Cross-language Development Experience on the JVM

Dmitry Jemerov
Development Lead

Ilya Sergey
Software Developer
Speaker’s qualifications

- Dmitry Jemerov leads the development of RubyMine and the Ruby plugin for IntelliJ IDEA, and has also worked on the Python plugin.

- Ilya Sergey is one of the core developers of the Groovy and Scala plugins for IntelliJ IDEA.
Overall Presentation Goal

Learn about the advantages and pitfalls of using Ruby, Python, Groovy and Scala as part of large Java projects
Where We Come From

- IntelliJ IDEA is a huge (~3M LOC) desktop Java app
- Lots of complex Java APIs
- Developing custom language plugins partly in the languages themselves
- Dogfood is a major driver
- Letting users extend the IDE more easily
Why Use Multiple Languages

- Saying more by typing less
- Low-ceremony languages
- Expressive high-level constructs
  - Closures (Ruby blocks, Python lambdas)
  - List and map literals
- And much more
Why Use Java, then?
Why Use Java, then?

- Statically typed interfaces are great for defining a framework
- Easy to jump in any part of code and see what things exist around you and what you can do with them
- Performance of low-level parts of the system is essential
Scala?
Scala?

- Best of both worlds
  - Statically typed
  - No loss of performance compared to Java
  - Low ceremony

- Scary
  - Lots of special cases in the language
  - Can run into problems which are difficult to figure out
  - Match expressions as bug-prone as dynamic types
Script/Java Integration
Script/Java Integration

- Implementing interfaces defined in Java
- Generic types on the Java side
- Calling Java methods
Script/Java Integration

- Implementing interfaces defined in Java
  - Generic types on the Java side
- Calling Java methods
- Creating Java objects
Script/Java Integration

- Implementing interfaces defined in Java
  - Generic types on the Java side
- Calling Java methods
- Creating Java objects
- Creating typed Java arrays
Getting Script to Run
Getting Script to Run

- javax.scripting API
Getting Script to Run

- `javax.scripting` API
- Language-specific embedding APIs
  - Ruby, Python
- Cross-compilation and direct calls from Java
  - Groovy, Scala
- No cross-compilation because no way to make definitions visible to Java
Which Scripts to Run?
Which Scripts to Run?

• Running all scripts in predefined set of folders
  • Great for user extensions
  • Not lazy
  • Difficult to implement reloading

• Defining components implemented in script directly in plugin manifest files
  • On-demand loading
  • Reloading on change
<idea-plugin>

<extensions>

<codeInsight.overrideMethod
    language="ruby"
    factoryClass="o.j.r.RubyExtensionFactory"
    factoryArgument="/rb/
        ruby_override_handler.rb"
    implementationClass="RubyOverrideHandler"/>

...


<idea-plugin>
  <extensions>
    <codeInsight.overrideMethod
      language="ruby"
      factoryClass="o.j.r.RubyExtensionFactory"
      factoryArgument="/rb/ruby_override_handler.rb"
      implementationClass="RubyOverrideHandler"/>
  </extensions>
</idea-plugin>
<idea-plugin>

<extensions>

<codeInsight.overrideMethod
    language="ruby"
    factoryClass="o.j.r.RubyExtensionFactory"
    factoryArgument="/rb/
ruby_override_handler.rb"
    implementationClass="RubyOverrideHandler"/>

...


API Facade
Requirements of good Java class design are opposite to requirements of idiomatic Ruby/Python/Groovy
Requirements of good Java class design are opposite to requirements of idiomatic Ruby/Python/Groovy

- Java: decoupling; many small classes
- Script: aggregation; all necessary methods under one class
- Java: implementing interfaces
- Script: passing around closures
API Facade

- Requirements of good Java class design are opposite to requirements of idiomatic Ruby/Python/Groovy
  - Java: decoupling; many small classes
  - Script: aggregation; all necessary methods under one class
- Java: implementing interfaces
  - Script: passing around closures
- Need to create script-friendly facades that wrap IntelliJ IDEA APIs (mostly well-designed from Java POV)
class StringToSymbolIntention
  include IntentionAction

  def getFamilyName
    "StringToSymbol"
  end

  def getText
    RBundle.message("ruby.intentions.string.to.symbol", []).to_java
  end

  def isAvailable(project, editor, psi_file)
    canIntent?(base_string_at(psi_file, editor))
  end

  def invoke(project, editor, psi_file)
    string = base_string_at(psi_file, editor)
    symbol = RubyPsiUtil.getTopLevelElements(project, "#{string.content}")[0]
    RubyPsiUtil.replaceInParent(string, [symbol].to_java(:'com.intellij.psi.PsiElement'))
  end

  def startInWriteAction
    true
  end

private

  def base_string_at psi_file, editor
    PsiHelper.get_element_at editor, psi_file, RBaseString
  end

  def canIntent?(string)
    return false unless string
    !string.hasExpressionSubstitutions and TextUtil.isCID string.content
  end
class StringToSymbolIntention
  include IntentionAction

  def getFamilyName
    "StringToSymbol"
  end

  def getText
    RBundle.message("ruby.intentions.string.to.symbol", []).to_java
  end

  def isAvailable(project, editor, psi_file)
    canIntent?(base_string_at(psi_file, editor))
  end

  def invoke(project, editor, psi_file)
    string = base_string_at(psi_file, editor)
    symbol = RubyPsiUtil.getTopLevelElements(project, ".\{string.content\}\[0\]
    RubyPsiUtil.replaceInParent(string, [symbol].to_java(:'com.intellij.psi.PsiElement'))
  end

  def startInWriteAction
    true
  end

private

  def base_string_at psi_file, editor
    PsiHelper.get_element_at editor, psi_file, RBaseString
  end

  def canIntent?(string)
    return false unless string
    !string.hasExpressionSubstitutions and TextUtil.isCID string.content
  end
end

• Three getter methods
• Availability check
• Execute the intention
register_intention_action "Convert string to symbol",
  :category => "Ruby",
  :description => "Converts a string into a symbol",
  :before => "puts <spot>"Foo"</spot>",
  :after => "puts :Foo" do |context|
  s = context.element_at_caret(RBaseString)
  if s and not s.has_expression_substitutions and TextUtil.isCID s.content
    context.action do
      s.replace context.create_element("#:\{s.content\}")
    end
  end
end
end
register_intention_action "Convert string to symbol",
    :category => "Ruby",
    :description => "Converts a string into a symbol",
    :before => "puts <spot>"Foo"</spot>",
    :after => "puts :Foo" do |context|

s = context.element_at_caret(RBaseString)
if s and not s.has_expression_substitutions and TextUtil.isCID s.content

  context.action do
    s.replace context.create_element(":#{s.content}")
  end

end

end
Debugging
Debugging

- If scripts are compiled to Java classes, JVM debugging infrastructure can be used as is
  - Works for Groovy and Scala
- Infrastructure for debugging interpreted scripts is different
  - Python pdb, Ruby rdebug
- No way to integrate with JVM debugger
If scripts are compiled to Java classes, JVM debugging infrastructure can be used as is.

- Works for Groovy and Scala

Infrastructure for debugging interpreted scripts is different:

- Python pdb, Ruby rdebug
- No way to integrate with JVM debugger

IDEA uses interpreted Ruby and Python scripts => have to use print statements for debugging 😞
Unit Testing
Unit Testing

Unit Testing

- IntelliJ IDEA tests run under custom JUnit-based test runner which doesn’t know about script
Unit Testing

- IntelliJ IDEA tests run under custom JUnit-based test runner which doesn’t know about script
- Tests for IDEA script code are mostly written in Java
- Tests mostly data-driven, so not much pain
IDE Integration

- Navigation from script to Java
- Problem: overload resolution
- Completion of Java methods
IDE Integration

- Navigation from script to Java
- Problem: overload resolution
- Completion of Java methods
- Auto-import for Java classes
IDE Integration

- Navigation from script to Java
- Problem: overload resolution
- Completion of Java methods
- Auto-import for Java classes
- Generation of interface implementation stubs
IDE Integration

- Navigation from script to Java
- Problem: overload resolution
- Completion of Java methods
- Auto-import for Java classes
- Generation of interface implementation stubs
- Navigation from Java interfaces to script implementations
IDE Integration

- Navigation from script to Java
- Problem: overload resolution
- Completion of Java methods
- Auto-import for Java classes
- Generation of interface implementation stubs
- Navigation from Java interfaces to script implementations
- Cross-language refactoring: rename, change signature
Java/Ruby Integration with IntelliJ IDEA Ruby plugin
Make it Groovy!

• Why migrate from Java to Groovy?
• How to do it seamlessly?
• What does IntelliJ IDEA offer?
  • Simple file rename
  • “Dynamize” invalid Java-like code
  • Various intentions to make code good-looking

www.devoxx.com
“Dynamize” invalid code

```java
public class MyClass {
    void doSomething() {
        new Runnable() {
            public void run() {
                System.out.println("Hello, world!");
            }
        }.run();
    }
}
```
“Dynamize” invalid code
“Dynamize” invalid code
Groovy-aware intentions (1/2)

- Convert loop body to closure

```groovy
def list = [1, 2, 3]
for (elem in list) {
    println(elem)
}
```
Groovy-aware intentions (1/2)

- Convert loop body to closure
Groovy-aware intentions (2/2)

- Convert ternary operator to “Elvis”
• Convert ternary operator to “Elvis”
Groovy-based DSL tools

- Gradle
- GroovyRestlet
- GroovyLab
- Gant
- G...
includeTargets << gant.targets.Clean
cleanPattern << ['**/~/', '**/*.bak']
cleanDirectory << 'build'

target(stuff: 'A target to do some stuff.') {
    println('Stuff')
    depends(clean)
    echo(message: 'A default message from Ant.')
    otherStuff()
}

target(otherStuff: 'A target to do some other stuff') {
    println('OtherStuff')
    echo(message: 'Another message from Ant.')
    clean()
}

setDefaultTarget(stuff)
Groovy-based DSL tools
Groovy-based DSL tools

Groovy Runtime

DSLRunner.groovy
Groovy-based DSL tools
Groovy-based DSL support

- Code completion
- Debugger
- Groovy DSL Script
- Running script
- Navigation
Groovy-based DSL support

Groovy DSL Script
How to compile it all together?

Java

Groovy
Groovy approach: stubs

Just give to Java compiler what it does expect to see, even the fake facades.
Compilation with stubs

Java

Groovy
Compilation with stubs

Java

Groovy

Groovy Stubs
Compilation with stubs

Groovy

Bytecode
Compilation with stubs
Compiling Groovy with stubs

```xml
<target name="compile.all">

  <taskdef name="generateStubs"
    classname="org.codehaus.groovy.ant.GenerateStubsTask"
    classpathref="groovy.path"/>

  <mkdir dir="${stubsOutput}"/>

  <generateStubs destdir="${stubsOutput}" srcDir="${src.dir}"
    include name="**/*.groovy"/>
  <generateStubs destdir="${stubsOutput}" srcDir="${src.dir}"
    include name="**/*.java"
    classpath
    path refid="groovy.path"/>

</generateStubs>

  <javac destdir="${production.dest.groovy}" debug="on"
    src refId="javaToCompile"
    classpath refid="groovy.path"/>

</javac>

  <groovyc destdir="${project.dir}/test" srcDir="${project.dir}/test"
    include name="**/GroovyTest.groovy"
    classpath...>

</groovyc>

</target>
```
What about Scala?
Scala approach: *analysis*

Analyse alien code as it would have been our own...
Incremental compilation

- Compile modified and new files
- Compile dependent file set
Incremental compilation

- Compile modified and new files
- Compile dependent file set
Incremental compilation pitfalls

- Scala compiler infers top-level signatures
- Internal changes may propagate to other files

```scala
class A {
  def getFirstValue = "abc"
}

class B {
  def getSecondValue = 
    (new A).getFirstValue
}

class C {
  def getThirdValue = 
    (new B).getSecondValue.length
}
```
Incremental compilation pitfalls

- Scala compiler infers top-level signatures
- Internal changes may propagate to other files

```scala
class A {
    def getFirstValue = "abc"
}
```

```scala
class B {
    def getSecondValue = 
        (new A).getFirstValue
}
```

```scala
class C {
    def getThirdValue = 
        (new B).getSecondValue.length
}
```
Incremental compilation pitfalls

- Scala compiler infers top-level signatures
- Internal changes may propagate to other files

```scala
class A {
  def getFirstValue = 42
}

class B {
  def getSecondValue =
    (new A).getFirstValue
}

class C {
  def getThirdValue =
    (new B).getSecondValue.length
}
```
Incremental compilation pitfalls

- Scala compiler infers top-level signatures
- Internal changes may propagate to other files

```scala
class A {
  def getFirstValue = 42
}

class B {
  def getSecondValue =
    (new A).getFirstValue
}

class C {
  def getThirdValue =
    (new B).getSecondValue.length
}
```
Scala Solution

- Keep looking for new dependencies until no more affected files are found
Scala Solution

Keep looking for new dependencies until no more affected files are found

```scala
class A {
    def getFirstValue = 42
}

class B {
    def getSecondValue =
        (new A).getFirstValue
}

class C {
    def getThirdValue =
        (new B).getSecondValue.length
}
```
Scala Solution

Keep looking for new dependencies until no more affected files are found

```scala
class A {
  def getFirstValue = 42
}

class B {
  def getSecondValue = (new A).getFirstValue
}

class C {
  def getThirdValue = (new B).getSecondValue.length
}
```

1st pass
Scala Solution

Keep looking for new dependencies until no more affected files are found

```
class A {
    def getFirstValue = 42
}
```

```
class B {
    def getSecondValue = (new A).getFirstValue
}
```

```
class C {
    def getThirdValue = (new B).getSecondValue.length
}
```

1st pass

2nd pass
Recognizing Scala Bytecode

- Scala-specific data stored in ScalaSignature attribute
- It might be
  - Type aliases
  - Compound types
  - Variance annotations
- Java bytecode processing tools may need adjustment
Scala Decompiled Declarations

Java top-level

```java
package scala;

public final class Predef {

    public static final <T> java.lang.Class<T> classOf() { /* compiled code */ }
    public static final java.lang.Boolean boolean2Boolean(boolean b) { /* compiled code */ }
    public static final java.lang.Double double2Double(double v) { /* compiled code */ }
    public static final java.lang.Float float2Float(float v) { /* compiled code */ }
    public static final java.lang.Long long2Long(long l) { /* compiled code */ }
}
```

Scala top-level

```scala
package scala

object Predef extends java.lang.Object with scalaScalaObject{
    def this() = { /* compiled code */ }
    type byte = scala.Byte
    type short = scala.Short
}
```
Scala unpickler for IntelliJ IDEA

• Thanks to Andrew Foggin

• IntelliJ IDEA has its own Scala unpickler

• Top-level signatures of compiled Scala files are obtained now by parsing ScalaSignature attribute

• Produced text is treated as usual Scala file

• In other words, we have **scalap** implementation for internal usage
Groovy and Scala Benchmark
Summary

- Using JVM-based languages to implement the business logic of your app makes it more concise and easy to modify.

- Cross-language development support in IntelliJ IDEA makes it easy to write and maintain code in multiple languages within the same project.

- Some pieces of the infrastructure (like cross-language debugging) still need more tool support.
Tired of the verbosity of Java?

There’s plenty of alternatives today. Pick one and start using it in your project!
Q&A

Dmitry Jemerov
yole@jetbrains.com

Ilya Sergey
ilya.sergey@jetbrains.com
Thanks for your attention!

IntelliJ IDEA: [http://jetbrains.com/idea](http://jetbrains.com/idea)
Ruby plugin: [http://plugins.intellij.net/plugin/?id=1293](http://plugins.intellij.net/plugin/?id=1293)
Scala plugin: [http://plugins.intellij.net/plugin/?id=1347](http://plugins.intellij.net/plugin/?id=1347)
Python plugin: [http://plugins.intellij.net/plugin/?id=631](http://plugins.intellij.net/plugin/?id=631)