foo.bar.gee ); // 42

print( foo.bar ); //'this' reference (XThis)
to Bsh object: auto: bar

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

Props Example 1

```
// Home directory
myApp.homeDir="/pkg/myApp";

// User Info
myApp.user.defaultUser="Bob";
myApp.color.fgcolor = "Aqua";

// Complex properties
myApp.color.bgcolor = Color.BLUE;  // Real enumerations for free!
myApp.color.colorset =
        new Color [] { Color.RED, Color.GREEN, Color.BLUE };

// Script application behavior
onStartup() {
    print( "Hello User: " + USERNAME );
}

// Include more config and scripts via source(), eval(), etc.
source( System.getProperty("user.home") + "/" + ".myAppConfig" );

// Configure with real objects, with real arguments
SocketFactory.setDefaultSocketFactory( new MySocketFactory(42) );
```

```
// Create interpreter and read myprops.bsh
Interpreter config = new Interpreter();
config.source("myprops1.bsh");

// Read simple string properties
String homeDir = (String)config.get("myApp.homeDir");
String defaultUser = (String)config.get("myApp.user.defaultUser");

// Read true object properties
Color fgcolor = (Color)config.get("myApp.color.bgcolor");

// True nested properties (not yet as pretty as it could be)
NameSpace myAppColor = ((This)config.get("myApp.color")).getNameSpace();
// Iterate over myApp.color nested properties
String [] varNames = myAppColor.getVariableNames();
//for( String name : varNames ) { }

// Set the USERNAME variable for the script's use
config.set("USERNAME", "Pat");

// Execute user's scripted onStartup() method, if it exists
config.eval("onStartup()");
```

Properties Example 2

```
public class MyApp2
{
    // JavaBean accessor methods
    public void setHomeDir( String homeDir ) { ... }
    public User getUser() { return new User(); }

    void readProps() throws IOException, EvalError
    {
        // Create interpreter and read myprops.bsh
        Interpreter config = new Interpreter();
        // Set this object as "myApp"
        config.set("myApp", this);
        config.source("myprops2.bsh");

        // No property fetching code necessary!
        // Execute behavior...
    }
```

Java Syntax Compatibility

- Full Java 1.4 syntax support (on all VMs)
- Some Java 5 Features (on all VMs)
  - Boxing, enhanced for-loop, static imports

True Scripted Classes

- Generated classes with real Java types, backed by the interpreter.
- Scripts can now go anywhere Java goes: Extend / Implement arbitrary Java classes
- Load classes from .java source files

Scripted Classes
(How do they fit in?)

- Expose all methods and typed variables of the class.
- Bound in the namespace in which they are declared.
- May freely mix loose / script syntax with full Java class syntax.

Scripted Class Example

```
// HelloWorld.bsh

showMessage() { print(“Hello!”); }
count = 5;

class HelloWorld extends Thread {
   public void run() {
      for(i=0; i<count; i++)
         showMessage();
   }
}

new HelloWorld().start();
```

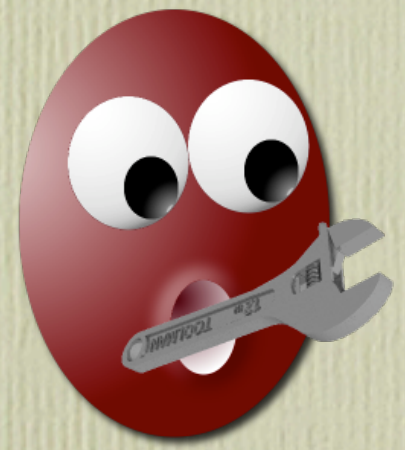
Full Java Semantics

- this, super, static and instance variables and blocks.
- Full superclass visibility
- Full constructors functionality this(), super(), controlled superclass construction

Scripted Class Implementation

- Light weight bytecode generator (17K subset of ASM)
- Stub classes delegate all method and constructor calls to interpreter. All fields generated to class.
- Generated accessor methods for superclass visibility

Current Limitations



- Reflective Access Permissions
- Bugs to work out...

New APIs

- javax.script (JSR-223) - Powerful, Pluggable Scripting languages for Java.
- The BeanShell API Compiler (New!) - True persistent classes backed by scripts.

Javax.Script JSR-223

- Standardized packaging, discovery, metadata
- Support for OO Scripting Languages
- Pluggable Namespaces and Script Contexts
- Minimalist language neutral script generation capabilities.
- First Class Support by BeanShell

The BeanShell API Compiler

(Who needs a Scripting API?)

- BeanShell generated stub classes can now be saved to ordinary dot class files.
- Class Initialization code launches an interpreter for the associated script file.
- Indistinguishable from ordinary classes from the outside. Chewy on the inside.

Compiling an API Class

```
// HelloWorld.bsh
showMessage() { print("Hello!"); }
count = 5;

class HelloWorld extends Thread {
   public void run() {
      for(i=0; i<count; i++)
         showMessage();
   }
}
```

```
% java bsh.Build HelloWorld.bsh
# Produces HelloWorld.class
```

Compiled API Classes

- Find associated script file by name

```
% HelloWorld.bsh
% HelloWorld.class
```

- Script files can evolve after the class is generated as long as they fulfill the basic contract.
- Can contain any amount of additional “loose” Java code in or outside the class body.
- Launch an interpreter on class static initialization

Another Example

```
// Make Foo.bsh launchable via main() method

class Foo {
  public static void main( String [] args ) { } };

foo = “I’m alive!”;
print(foo);
```

```
% java bsh.Build Foo.bsh
% java Foo  // I’m alive!
```

Many Possibilities...

- Real classes as properties files...
- Servlets and web apps with runtime modifiable classes.

Where Are We Going...?

- Java 5 Syntax Compatibility
- Performance, Control, Security
- Pluggable / Extensible Syntax...?

Java 5 Syntax Compatibility

- Targeted for 3.0
- Beyond Boxing, for-loops, and static import...
- Generics, Annotations, Varargs...

Performance

- More Bytecode where allowed?
- Tuning... We have not yet begun to tune.

More Control

- Basic “Job Control”
- Security...

Pluggable Syntax?

- Always a strong desire to add more and more specialized syntax to any language, especially a scripting language.
- Regular Expressions, XML, SQL, etc... (We all want this stuff.)
- How do we accommodate new, exotic grammar and stay Java Compatible?

What are the criterion for an extension mechanism?

- Aesthetically appealing, Simple, and **Java-Like**
- Unambiguous with respect to Java (now and as Java evolves)
- Pluggable, easy to write new grammars, encourage creativity and experimentation.
- Tight enough integration to be useful.
- Ability to “make it go away” completely if you hate it (very low footprint in the core).

Options...

- Re-implement BeanShell in a meta-language.
- Provide a language “escape hatch”.
- Pro-s and Con-s to each...

Let's Look at eval()...

- Executes a String as code
- Produces a traditional return value
- Produces side effects in the **caller's scope**

```
value =
   eval( "foo=2; bar=3; someMethod()" );

print( foo ); // 2
```

A Possibility

- Extend the concept of the eval() method to encompass free-form syntax within the traditional formal method arguments space.

```
value =
    eval( foo=2; bar=3; someMethod(); );

// or, nicely formatted...

eval(
    foo=2;
    bar=3;
    someMethod();
);
```

Extending to new Syntax:

```
import bsh.regex;

String myString = "My name is Pat Niemeyer";

regex(
    myString/s/Pat/Patrick/g
    myString/My name is (\w+) (\w+)/
    firstName = $1
    lastName = $2
);

print( firstName ); // Patrick
```

Things to note...

- The example regex() dialect method is unambiguously identified by an import.
- Syntax within the bounds of the method invocation is completely domain specific (free-form) and may even by line-oriented, as shown.
- Dialect has full access to read values from the caller's environment and produces eval-style side effects (variable assignments and method calls) in the caller's environment.

More (Super Speculative) Examples:

```
// Implement some Bourne Shell functionality

String myString = "Pat Niemeyer";
InputStream myStream = url.openStream();
File myFile = ...;

sh(
    cat stream > foo.txt
    lines=`grep "foo" myFile`
    echo $myString | someApplication
);

print(lines); // foo this foo that...
```

More (Super Speculative) Examples:

```
// SQL - Implement specialized SQL syntax

rowset = sql(
   open db://somedatabase
   select * from Foo where Bar
);

for ( row : rowset )
     print( row );
```

More (Super Speculative) Examples:

```
// Simple multi-line "here" document

String myString = doc(
    This is multi-line text
        with a platform specific
            line ending...
    Foo!
);
```

More (Super Speculative) Examples:

```
// XML and XPath
Document xmlDoc = xml(
   <Library>
     <Book name="Learning Java" category=>
        <stuff/>
     </Book>
   </Library>
);

String myText = ...;
category = xpath(
  myText/Library/Book[name="Learning Java"]/
@category );
```

More (Super Speculative) Examples:

```
// Lists, Maps, and Closures?

myList = list( 1, 2, ( "foo", "bar" ) );

myMap = map( foo=”foo” bar=someObject );

myCode = code( ... ); // ?
```

More (Super Speculative) Examples:

```
// Nesting should work...
list=list();

sh(
  cd /files
  foreach f in *.txt
  do
     eval(
       String path = f.getCanonicalPath();
       list.add( new URL(path) );
     )
  done;
);
```

More (Super Speculative) Examples:

```
// Completely crazy things...
// Implement a subset of awk for BeanShell
// which can invoke BeanShell methods
someMethod( arg ) { ... }

result = bawk(
    /Name/ {
        names++;
        someMethod( $1 );
    }

    END { print "the end" }
);
```

Dialect Implementation Issues

- Preprocessing good and bad...
- API for Dialect “plugins” is essentially just the current BeanShell NameSpace API
- Dialects could be compiled or scripted right in the app
- What tools can we offer to Dialect writers to make this easy? A Lexer / Yaccer? Template languages?

A Complementary Extension: "Smart" Dynamic Types

- What if we provide an interface that allows a scripted object to completely control its expression and polymorphism within the interpreter?

```
interface DynamicType {
    getMethods();
    getVariables();

    isAssignableTo();
    assignTo();
    ...
}
```

What could we do with that?

- Methods that are sensitive to assignment context...

```
cat(file); // Output to stdout

String txt = cat(file); // Capture to String
System.out.println( cat(file) ); // String

InputStream in = cat(file); // Stream it
```

Wrapping Up...

Favorite Pair of Quotes...

"... it's been a long time since I've seen a non-commercial project that is so well-documented"

-- Robert F Schmitt.

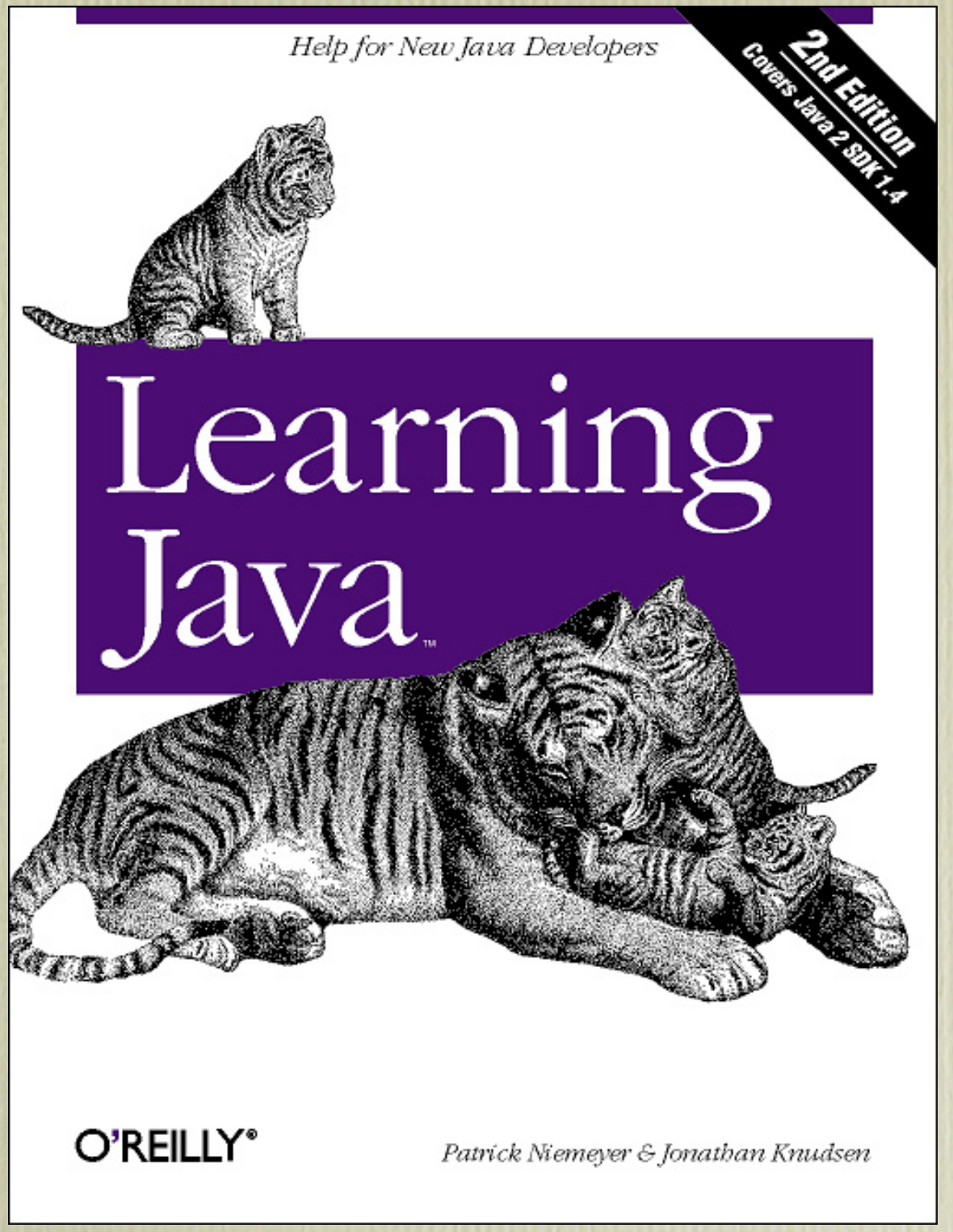
Favorite Pair of Quotes...

"... it's been a long time since I've seen a non-commercial project that is so well-documented"

-- Robert F Schmitt.

"In going through your tutorial I have found a few spelling errors and poorly constructed sentences. I am assuming English is not a first language?"

-- Bob Linden


Help for New Java Developers
2nd Edition
Covers Java 2 SDK 1.4
Learning
Java™
O'REILLY®
Patrick Niemeyer & Jonathan Knudsen