JVM dynamic languages metaobject protocol

Attila Szegedi
Chief Architect, Adeptra Inc.

JVM Language Summit, Santa Clara, 25th September 2008

What is this about?

- It is about language interoperability within a single JVM instance

Technical goal

- Code written in one language should be able to:
  - call code written in another language
  - manipulate native objects of another language
- Identify as many other common cross-language concepts as possible (sequences, dictionaries)

Social goal

- Establish interoperability mindset in JVM language implementers.
- Create a collaboration platform.
- “If we write a specification that is ignored, we’re just fiction writers.” (Ian Hickson, on HTML5)

Social goal matters

- In long term, technical goal is the means to achieve the social goal.
- Even if the current technical approach would fail, as long as it starts the social process, I'll claim it's a success!

Idea vs. implementation

- The idea is more important, but...
- ... you also need something tangible to get people interested. Hence working code.
- The idea is permanent, the tangible implementation is expected to vary.

Current (“2007”) implementation

- There is a MOP interface with typical operations. Currently:
  - properties: get, put, has, delete, enumeration
  - invocation: with both named arguments and positional argument
  - type coercion

Externalized vs. internalized behavior

- Most languages expect objects to implement an interface in order to handle them:
  - Scriptable, IRubyObject, PyObject
- Runtimes invoke methods on interface:
- `x=obj.prop` → `((Scriptable)obj).get("prop")`

Externalized vs. internalized behavior

- The objects internalize the behavior
- With MOP, the behavior is externalized from objects into the MOP
- `x=obj.prop` → `mop.get(obj, "prop")`
- The object no longer needs to be aware of the language runtime accessing it
- No wrappers for POJOs etc. needed anymore.

Properties

- `x=obj.prop` → `mop.get(obj, "prop")`
- `obj.prop=x` →
  `mop.put(obj, "prop", x, mop)`
- `for(var in in obj){…}`→
  `Iterator<Map.Entry> it =`
  `mop.properties(obj)`
  or
  `mop.propertyIds(obj)`
- `x=obj[1]` → `mop.get(obj, 1)`

Calls

- `fn(x1, x2)` →
  `mop.call(fn, mop, x1, x2)`
- `fn(a: x1, b:x2)` →
  `mop.call(fn, mop, {a:x1, b:x2})`
- `obj.fn(x1, x2)` →
  `mop.call(obj, "fn", mop, x1, x2)`

Type coercion

- `print(obj)` →
  `mop.representAs(obj, String.class)`
- `if(obj)` →
  `mop.representAs(obj, Boolean.class)`

Composability

- Protocols are composable
- Special (Result enum) return codes instead of exceptions: `doesNotExist`, `noRepresentation`, `notWritable`, `noAuthority`.
- `noAuthority` is the key to composition

Composability

```
  public Object get(Object target, Object propertyId)
{
    for (MetaobjectProtocol mop : members) {
        Object res = mop.get(target, propertyId);
        if(res != Result.noAuthority) {
            return res;
        }
    }
    return Result.noAuthority;
}
```

Type coercion on call

```
public Object call(Object fn, CallProtocol cp, Object...
args) {
 for (MetaobjectProtocol mop : members) {
     Object res = mop.call(fn, cp, args);
     if(res != Result.noAuthority) {
         return res;
     }
 }
 return Result.noAuthority;
}
```

Type coercion on call

```
mop [
  jythonMop,
  beansMop
]
```

```
javaObj.fn(jyObj) →
mop.call(javaObj, "fn", mop, jyObj) →
beansMop.call(javaObj, "fn", mop, jyObj) →
  public void fn(boolean x)
mop.representAs(jyObj, Boolean.class) →
jythonMop.representAs(jyObj, Boolean.class)
```

Faster than linear dispatch

- We have an optimized composite MOP for cases where members can be selected based on target object's class

Initializing

```
import org.dynalang.mop.impl.*;

MetaobjectProtocol mop =
    StandardMetaobjectProtocol
        .createStandardMetaobjectProtocol();
```

- Gathers all MOP implementations in the classpath of current thread context class loader. Adds POJO/list/map MOPs at end.
- They need to be registered in `/META-INF/services/org.dynalang.mop.BaseMetaobjectProtocol`
- JAR file service configuration mechanism

Initializing with native MOP

```
import org.dynalang.mop.impl.*;

MetaobjectProtocol mop =
    StandardMetaobjectProtocol
        .createStandardMetaobjectProtocol(
            rhinoMop);
```

- Also gathers all from classpath, but:
- makes sure the specified MOP is first in the resulting composite.

Built-in POJO support

- BeansMetaobjectProtocol included
- Uses JavaBeans introspector
- Translates property access into getXx/isXx, setXx reflected calls
- MOP.call() handles overloaded and vararg methods compliant to JLS 15.12.2 algorithm
- Specific support for static methods and constructors (not in MOP API)

Access control

- Currently, only public methods/constructors are supported.
- Java package-private and protected concepts are hard to map:
  - when does the dynamic code live in same package as a Java class?
  - when does the dynamic code belong to a subclass of a Java class?

POJO call example

- Invoke method doIt with Object[] args:
  `mop.call(obj, "doIt", mop, args)`
- More efficient for repeated invocation with same arguments:

```
final DynamicMethod m = beanMop.getInstanceMethod("doIt");
final SimpleDynamicMethod sm;
if m instanceof OverloadedDynamicMethod) {
    sm = ((OverloadedDynamicMethod)m).getResolvedMethodFor(mop, args);
}
else
    sm = (SimpleDynamicMethod)m;
}
…
sm.call(obj, mop, args); // repeatable; handles vararg packing
```

2008: invokedynamic bootstrap dispatch

- Mechanism similar to the previous mechanism for obtaining a SimpleDynamicMethod could be used to obtain a MethodHandle.

invokedynamic bootstrap dispatch

- The JAR service discovery mechanism can be used for creating a composite of following interface implementers:

```
public interface DynamicInvocationBootstrapper {
    public Object bootstrapInvokeDynamic(
        CallSite site, Object receiver,
        Object... arguments);
}
```

- Method signature looks familiar, doesn't it?

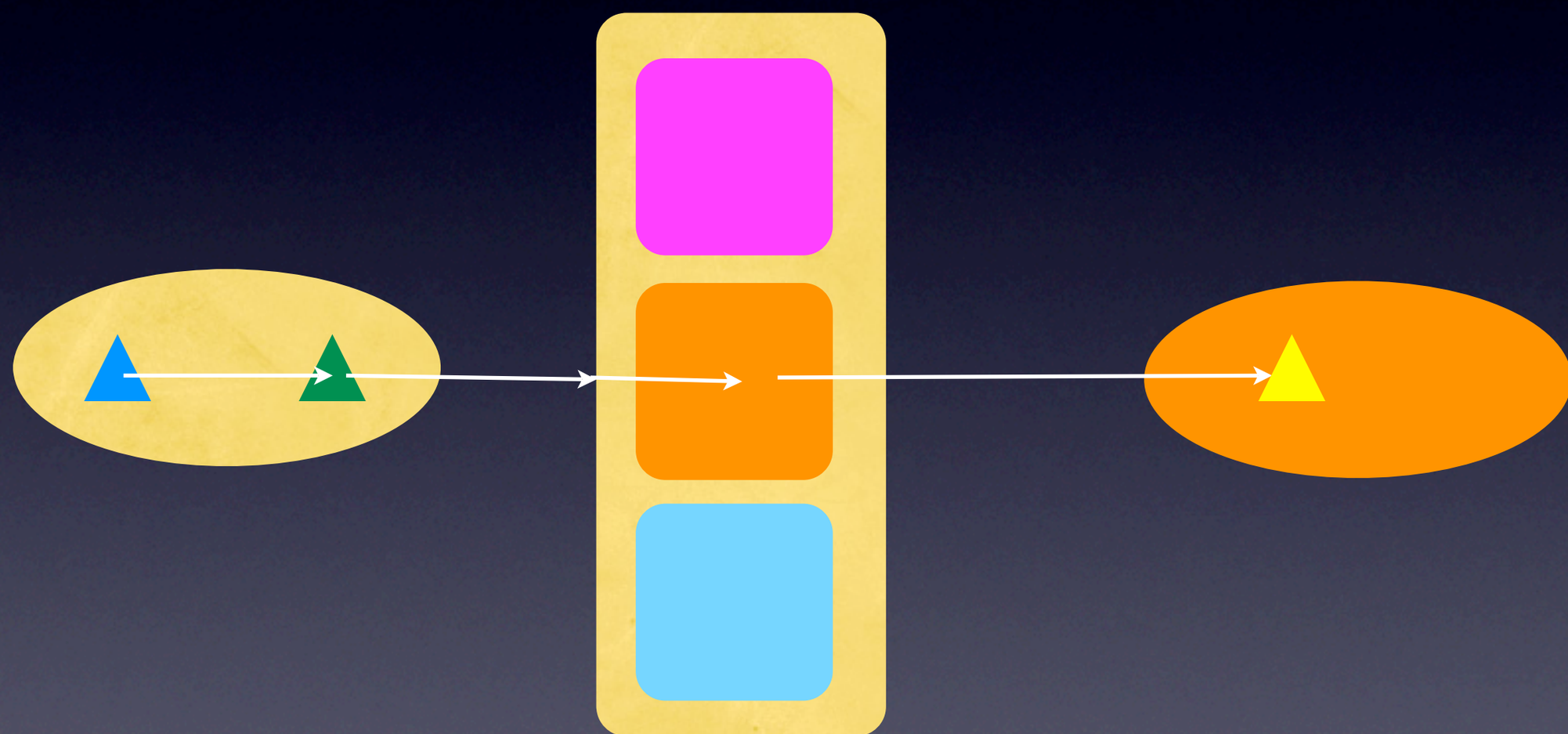
invokedynamic bootstrap dispatch

- Classes' own bootstrapper implementations would delegate to the composite, and it would in turn delegate to all discovered language-specific bootstrappers, until one succeeds.
- Opens way for cross-language bootstrapInvokeDynamic dispatch. Allows a JRuby compiled caller bytecode to bind to Jython method when receiver is a Jython object, etc.

Composability again

```
public Object bootstrapInvokeDynamic(CallSite site, Object receiver, Object... arguments) {
  for (DynamicInvocationBootstrapper dib : members) {
      Object res = dib.bootstrapInvokeDynamic(site, receiver, arguments);
      if(res != Result.noAuthority) {
          return res;
      }
  }
  return Result.noAuthority;
}
```

Using composite MOP to get method handle



POJO bootstrapper

- POJO bootstrapper is the last (fallback) in composite bootstrapper.
- It uses the logic similar to one for obtaining the SimpleDynamicMethod previously shown.
- After resolving overloads, it combines unreflect, convertArguments (if needed), collectArguments (if vararg method), and guardWithTest (to invalidate when type of the target changes)

Use invokedynamic for property access

- Possible approaches:
- Compile `obj.foo` as invocation of `obj.getFoo`?
  - Breaks down with `obj.Foo`
- Use some other naming convention?
  - `obj.foo → obj.getprop:foo`
  - `obj.Foo → obj.getprop:Foo`
- Invoke a generic get() method, pass property id as an argument; utilize dropArgument?

Use invokedynamic for property access

- Last two aren't mutually exclusive:
  - Use `obj.getprop:foo` when identifier is fixed in source code (`obj.foo`)
  - Compile generic `obj.get(someExpr)` when the identifier is not fixed in source code (`obj[someExpr]`), with a guardWithTest on name.

MOP summary: we have...

- Protocol for language independent property access on objects and for invocation of methods,
- designed for composability,
- Factory that builds an optimized composite MOP from all MOPs available to a class loader,
- MOP for POJOs, usable as a fallback when no other MOP matches. Handles overloaded methods, varargs, type conversions,

MOP summary: we need...

- to extend the protocol to cover other cross-language concepts:
  - arrays, associative arrays, etc.
- Solution for access control on Java package-private and protected access.

Summary: invokedynamic protocol

- ... is actually independent from current MOP effort, although they're twins brothers:
  - similar composable protocol for invokedynamic bootstrapping,
  - with similar factory mechanism that will create a composite from all accessible implementations,
  - with similar default protocol implementation for POJOs.

Your turn!

- Lots of decisions are not yet made.
- Everything is subject to discussion and change, so…
- … let's talk!