Java™

Square Pegs, Round Holes

or: How To Fit a Language On the JVM (Without a Hammer.)

- Attila Szegedi
  Principal Member of Technical Staff

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.

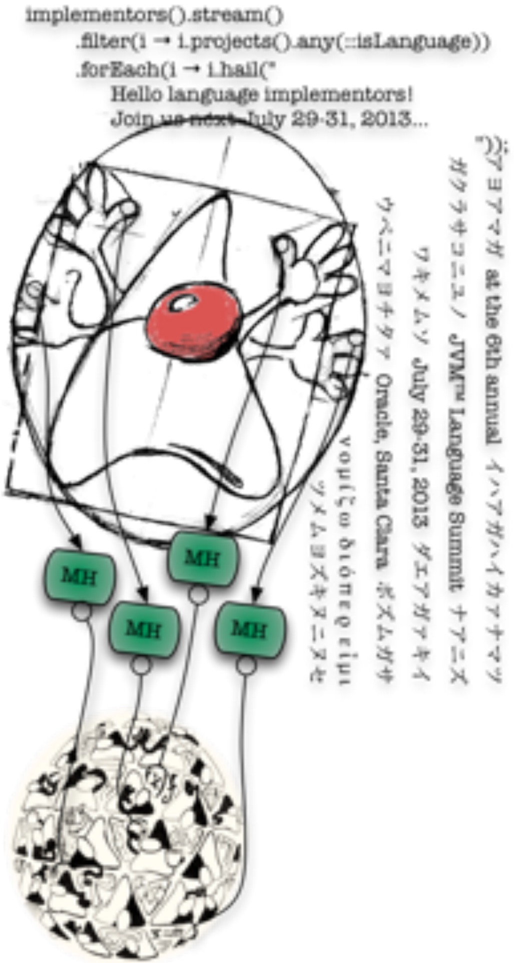
Program Agenda

- Extending Java Classes
- Type Conversion Fun
- Arrays
- The Incredible Package Illusion
- Linking
- Security

So, You Want To Write a Language on JVM

- You likely want to have your language somehow interface with the underlying platform: VM, libraries, etc.
- There are some typical things that you need to solve for a rich interop story.
  - Extending JVM classes and implementing interfaces in your language
  - Type conversions
  - Handling arrays (yes, it can be quite a special case)
  - Invocation of Java methods

Extending Java Classes



```
implementors().stream()
        .filter(i → i.projects().any(::isLanguage))
        .forEach(i → i.hail("
            Hello language implementors!
            Join ~~us next~~ July 29-31, 2013...
"));
```

アヨアマガ at the 6th annual イハアガハイカァナマツ
ガクラサコニュノ JVM™ Language Summit ナアニズ
ワキメムソ July 29-31, 2013 ダエアガァキイ
ウベニマヨキタア Oracle, Santa Clara ボズムガサ
νομίζω διόπερ εἰμι
ツメムヨズキヌニヌセ
MH
MH
MH
MH

Extend/Implement a JVM Class/Interface

- Typical Nashorn code:

```
var r = new java.lang.Runnable() {
   run: function() {
      print(“Hello!”)
   }
}
```

Extend/Implement a JVM Class/Interface

- Simpler Nashorn code:

```
var r = new java.lang.Runnable(function() {
    print(“Hello!”)
})
```

Adapter Classes

- Obviously, when you write code like this, we instantiate an adapter class.
- Question is: how is it supposed to look like when implemented around invokedynamic?

Anatomy of an Adapter

```
package jdk.nashorn.javaadapters.java.lang;

public final class Runnable implements java.lang.Runnable {

  private final ScriptObject global;

  private final MethodHandle run;
  private final MethodHandle toString;
  private final MethodHandle hashCode;
  private final MethodHandle equals;

  ...
}
```

Anatomy of an Adapter - Fields

Can’t define classes in java.* package

```
package jdk.nashorn.javaadapters.java.lang;

public final class Runnable implements java.lang.Runnable {

  private final ScriptObject global;

  private final MethodHandle run;
  private final MethodHandle toString;
  private final MethodHandle hashCode;
  private final MethodHandle equals;

  ...
}
```

Need defining context

Overridable toString, hashCode, equals

Anatomy of an Adapter - Constructors

```
public <init>(Ljava/lang/Object;)V
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V

    ALOAD 0
    ALOAD 1
    LDC "run"
    LDC ()V.class
    INVOKESTATIC JavaAdapterServices.getHandle();
    PUTFIELD run;
```

Repeated for every MethodHandle field

Anatomy of an Adapter - Constructor in Java

```
public Runnable(Object o) {
    super();
    this.run = JavaAdapterServices.getHandle(o, “run”,
        MethodType.methodType(void.class));
    this.equals = JavaAdapterServices.getHandle(o, “equals”,
        MethodType.methodType(boolean.class, Object.class));
    this.hashCode = JavaAdapterServices.getHandle(o, “hashCode”,
        MethodType.methodType(int.class));
    this.toString = JavaAdapterServices.getHandle(o, “toString”,
        MethodType.methodType(String.class));

    this.global = Context.getGlobal();
    this.global.getClass();
}
```

Anatomy of an Adapter

```
public static MethodHandle getHandle(Object obj, String name, MethodType type) {
    final ScriptObject sobj = (ScriptObject)obj;

    // Since every JS Object has a toString, we only override
    // "String toString()" it if it's explicitly specified
    if ("toString".equals(name) && !sobj.hasOwnProperty("toString")) {
        return null;
    }

    final Object fnObj = sobj.get(name);
    if (fnObj instanceof ScriptFunction) {
        return adaptHandle(
            ((ScriptFunction)fnObj).getBoundInvokeHandle(sobj), type);
    } else if(fnObj == null || fnObj instanceof Undefined) {
        return null;
    } else {
        throw typeError("not.a.function", name);
    }
}
```

Anatomy of an Adapter - Constructors

```
public <init>(ScriptFunction)V
   ALOAD 0
   INVOKESPECIAL java/lang/Object.<init> ()V

   ALOAD 0
   ALOAD 1
   LDC ()V.class
   INVOKESTATIC JavaAdapterServices.getHandle();
   PUTFIELD run

   ALOAD 0
   ACONST_NULL
   PUTFIELD toString
```

Single method handle for the function.

null for all other overrides

Anatomy of an Adapter - Constructor in Java

```
@Override
public Runnable(ScriptFunction f) {
    super();
    this.run = JavaAdapterServices.getHandle(f,
        MethodType.methodType(void.class));

    this.equals = null;
    this.hashCode = null;
    this.toString = null;

    this.global = Context.getGlobal();
    this.global.getClass();
}
```

Anatomy of an Adapter - Constructors

- A public constructor is emitted for every superclass public or protected constructor.
- We add Object at the end of superclass constructor signature.
  - Yes, we convert variable arity constructors into fixed arity.
  - Allows us to use the `new T(x, y) { ... }` syntax extension.
- If we're implementing a SAM, another constructor with ScriptFunction as its final argument is emitted too.
- Dynalink overloaded method resolution ensures the right constructor is picked up at run time.

Anatomy of an Adapter - Methods

- For an abstract method:

```
  public run()V
    ALOAD 0
    GETFIELD run;
    DUP
    IFNONNULL L4
    POP
    NEW java/lang/UnsupportedOperationException
    DUP
    INVOKESPECIAL UnsupportedOperationException.<init> ()V
    ATHROW
L4  INVOKEVIRTUAL java/lang/invoke/MethodHandle.invokeExact
    RETURN
```

Anatomy of an Adapter - Methods

- For a non-abstract method:

```
  public toString()Ljava/lang/String;
    ALOAD 0
    GETFIELD toString
    DUP
    IFNONNULL L4
    POP
    ALOAD 0
    INVOKESPECIAL java/lang/Object.toString;
    ARETURN
L4  INVOKEVIRTUAL java/lang/invoke/MethodHandle.invokeExact
    ARETURN
```

Anatomy of an Adapter - Methods

- Exception handling

```
public toString()Ljava/lang/String;
  TRYCATCHBLOCK L0 L1 L3 java/lang/RuntimeException
  TRYCATCHBLOCK L0 L1 L3 java/lang/Error
  TRYCATCHBLOCK L0 L1 L2 java/lang/Throwable
  ...
 L2
  NEW java/lang/RuntimeException
  DUP_X1
  SWAP
  INVOKESPECIAL RuntimeException.<init>(Throwable)V
 L3
  ATHROW
```

How often do you get to use this opcode?

Anatomy of an Adapter - Method in Java

```
@Override
public boolean equals(Object o) {
    if(equals == null) {
        return super.equals(o);
    }
    try {
        return equals.invokeExact(this, o);
    } catch(RuntimeException|Error) {
        throw e;
    } catch(Throwable t) {
        throw new RuntimeException(t);
    }
}
```

Anatomy of an Adapter - Method in Java, real

```
@Override
public boolean equals(Object o) {
    if(equals == null) {
        return super.equals(o);
    }
    final Global currentGlobal = Context.getGlobal();
    final boolean differentGlobal = currentGlobal != global;
    if(differentGlobal) {
        Context.setGlobal(global);
    }
    try {
        return equals.invokeExact(this, o);
    } catch(RuntimeException|Error) {
        throw e;
    } catch(Throwable t) {
        throw new RuntimeException(t);
    } finally {
        if(differentGlobal) {
            Context.setGlobal(currentGlobal);
        }
    }
}
```

Anatomy of an Adapter

- MethodHandle objects - the behavior - is instance bound.
- What if you want class bound?

```
var Hello = Java.extend(java.lang.Runnable, {
   run: function() {
      print(“Hello!”)
   }
})

var h1 = new Hello()
var h2 = new Hello()
```

Anatomy of an Adapter - Class Behavior

```
package jdk.nashorn.javaadapters.java.lang;

public final class Runnable implements java.lang.Runnable {

  private final ScriptObject global;
  private static final ScriptObject staticGlobal;

  private final MethodHandle run;
  private final MethodHandle toString;
  private final MethodHandle hashCode;
  private final MethodHandle equals;

  private static final MethodHandle run_static;
  private static final MethodHandle toString_static;
  private static final MethodHandle hashCode_static;
  private static final MethodHandle equals_static;
  ...
```

Anatomy of an Adapter - Class Behavior

Pass parameter to static block in a thread local.

```
static {
    Object o = JavaAdapterServices.getClassOverrides();
    if(o instanceof ScriptFunction) {
        run_static = JavaAdapterServices.getHandle((ScriptFunction)o, …);
        hashCode_static = null;
        ...
    } else {
        run_static = JavaAdapterServices.getHandle(o, “run”, …);
        hashCode_static = JavaAdapterServices.getHandle(o, “hashCode”, …);
        ...
    }
}
```

Anatomy of an Adapter - Class Behavior

```
@Override
public boolean equals(Object o) {
    try {
        if(equals != null) {
            return equals.invokeExact(o);
        } else if(equals_static != null) {
            return equals_static.invokeExact(o);
        }
    } catch(RuntimeException|Error) {
        throw e;
    } catch(Throwable t) {
        throw new RuntimeException(t);
    }
    return super.equals(o);
}
```

Anatomy of an Adapter - Class Behavior

- Of course, actual code is more complex, as it needs to deal with management of appropriate global too.
- Up to three constructors emitted for every superclass constructor:
  - One with same arguments as superclass constructor
  - One with added Object argument for instance overrides
  - One with added ScriptFunction argument for SAM instance override.

Overrides and Overloads in Adapters

- JavaScript has no concept of overloaded methods.
- Our adopted policy is: a named function is used as the implementation of all non-final overloads with the same name.
- True for adapters written in JavaScript; if your language could distinguish between overloads, feel free to write a different adapter.

Adapters and Security

- Adapters can only extend/implement public classes/interfaces.
  - Classes/interfaces in restricted packages subject to access check.
- Can only override public and protected methods.
  - `@CallerSensitive` methods can't be overridden as it'd mess up the caller identification.

Adapters and Security

- Adapters are defined in a separate ProtectionDomain with AllPrivileges.
  - So as to not narrow the caller privileges.
  - They are just pass-through delegates, so no risk of privilege escalation.
  - They <u>don't</u> use doPrivileged blocks.
  - Effective permissions are the intersection of permissions of caller and delegate function.

Adapters and Security

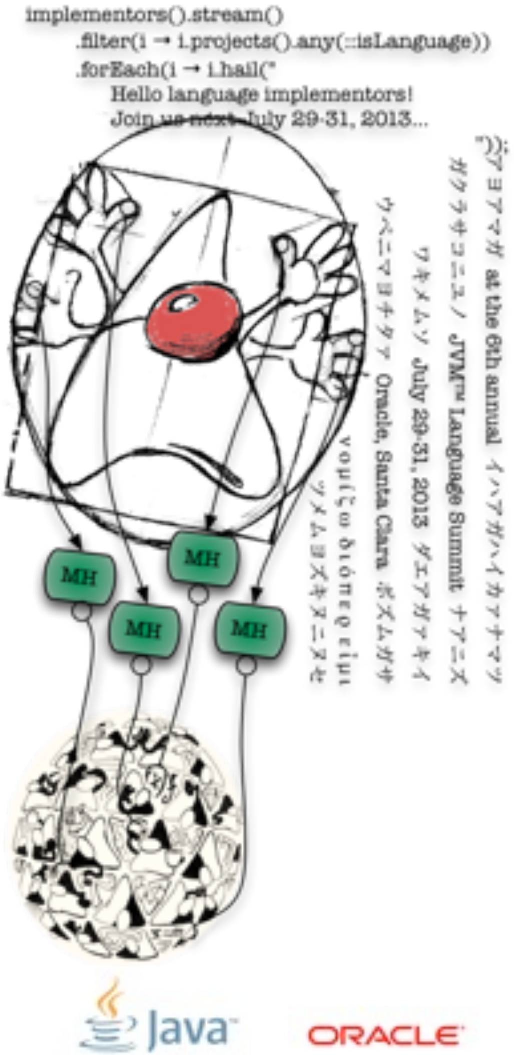
```
private static final ProtectionDomain GENERATED_PROTECTION_DOMAIN =
    createGeneratedProtectionDomain();

private static ProtectionDomain createGeneratedProtectionDomain() {
    final Permissions permissions = new Permissions();
    permissions.add(new AllPermission());
    return new ProtectionDomain(
        new CodeSource(null, (CodeSigner[])null), permissions);
}

...
defineClass(name, classBytes, 0, classBytes.length,
    GENERATED_PROTECTION_DOMAIN);
```

- Obviously, our own code needs “createClassLoader” permission to create a class loader to invoke defineClass in.

Where Should I Define Thee?

- What’s the parent class loader for your adapter class?
- Take the set of class loaders for the extended class and all implemented interfaces…
- …as well as the class loader for your language runtime classes.
- Find the “Maximum Visibility Loader” - one that sees classes in all the other loaders.
- If there’s no such loader, can’t define the adapter!
- Can cause surprises in exotic situations.

Type Conversion Fun



Ain’t No Script Like JavaScript…

- … for ultimate type conversion fun.
- ‘cause `[+!+[]]+[+[]] == 10`, of course!
- Most type conversions are straightforward; some are more interesting.
- We mostly encounter type conversions to target Java types when invoking Java methods and have to match parameters to their signatures.

Stuff that's almost trivial

```
var x = new (Java.type("boolean[]"))(1)
test(0)
test(1)
test({})
test([])
test("")
test("false")
test(null)
test(undefined)

function test(v) {
   x[0] = v
   print(JSON.stringify(v) + " => " + x[0])
}
```

```
0 => false
1 => true
{} => true
[] => true
"" => false
"false" => true
null => false
undefined => false
```

Stuff that's almost trivial

```
var x = new (Java.type("java.lang.Boolean[]"))(1)
test(0)
test(1)
test({})
test([])
test("")
test("false")
test(null)
test(undefined)

function test(v) {
   x[0] = v
   print(JSON.stringify(v) + " => " + x[0])
}
```

```
0 => false
1 => true
{} => true
[] => true
"" => false
"false" => true
null => null
undefined => null
```

- When converting to boxed type, we can preserve nulls.

Stuff that's nifty

- If the target is a SAM type, and you supply a ScriptFunction, we supply an on-the-fly allocated adapter.

```
Collections.sort(new function(x, y) { return y < x })
```

- All of this is handled with Dynalink linkers implementing the optional `GuardingTypeConverterFactory` interface's `GuardedInvocation getTypeConverter(Class from, Class to)` method.

Comparison prioritization

- Yet another Dynalink feature. To wit:

```
new Thread(new function() { print(“Hello!”) })
```

- Can apply both to `Thread(Runnable)` and sadly also to `Thread(String)` method, as JS has implicit object-to-string conversion.

Comparison prioritization

- By definition needed whenever your language allows more conversions than JLS does.
- ‘Cause you’ll end up widening the set of applicable overloaded methods at a call site.

```
public interface ConversionComparator {
    enum Comparison { TYPE1_BETTER, TYPE2_BETTER, INDETERMINATE }

    public Comparison compareConversion(Class sourceType,
        Class targetType1, Class targetType2);
}
```

Comparison prioritization

```
public class NashornLinker implements
    TypeBasedGuardingDynamicLinker,
    GuardingTypeConverterFactory,
    ConversionComparator {

    ...
    public Comparison compareConversion(Class sourceType,
        Class targetType1, Class targetType2) {
    if(ScriptObject.class.isAssignableFrom(sourceType)) {
        if(targetType1.isInterface()) {
            if(!targetType2.isInterface()) {
                return Comparison.TYPE_1_BETTER;
            }
        } else if(targetType2.isInterface()) {
            return Comparison.TYPE_2_BETTER;
        }
    }
    return Comparison.INDETERMINATE;
}
```

Handling Strings, Numbers, and Booleans

- JavaScript string, number, and boolean values are represented as Java String, Number, and Boolean objects. These are considered JS primitive types and handled as such.
- Still possible to invoke Beans methods on them, e.g. `“foo”.hashCode()`.

Primitives’ Conversion Prioritization

- Say you’re invoking bean.foo(“123”) and you have overloaded methods foo(int) and foo(Object).
- Three types in play: type of value “123” (String), and the types of formal parameters in the method.
- Which one do you choose?

Primitives' Conversion Prioritization

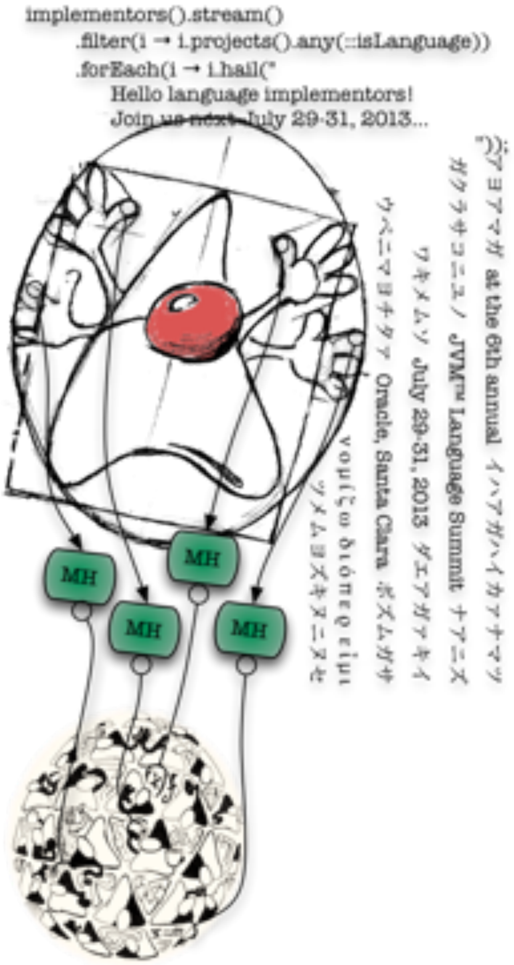
- Conversion prioritization turned out to be somewhat hairy:
  - If exactly one target type matches source type, pick it.
  - if exactly one target type is a number, pick it
  - if exactly one target type is char, pick it (number to UTF-16 conversion)
  - Between possible two number types, choose the wider one.
  - In all other cases, if one of the target types is string, choose it as strings can represent any value without precision loss.

SAM Type Conversion

- A.K.A. instant lambdas
- Pass a function in place of an argument expecting a SAM type…
- … it gets wrapped.

```
new Thread(function() { print(“Hello!”) }).start()
```

Arrays



```
implementors().stream()
        .filter(i → i.projects().any(::isLanguage))
        .forEach(i → i.hail("
            Hello language implementors!
            Join us next July 29-31, 2013...
"));
```

Java Array vs. Your Language’s Arrays

- JavaScript arrays are particularly nasty:
  - They can grow and shrink.
  - They can be sparse.
- If we provided an automatic conversion from JavaScript array to Java array, what would the semantics be?
  - Decision: we don’t provide automatic conversion!

Java Array vs. Your Language’s Arrays

- This won’t work:

```
java.lang.Array.toString([1, 2, 3])
```

- This will:

```
java.lang.Array.toString(Java.to([1, 2, 3]))
```

- Rare case of mandating explicit conversion.

Java Array vs. Your Language's Arrays

- API design is a tradeoff.
- No way we are making a copying operation with linear time and unbounded memory needs implicit.
- The programmer is also expected to understand there are no two-way updates between Java and JavaScript arrays.

Java Array vs. Your Language's Arrays

- `Java.to(jsObj[, clazz])` converts a JavaScript `Array` to a Java array, a `List`, or a `Deque`.
  - Default value for `clazz` is `Object[]`.
- `Java.from(javaObj)` converts a Java array or `List` to a JavaScript `Array`.
- Finally, it is perfectly possible to use Java arrays and lists in JS; `for..in` and `[]` syntax work on them.
- Only if you need functionality from the JS prototype (e.g. map, reduce, etc.) is when you need to explicitly use `Java.from` to get an actual JS `Array`.

Component Type Conversion

- `Java.to([12, false, “55”], int.class])` will use language conversion logic to int elementwise.

Component Type Conversion

```
public Object asArrayOfType(final Class<?> componentType) {
    final Object[] src = asObjectArray();
    final int l = src.length;
    final Object dst = Array.newInstance(componentType, l);
    final MethodHandle converter =
        linkerServices.getTypeConverter(Object.class, componentType);
    for (int i = 0; i < src.length; i++) {
        Array.set(dst, i, converter.invokeExact(converter, src[i]));
    }
    return dst;
}
```

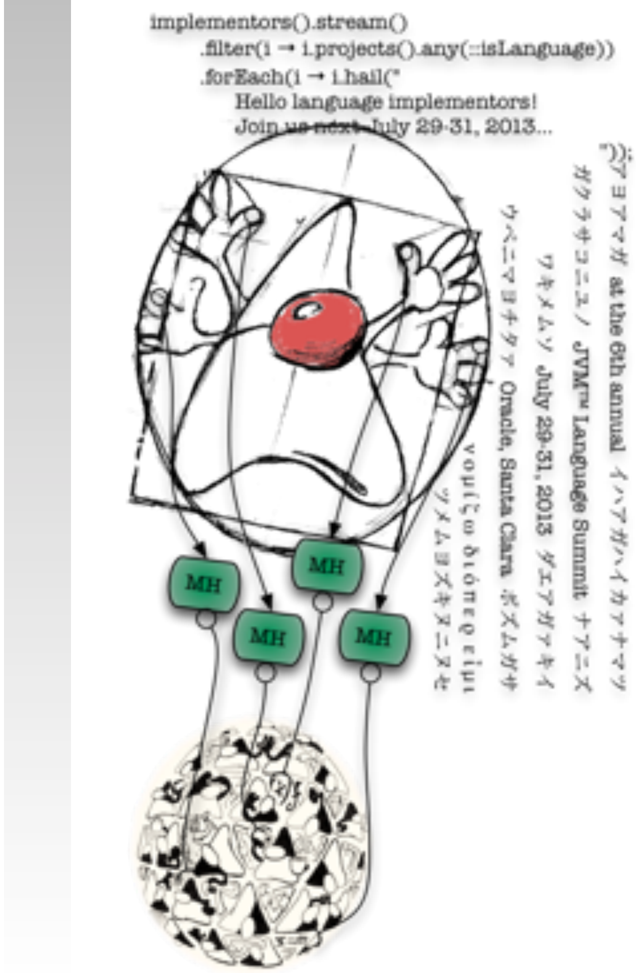
Java Objects are Maps, So Why Aren’t…

- Fun fact: `ScriptObject` implements `java.util.Map`.
- So, why doesn’t our `NativeArray` implement `java.util.List`?

Java Objects are Maps, So Why Aren’t…

- `NativeArray` extends `ScriptObject`, implicitly implements `Map`.
- Turns out no class can implement both `List` and `Map`:
  - `boolean List.remove(Object)`
  - `Object    Map.remove(Object)`
- Some battles you can’t win.

Statics


implementors().stream()
.filter(i → i.projects().any(::isLanguage))
.forEach(i → i.hail("
Hello language implementors!
Join ~~us next~~ July 29-31, 2013...
"));
at the 6th annual
JVM™ Language Summit
July 29-31, 2013
Oracle, Santa Clara
MH
MH
MH
MH

Class vs. Statics

- As a Java programmer, you understand the difference between `java.io.File` and `java.io.File.class`.
- One is a compile-time identifier providing access to constructors and static members. It is not reified at run-time.
- Other is a reified run-time representation of an object’s class.
- They are separate concepts.
- Probably shouldn’t mix them up in other languages either.

StaticClass

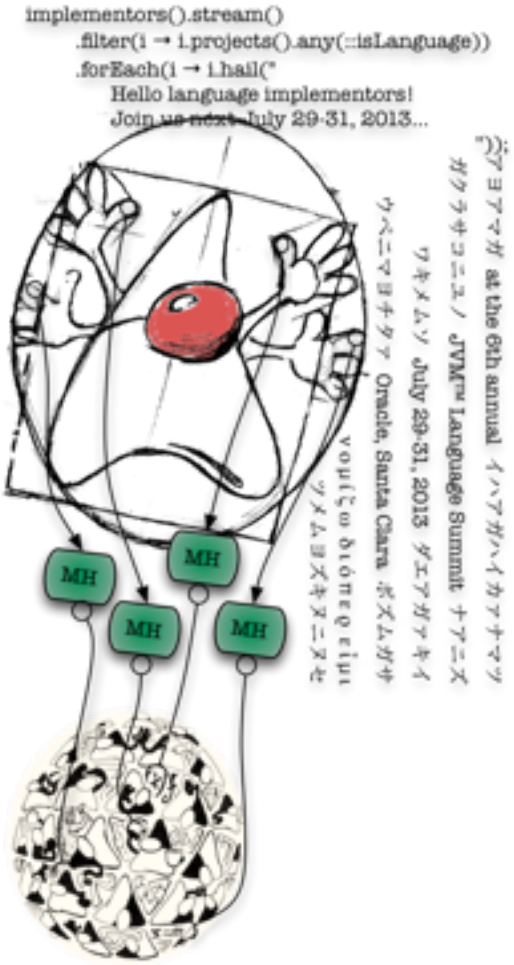
- Dynalink has a `StaticClass` class that is a reification of Java’s compile-time class identifier.
- It’s just a boring little wrapper around a `Class` object.
- However, the linker will correctly link to static members of the represented class when faced with such object.
- It will also link to constructors when linking `dyn:new` operation.

StaticClass

- `dyn:getProp:class` is also linked to retrieve the runtime `Class`.

-
  ```
  var fileClazz = Java.type(“java.io.File”).class // or
  var fileClazz = java.io.File.class // familiar, right?
  ```
- On the other hand, `dyn:getProp:static` is linked to static class from a `Class` object - a capability you don’t have in the Java language.
  ```
  var file = ... // somehow get a java.io.File
  file.class === java.io.File.class // like in Java
  file.class !== java.io.File // wouldn’t compile in Java!
  file.class.static === java.io.File // neither would this!
  file instanceof java.io.File // true: special handling
  ```

The Incredible Package Illusion



```
implementors().stream()
    .filter(i → i.projects().any(::isLanguage))
    .forEach(i → i.hail("
        Hello language implementors!
        Join us next July 29-31, 2013...
"));
```

アヨアマガ at the 6th annual イハアガハイカァナマツ
ガクラキヨニユノ JVM™ Language Summit ナアニズ
ワキメムソ July 29-31, 2013 ダエアガァキイ
ウベニマヨキタァ Oracle, Santa Clara ボズムガサ
νομίζω διόπερ εἰμι
ツメムヨズキヌニヌセ
MH
MH
MH
MH

Packages Aren’t Reified

- There’s no object in JVM that represents a package, e.g. `java.util`.
- `java.lang.Package` doesn’t count.
- Can’t verify existence of a package.
- Yet dynamic languages often want to give users stuff like:
  `var list = new java.util.List().`

Stepping Stones

- Typical solution: “stepping stones” approach: provide a `java` object, that provides a `util` object, that provides a `List` object.
- Problem: must optimistically presume any identifier to be a package when a class of that name is not found

  ```
  var PirateList = java.util.ArrrayList
  var list = new PirateList()
  ```

- Typos can end up being detected late; effort must be taken to report the right error.

Stepping Stones

```
@Override
protected GuardedInvocation findCallMethod(CallSiteDescriptor desc) {
    final MethodType type = desc.getMethodType();
    return new GuardedInvocation(MH.dropArguments(CLASS_NOT_FOUND, 1,
        type.parameterList().subList(1, type.parameterCount())),
        TYPE_GUARD);
}

@SuppressWarnings("unused")
private static void classNotFound(final NativeJavaPackage pkg) throws
ClassNotFoundException {
    throw new ClassNotFoundException(pkg.name);
}
```

Linking a Thrower vs. Throwing an Exception

- In previous example, we linked an exception throwing method handle.
- We could’ve also thrown the exception from linking code too.
- Design choice:
  - Linking an exception thrower eliminates linker plumbing frames from the call stack.
  - Throwing in-situ can help debugging the runtime because it does not eliminate those same stack frames.
    - Possible compromise: `Throwable.addSuppressed()`

Nashorn solution

- We support stepping stones, but try to steer users towards avoiding them.
- Preferred idiom is `Java.type()`.

  ```
  var List = Java.type(“java.util.ArrayList”)
  var list = new List()
  ```

- Can invoke it directly too, but a bit awkward because of call operator precedence:

  ```
  var list = new (Java.type(“java.util.ArrayList”))
  ```

Also supports arrays

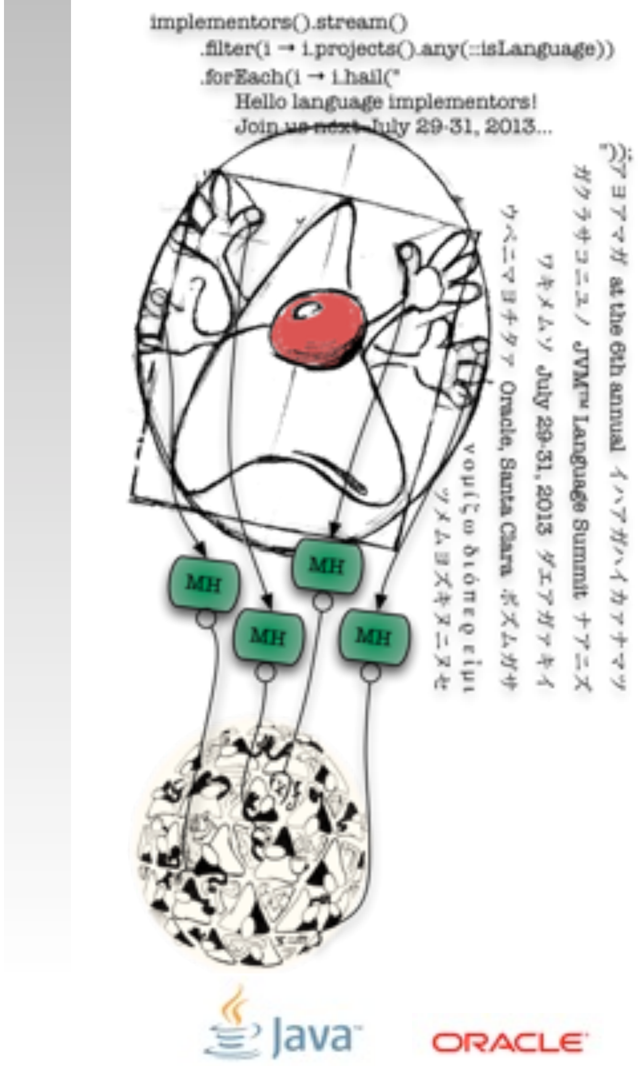
-
  ```
  var intArr5 = new (Java.type(“int[]”))(5)
  ```

-
  ```
  var IntArray = Java.type(“int[]”)
  var intArr5 = new IntArray(5)
  ```

Educate the Users

- In general, we try to actively educate users to use `Java.type()` and if we can help it don’t even mention stepping stones.
- It’s your choice how much of JVM do you want to expose or hide.
- Hey Dorothy, You’re not in Java anymore.

Linking



```
implementors().stream()
        .filter(i → i.projects().any(::isLanguage))
        .forEach(i → i.hail("
            Hello language implementors!
            Join ~~us next~~ July 29-31, 2013...
"));
```

アヨアマガ at the 6th annual イハアガハイカァナマツ
ガクラキヨニュノ JVM™ Language Summit ナアニズ
ワキメムソ July 29-31, 2013 ダエアガァキイ
ウベニマヨキタァ Oracle, Santa Clara ボズムガサ
νομίζω διόπερ εἰμι
ツメムヨズキヌニヌセ
MH
MH
MH
MH
Java
ORACLE

Dynalink Evolved

- Nashorn embeds Dynalink.
- Dynalink underwent lots of improvements as a result of having an actual client runtime.
- Still available as external standalone project.

Composite Operations

- JavaScript doesn’t have separate namespaces for methods, properties, and collection elements.
- Which one of `dyn:getProp`, `dyn:getElem`, or `dyn:getMethod` do you emit for `obj.foo`?
- Correct answer: all of them!

Composite operations

source operation	operation name
`obj.foo`	`dyn:getProp|getElem|getMethod:foo`
`obj.foo()`	`dyn:getMethod|getProp|getElem:foo`
`obj[x]`	`dyn:getElem|getProp|getMethod`
`obj[x]()`	`dyn:getMethod|getElem|getProp`

Composite Operations

- BeansLinker correctly supports them.
- In most cases, can evaluate the effective operation at link time.
- Except `getElem` on a map followed by `getProp` and/or `getMethod`.

Linking Security

- Dynalink BeansLinker uses publicLookup for cacheable method handles (most of them).
- Completely prevents access to restricted packages (in presence of a security manager!)
- Correctly handles methods marked as `@CallerSensitive`.
  - Method handles are never cached, but unreflected on every link request, with caller's Lookup.

Miscellaneous Dynalink Improvements

- Inner classes are properties of StaticClass
- Detection of frequently relinked call sites; `LinkRequest.isCallSiteUnstable()`.
- `dyn:callMethod` was split into `dyn:getMethod` and `dyn:call`.
  - ‘cause you don’t always call a named function on an object.
- `dyn:new` for invoking constructors.
- Manual overload resolution: `dyn:getMethod:println(char)`.
  - Never really needed; usable by programmer as a performance enhancement. Really introduced for compatibility with Rhino.

Leveraging It From Java

```
private static final MethodHandle REDUCE_CALLBACK_INVOKER =
    Bootstrap.createDynamicInvoker("dyn:call", Object.class,
        Object.class, Undefined.class, Object.class, Object.class,
        long.class, Object.class);

...
private static Object reduceInner(...) {
    ...
    return new IteratorAction<Object>(...) {
        protected boolean forEach(...) {
            result = REDUCE_CALLBACK_INVOKER.invokeExact(
                callbackfn, ScriptRuntime.UNDEFINED, result, val, i,
                    self);
            return true;
        }
    }.apply();
```

Leveraging It From Java

- `createDynamicInvoker` is simply a dynamic invoker on a Nashorn-linked call site.
- `dyn:call` will be able to invoke anything that Nashorn can invoke.

```
public static MethodHandle createDynamicInvoker(
    final String opDesc, final MethodType type) {
        return bootstrap(MethodHandles.publicLookup(), opDesc, type,
            0).dynamicInvoker();
}
```

“InvokeByName” pattern

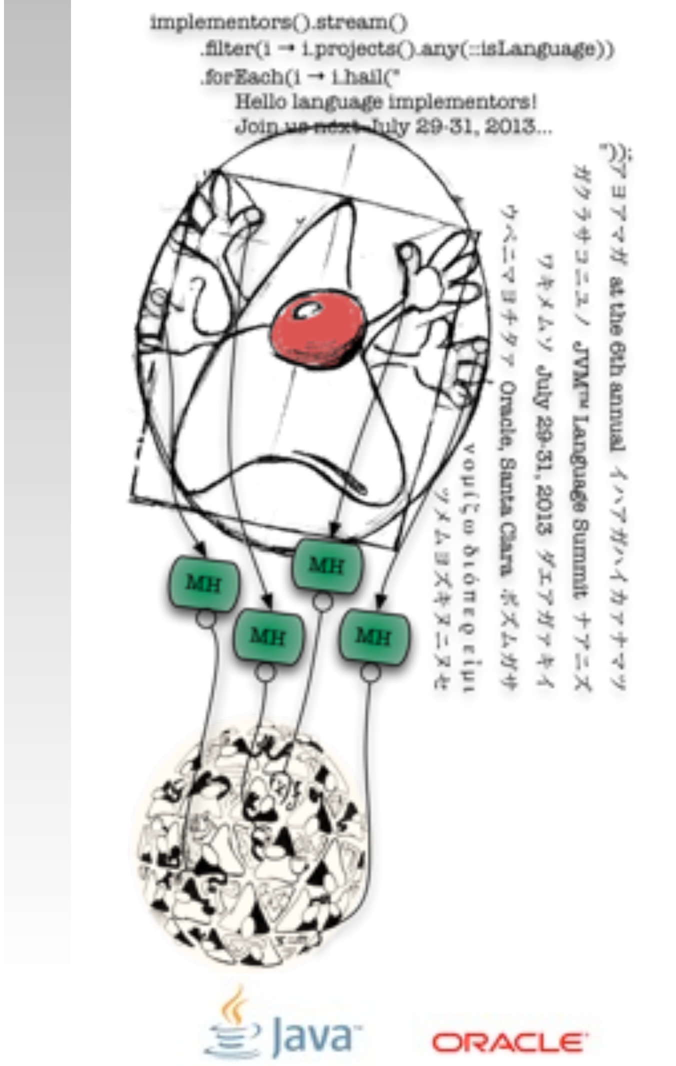
- Java code needs to invoke a function on a JavaScript object.
- e.g. `Array.toString` invokes `this.join()`.

```
class NativeArray {
    ...
    private static final InvokeByName JOIN = new InvokeByName("join",
        ScriptObject.class);
    ...
    public static Object toString(final Object obj) {
        ...
        final ScriptObject sobj = (ScriptObject)obj;
        final Object joinFn = JOIN.getGetter().invokeExact(sobj);
        if (Bootstrap.isCallable(joinFn)) {
            return JOIN.getInvoker().invokeExact(joinFn, sobj);
        }
```

InvokeByName

```
public class InvokeByName {
    private final MethodHandle getter;
    private final MethodHandle invoker;

    public InvokeByName(String name, Class targetClass, Class rtype,
        Class... ptypes) {

        getter  = Bootstrap.createDynamicInvoker(
            "dyn:getMethod|getProp|getElem:" + name, Object.class,
                targetClass);

        final Class[] finalPtypes = ...; // omitted type massaging
        invoker = Bootstrap.createDynamicInvoker("dyn:call", rtype,
            finalPtypes);
    }
```

Security



```
implementors().stream()
        .filter(i → i.projects().any(::isLanguage))
        .forEach(i → i.hail("
            Hello language implementors!
            Join us next July 29-31, 2013...
"));
```

アヨアマガ at the 6th annual イハアガハイカアナマツ
ガクラキヨニユノ JVM™ Language Summit ナアニズ
ワキメムソ July 29-31, 2013 ダエアガアキイ
ウベニマヨチタア Oracle, Santa Clara ポズムガサ
νομίζω διόπερ είμι
ツメムヨズキヌニヌヒ
MH
MH
MH
MH

Security So Far

- Dynalink prevents access to non-public members
- Also to classes in restricted packages.
  - That’s stricter than Java, but a conscious decision.
  - Package restrictions are only in place with SecurityManager.
- Dynalink correctly handles `@CallerSensitive` methods.

Nashorn Additional Security Features

- Nashorn prevents `Java.type()` access to classes in restricted packages.
- Nashorn ties access to reflective classes to a a new `"Nashorn.JavaReflect"` runtime permission.
  - `Class`, `ClassLoader`, everything in `java.lang.reflect` and `java.lang.invoke` packages.

Rationale

- None of the restrictions are in place when there is no security manager.
- Most uses are unaffected even under a security manager.
- You can do less things from JavaScript than from Java
  - Namely, manipulate stuff in restricted packages.
  - Actually, you can: through reflection; if you have the permission.
  - Even then, it won't be pleasant.
- Nashorn runs with `AllPermission` since it lives in `jre/lib/ext`.

Program Agenda

- Extending Java Classes
- Type Conversion Fun
- Arrays
- The Incredible Package Illusion
- Linking
- Security

Java™