Writing Your Own JSR-Compliant, Domain-Specific Scripting Language

David Smith, Software Engineer
John Colosi, Software Engineer

TS-5693
Prove the power of Domain Specific Languages to simplify work, communicate ideas, and manipulate space-time
Demonstrate by example some techniques for using Java™ Specification Request (JSR)-223 to build your own simple, portable Domain Specific Language
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
What is JSR-223?

JSR-223 defines certain contracts between the Java Virtual Machine (JVM™) and a scripting language

- It covers:
  - Motivations behind its creation
  - How the JVM tool will discover available scripting languages at runtime
  - The classes and interfaces that define relationships between a script engine and the JVM tool

- Scripting support was introduced with Java 6 platform

- At scripting.dev.java.net you can find a list of JSR-conforming scripting-language implementations
What is a scripting language?

- A language whose programs control applications rather than exist as applications on their own
- Scripts ordinarily contain sequences of commands intended for the target application
- Usually interpreted
- Definition has become muddled
What is a DSL?

- Full of terms and jargon particular to the domain
- Not necessarily a computer language, even though that’s how engineers might think about them
- Often impenetrable to outsiders
- The same way design patterns can be referred to by name and connote lots of information to those knowledgeable
Shared context

When there are things we all understand, we can factor them out of the communication

DSLs assume a shared context between language user and language interpreter

The “domain” in DSL signifies that context
What is Domain Specific Language?

> Spoken languages are imperfect
  • Rich
  • Not concise
What is Domain Specific Language?

> “The way an object in front of a light source appears darker than it really is”
  • Silhouette
What is Domain Specific Language?

> “The strange effect of perspective when a large viewing angle is forced into a smaller viewing angle”
  • Fisheye
Diamonds in the Rough

When my wife says:

- “What happened to the last batter?”
Diamonds in the Rough

There were runners at First...
Diamonds in the Rough

… and Second base…
Diamonds in the Rough

… with 1 out…
Diamonds in the Rough

> ... when the batter...
Diamonds in the Rough

> ... hit a pop fly in the infield...
Diamonds in the Rough

… The defense could just drop the ball…
Diamonds in the Rough

> ... and turn a double play...
Diamonds in the Rough

… to avoid this, the batter was called out…
Diamonds in the Rough

> … and the runners…
Diamonds in the Rough

... hold their positions
Diamonds in the Rough

That’s a mouthful
Diamonds in the Rough

If David asked the same question…
Diamonds in the Rough

Infield Fly Rule
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
Relationship of computer languages

Rich languages (e.g., Java)

Scripting languages (e.g., Groovy)

Domain-specific languages
DSLs vs. rich computer languages

> Rich languages start by not knowing what you’re talking about
  • The Java platform doesn’t know what a bank account is

> Rich languages must allow for all possibilities of application
  • You can use C to navigate a space shuttle, C++ to run a backend financial system, and a Java platform to make Duke show up in an applet
Duke
Why use a DSL?

- Deep Blue is a chess playing computer
Why use a DSL?

- In 1997, it won a match…
- …against Gary Kasparov, reigning world champion
Why use a DSL?

- But Deep Blue doesn’t know the rules of chess...
- It just has a good memory
Why use a DSL?

- Deep Blue needs something better than the SQL, the “Standard” Query Language
- CQL is the “Chess” Query Language
What could this mean?

:relation (:missingpiececount A 1)
CQL

> Syntax
  - K white king
  - N white knight
  - B white bishop
  - P white pawn
  - R white rook
  - Q white queen
  - A any white piece
  - . empty square

> Keywords
  - :mate
  - :check
  - :wtm (white to move)
  - :terminal (final position)

Syntax
  - k black king
  - n black knight
  - b black bishop
  - p black pawn
  - r black rook
  - q black queen
  - a any black piece
  - U any piece
:relation (:missingpiececount A 1)

Find games where White sacrificed a piece to improve his position
CQL

- CQL was developed by Gady Costeff and Lewis Stiller
- It is Copyright (c) 2003-2004 and is free
Other DSL examples

- Wiki markup language
- SQL
  - Imagine all the complexity behind this simple command
    - SELECT /*+ PARALLEL(emp,4) */ COUNT(*) FROM emp ;
- SPSS – stat *example*
- Solaris™ Flash technology actionscript – this is pretty rich though
- AI scripts
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
Beginning the implementations

- JSR-223 defines six classes and six interfaces
- All can be found in the javax.script package
- These fall into the categories of:
  - the scripting engine for a language
  - the context in which a script is running
JSR-223 API: relations between manager and factories

ScriptEngine Manager → Service locator

- ScriptEngine Factory (language x)
- ScriptEngine Factory (language y)
- ScriptEngine Factory (language z)

language x
language y
language z
JSR-223 API: the ScriptEngineManager

➢ The ScriptEngineManager
  • Auto-discovers scripting languages
  • Determines which engine factory to give you based on language name, MIME-type, or extension that you supply

➢ It’s a factory of factories
JSR-223 API: relations between factories and engines
JSR-223 API: the script engine and its factory

➢ ScriptEngineFactory
  • Produces instances of your engine
  • Maps MIMEs, extensions, and language names to itself
  • Provides syntax for method calls and output statements
  • Provides mechanism for building a script from statements

➢ ScriptEngine
  • Interprets and runs (“evaluates”) your scripts
  • Requires you to implement overloaded eval() method for script interpretation
Code example: hello world

// get Groovy engine from the SEM
ScriptEngineManager sem = new
    ScriptEngineManager();
ScriptEngine groovyEng =
    sem.getEngineByName( "groovy" );
// run a script
groovyEng.eval("println 'hello world' ");
JSR-223 API: closer look at ScriptEngine class

public Object eval ( ... ) throws ScriptException

> The ( ... ) can be

• String
• Reader
• String, ScriptContext
• Reader, ScriptContext
• String, Bindings
• Reader, Bindings
JSR-223 API: Interactions between a Java platform application and an engine

Java application

```java
{  
  ... Java code  
  ...  
}
```

Diagram:
- **ScriptContext**
  - **Bindings**
  - **ScriptEngine**
    - **ScriptEngine Manager**
    - **ScriptEngine Factory**
JSR-223 API: the script’s runtime context

- **Bindings**
  - Holds name-value pairs
  - Objects can be bound into, retrieved from, and removed from the Bindings by both the script and the Java platform caller to the script

- **ScriptContext**
  - Context in which scripts run
  - Provides access to scoped Bindings
  - Provides reader and two writers (standard, error) for script input and output
JSR-223 API: Bindings hierarchy

- Bindings have scopes
  - Global scope – can be set in the ScriptEngineManager or the ScriptEngine before script evaluation
  - Engine scope – can be set in the ScriptEngine before script evaluation
  - “Local” scope – can be passed in at script-evaluation time and used for that evaluation only

- The ScriptEngine will look for a specific binding (property in a Bindings) by going from local to engine to global scope
Code example: round trip with “local” bindings

```java
// get the Groovy engine from the SEM
ScriptEngineManager sem = new ScriptEngineManager();
ScriptEngine groovyEng = sem.getEngineByName("groovy");

// create some bindings for our script run
Bindings runBindings = groovyEng.createBindings();
runBindings.put("greeting", "hello from script");

// run the script and supply those bindings
groovyEng.eval("println greeting", runBindings);
```
JSR-223 API: helper classes

- SimpleBindings
- SimpleScriptContext
- AbstractScriptEngine
JSR-223 API: optional engine facilities

- Compilable
- Invocable
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
Writing your own language: how and where to start?

- Sometimes the biggest hurdle is getting started
- JSR-223 is a friendly, welcoming specification
- DSLs are only and exactly as difficult to write as you decide they will be
Start at the end

➢ What should the eventual script look like?
   • Syntax
   • Semantics

➢ What features do you want to support?
   • What are the nouns that will be important?
   • What verbs – what actions?
An example script

showMap zipcode=22207
setZoomLevel 15
showIcons schools
showIcons libraries
plotRoute mySavedRoute
Begin implementation with AbstractScriptEngine

> Have createBindings() return a new SimpleBindings
> Consolidate the two remaining eval() methods
  > private Object eval(final Reader reader, final Bindings bindings, final ScriptContext context)
  > throws ScriptException
> Implement the new eval() method
  > Start with “Hello World”
  > Write a test that calls all of the different public eval() methods
Write your ScriptEngineFactory

- None of its methods require any computation!
- Two of its methods deal with your script’s syntax
  - `getMethodCallSyntax(…)` requires some thought, but not much
  - `getProgram(…)` requires even less thought
- Write a test to grab the engine from the factory and execute a script
Package it properly

> Use the service-provider convention for auto discovery
  - META-INF
    - /services
      - /javax.script.ScriptEngineFactory
      - com.my.script.MyScriptEngineFactory

> Write a test to get your ScriptEngineFactory via the ScriptEngineManager
Ummm… you’re now JSR-compliant!

▷ You’re almost done!

▷ You now have to change your new ScriptEngine’s private eval() method to do more than “Hello World”
  • Unless your intention was to have a HelloWorldScriptEngine
  • There’s nothing wrong with that, by the way
Advanced parsing

➤ If your language is complex, separate/encapsulate the parse and the execute concerns
  • Try ANTLR for parsing
  • Consider implementing Invocable and Compilable if you have a compilation step separate from the execution

➤ However, a complicated language means fewer users
Extending your interpreter

➤ Want to get fancy?
  • Support variables
  • Allow output redirection
  • Allow input redirection
  • Provide extensive error messages (ScriptException supports line-number support)
  • …what else…?

➤ What implicit procedures and variables will you expose to all scripts?
  • Unique ID generator?
  • Logging?
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
Case studies

- John & David are Java platform programmers
- John’s into video games
- David’s into music
- They understand the basics of the JSR
David’s MP3 player

filter band=Wolfhounds
shuffle on
repeat on
play
The Gaming Revolution

➤ The Next Big Thing
The Old School
The New School
The Gaming Revolution

➢ I want to make a language that asserts the basic principles of game play

➢ I wish I could cheat and write something like this:
   • Jump once
   • If I’m near the top of my jump, then assert that I can jump again
   • If I’m not, then assert that I can’t jump

➢ Can’t we all just get along

➢ If I need a test case for my test case, it’s too complicated
Enter Gameplan:
The Gameplay Description Language

- **Do**
  - Simple commands like “JUMP”
  - Assignments like Health=25

- **Wait – {}**
  - Simple commands like “{Standing}”
  - Assignments like {Height<300}
  - Comparisons like {ShotsInTheAir=0}
  - Sleep like {2000}

- **Assert – []**
  - Simple commands like “[JUMPING]”
  - Assignments like [Ammo = 100]
  - Comparisons like [Lives > 0]
Example Gameplan file

# Double jump
jump {nearpeak} jump [height>300]

# Can’t jump when you’re rolling
roll [rolling] {200} jump [height=0]

# Shooting
maxshotsintheair=1 shoot [shotsintheair=1] shoot shoot
    [shotsintheair=1]

# Upgrade
maxshotsintheair=4 shotsintheair=0 shoot
    [shotsintheair=1] shoot shoot [shotsintheair=3]
Verisign Gameplan Implementation (VGI)

- One object method
  - GamplanScriptEngine.eval(String, context)
- One properties file
  - Gameplan.definition.txt
- One Registration file
  - javax.script.ScriptEngineFactory
public Object eval(String script, ScriptContext context) throws ScriptException {

    // Start the Game object

    // Foreach line of input

    // If it looks like [***]
    // Assert the condition inside

    // If it looks like {***}
    // Wait on the condition inside

    // Otherwise
    // Change the state of the Game object

    // Stop the Game object
}
Gameplan.definition.txt

script.extensions = .game .gp
script.mimetypes = text/plain application/gameplan
script.name.short = Gameplan
script.name.long = Gameplan - Gameplay Description Language
script.name.list = Gameplan gpln gp GDL
script.version = 1.0

engine.name = Verisign Gameplan Interpreter
engine.class = com.vrsn.script.GameplanScriptEngine
engine.version = 2.3
javax.script.ScriptEngineFactory

- /META-INF/services/javax.script.ScriptEngineFactory
  - com.vrsn.script.gameplan.GameplanScriptEngineFactory
VeriSign Gameplan Interpreter
The Gaming Revolution

Don’t worry, we’ve slowed down the game for demonstration purposes. The real version will run almost 10% faster!
The Gaming Revolution

I know what you’re thinking, does this game really revolutionize gaming…
John & David write their DSLs, get traction

- MP3-player users love scripting their players
- Videogame users love scripting their players
- Because they’re so easy to use, others besides intended users pick up on them
  - QA for MP3 player want to start using it
  - {…some people besides users…} for videogame wanna use it
Allow plugins

- Allow for people to add to the language dynamically via plugins
- Plugins implement callback for handling script commands

```java
public interface ScriptPlugin {
    handleLine (String scriptLine)
}
```

- Plugins register their triggering keywords with scripts.
- Interpreter routes triggered commands to plugins
Registration mechanism

# register ‘assert’ plugin with engine
plugin assert com.verisign.mp3.qa.Assertions

# validate that plugin is working
assert activeSongCount=0
# validate filter features
filter genre=downtempo
assert activeSongCount=33
assert activeGenre=downtempo
saveFilter myTestFilter
clearFilter
assert activeSongCount=0
loadFilter myTestFilter
assert activeSongCount=33
assert activeGenre=downtempo
Videogame marketing folks

- you can have your Java platform code generate and execute such scripts while running
- you can save yourself compilation, build, packaging, and deployment cycles by separating the dynamic logic from the application proper
Synergy between music fans and gamers

> What gamer doesn’t want to control the game’s soundtrack?
> Gamers want to use their own music
> Gamers want the right music for the right moment
The beauty of JSR-223 compliance

- Because the scripting languages share common interfaces…
- Because the JSR specifies a discovery mechanism…
- Because the calling program doesn’t need to know anything about the language…
- The languages become interoperable
Combination script

```{ some Gameplan stuff }
[[[ mp3
filter band=Wolfhounds
shuffle on
repeat on
play
]]]
```
```
{{ more Gameplan }}
```
Agenda

- Background on JSR-223, scripting, and DSLs
- Domain Specific Languages in detail
- Walk-through of JSR-223
- Writing your own DSL
- DSL Case Studies
- Jumpstart your development
Introducing ScriptBlend

- Non-intrusive means of embedding different languages in the same script
- Scripts can share variables with one another
- Foreign script snippets can be inserted into flows midstream
ScriptBlend: Midstream insertion

```groovy
for (i in 1..5) {
  ...
}

def groovyHi = "hi from groovy";
scriptGlue.publish("groovyHi", groovyHi);
```

```bash
echo "$groovyHi"
```
More on ScriptBlend

- Supports nesting of foreign scripts (e.g., JavaScript™ technology snippet inside a Groovy snippet inside a Beanshell snippet)
- Provides a simple way to enable the ScriptEngineManager, individual ScriptEngineFactories, and individual ScriptEngines to interpret blended scripts
- Possible Eclipse plugin for creating and editing blended scripts
- More to follow…?
Excited to get started scripting?

- We have a starter kit to get you going
- Simplifies an already-friendly API
Starter Kit

- Implementations of JSR-223 must provide a factory and engine, but some common functionality can be assumed:
  - VerisignScriptEngineProperties
    - Loads properties from a resource
  - VerisignScriptEngineFactory
    - Loads Engine specified in the properties
  - VerisignScriptEngine
    - Reads input scripts intelligently
    - Funnels all eval methods into one which you implement

- All that’s left is to dream up the language
Presentation Map

- DSL
- JSR-223
- John and David’s Simple Framework
- Music Interaction Language
- VGI VeriSign Gameplan Interpreter
- Advanced Techniques
- Gameplan Examples
- Where to go from here

Gameplan
- Gameplay Description Language
Summary

- Domain Specific Languages provide portable power in a simple box that even the business guys can understand.

- Build a rocket ship with a one page manual that your customers can operate.
For More Information

List

• Cross-references to other sessions
• BOFs
• URLs
• Related books, etc.
Image Attributions

Creative Commons License
- First Base Flickr user timailius
- Second Base Flickr user billaday
- 1 out Flickr user newyork808
- Batter Flickr user pingnews
- Fly ball Flickr user foreverdigital
- Dropped ball Flickr user hobbes8calvin
- Double play Flickr user Michael (mx5tx)
- Umpire Flickr user Tom Clifton
- Deep Blue Flickr user davespilbrow
- Chess Pieces Flickr user Shefaet
- Earth Flickr user susemueller
- Silhouette Flickr user jekkyl
- Fisheye Flickr user ValterJacinto
- Baseball Flickr user B Tal

Stock Xchng, Public License
- Thunderbirds2 Rocketship
THANK YOU

David Smith, Software Engineer
John Colosi, Software Engineer

TS-5693