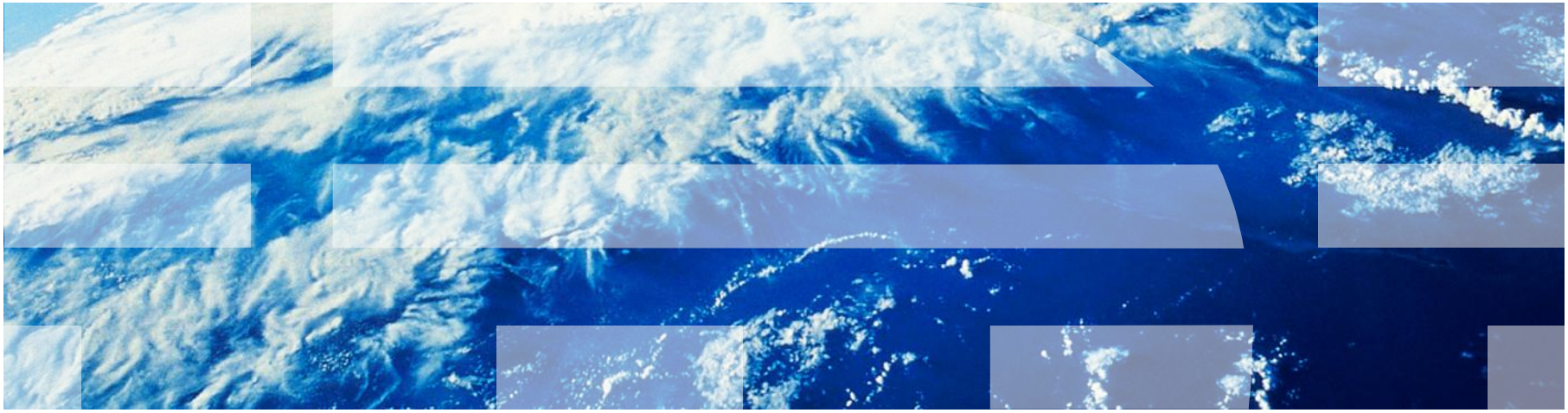
Dan Heidinga – J9 Virtual Machine

30 July 2012

# MethodHandle Introspection: Internals



# Standard disclaimer

IBM’s statements regarding its plans, directions and intent are subject to change or withdrawal at IBM’s sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.

# Agenda

- Why MethodHandle introspection?
- What is MethodHandle introspection?
- Why not full MethodHandle reflection?

# Lambda (JSR 335) implementation options

- JSR 335 investigated two competing implementations for lambdas:
  - Create an inner class for each lambda
  - Create a method for each lambda and use MethodHandles
- Want to choose the best strategy for now and later
  - “Recompile for better performance” is an unwelcome addition to Java
  - A compile time format locks the implementation in
  - Need to defer the choice till runtime
- Invokedynamic allows the runtime to chose the lambda implementation

# Lambda metafactory

- Resulting invokedynamic bootstrap method had this signature

```
static CallSite lambdaMetafactory(
          MethodHandles.Lookup caller,
          String invokedName,
          MethodType invokedType,
          Class<?> samClass,
          String samMethodName,
          MethodType samMethodType,
          MethodHandle handle,
          Class<?> methodClass,
          String methodName,
          MethodType methodType)
```

# Lambda metafactory bootstrap method

- Resulting invokedynamic bootstrap method had this signature

```
static CallSite lambdaMetafactory(
          MethodHandles.Lookup caller,
          String invokedName,
          MethodType invokedType,
          Class<?> samClass,
          String samMethodName,
          MethodType samMethodType,
          MethodHandle handle,
          Class<?> methodClass,
          String methodName,
          MethodType methodType)
```

Stacked by invokedynamic

# Lambda metafactory bootstrap method

- Resulting invokedynamic bootstrap method had this signature

```
static CallSite lambdaMetafactory(
          MethodHandles.Lookup caller,
          String invokedName,
          MethodType invokedType,
          Class<?> samClass,
          String samMethodName,
          MethodType samMethodType,
          MethodHandle handle,
          Class<?> methodClass,
          String methodName,
          MethodType methodType)
```

Stacked by invokedynamic

Describe the SAM target

# Lambda metafactory bootstrap method

- Resulting invokedynamic bootstrap method had this signature

```
static CallSite lambdaMetafactory(
          MethodHandles.Lookup caller,
          String invokedName,
          MethodType invokedType,
          Class<?> samClass,
          String samMethodName,
          MethodType samMethodType,
          MethodHandle handle,
          Class<?> methodClass,
          String methodName,
          MethodType methodType)
```

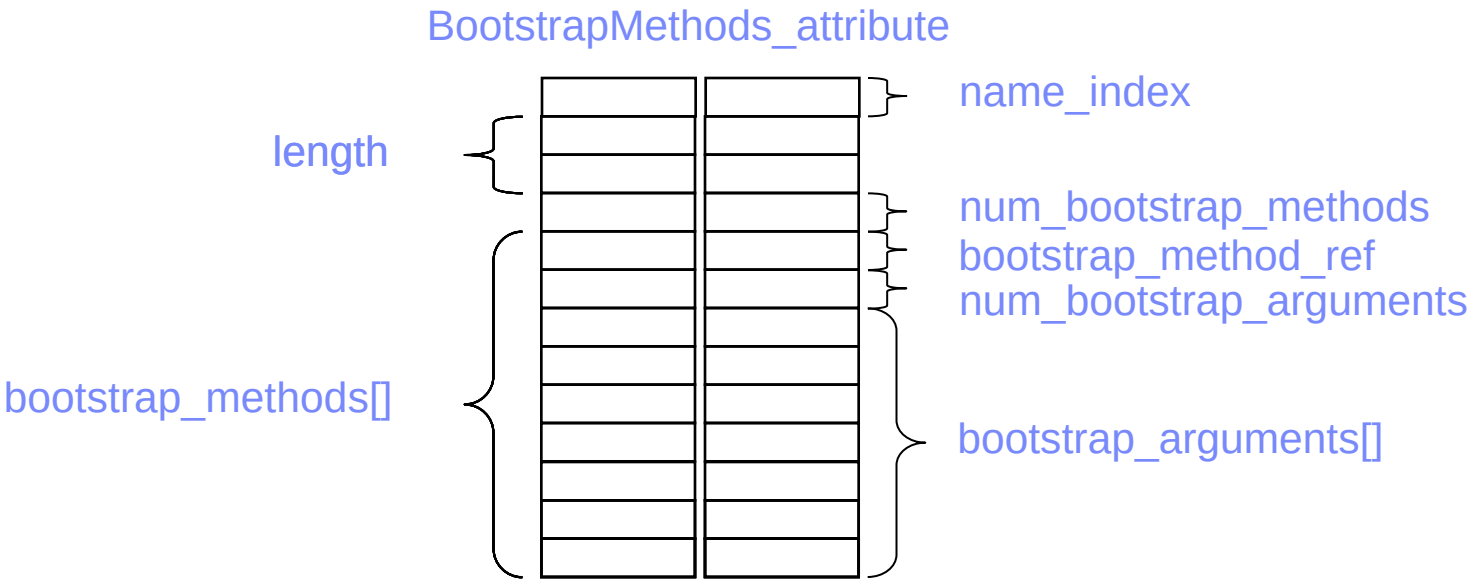
Stacked by invokedynamic

Describe the SAM target

Describe the implementation

# Lambda metafactory bootstrap method

- The bootstrap method argument list must support both inner classes and MethodHandles
  - Complicated signature
  - Redundant information
  - Wasted effort to marshal the unused arguments
  - Increased class file size



# Memory use example: Eclipse plugin directory

- My Eclipse installation has 783 jars in the plugins directory
  - There are 41,392 inner classes in those jars
- Two big assumptions:
  - The number of inner classes is a good approximation to the number of lambdas in large applications
  - The bootstrap arguments refer to constant pool entries that already exist
- 41,392 lambdas X $\frac{\text{14 bytes of arguments}}{\text{lambda}}$ = 566 KB of arguments
- If there were only two arguments, there would be 162 KB of argument data in the class file

# Lambda metafactory bootstrap method

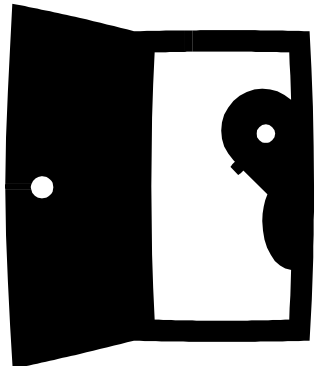
- A simpler bootstrap method would be better:

```
static CallSite lambdaMetafactory(
         MethodHandles.Lookup caller,
         String invokedName,
         MethodType invokedType,
         MethodHandle samMethod,
         MethodHandle implMethod)
```

  - Easier to understand
  - Fewer arguments need to be created and passed to the bootstrap method
  - No potential inconsistencies in the argument list
  - Smaller classfiles
- But this requires us to be able to peek inside the MethodHandles
  - At least for any MethodHandle that can be created by LDC

….But MHs are a black box

# Why are MethodHandles a black box?



"*Method handles are immutable and have no visible state…. Method handles cannot be subclassed by the user. Implementations may (or may not) create internal subclasses of MethodHandle which may be visible via the Object.getClass operation…. The method handle class hierarchy (if any) may change from time to time or across implementations from different vendors*."
  – java.lang.invoke.MethodHandle javadoc

- Why are MethodHandles a black box?
  – Provides maximum flexibility to the implementer
  – Recognizes that MethodHandles are pure behavior
  – Avoids security issues like revealing secrets in bound parameters
  – "Pour in JVM magic"

# Introducing MethodHandleInfo

```
interface MethodHandleInfo {
  public static final int
       REF_getField = 1,
       REF_getStatic = 2,
       REF_putField = 3,
       REF_putStatic = 4,
       REF_invokeVirtual = 5,
       REF_invokeStatic = 6,
       REF_invokeSpecial = 7,
       REF_newInvokeSpecial = 8,
       REF_invokeInterface = 9;

  public int getReferenceKind();
  public Class getDeclaringClass();
  public String getName();
  public MethodType getMethodType();
  public int getModifiers();
}

public class MethodHandles {
  …
  static MethodHandleInfo getInfo(MethodHandle methodHandle) { … }
}
```

# Introspection examples

- MethodHandleInfo is package private interface
  - JSR 292 expert group haven't committed to making it public (yet)
- How do I get a MethodHandleInfo object?  Reflection!
- Class::getDeclaredMethod and Method::setAccessible are required to access introspection

# Introspection examples

- MethodHandleInfo is package private interface
  – JSR 292 expert group haven't committed to making it public (yet)
- How do I get a MethodHandleInfo object? Reflection!
- Class::getDeclaredMethod and Method::setAccessible are required to access introspection

```
Method getInfo = MethodHandles.class.getDeclaredMethod("getInfo",
          new Class[] { MethodHandle.class } );
getInfo.setAccessible(true);
Object info = getInfo.invoke(null, handle);
```

# Creating an Introspector

```
public class Introspector {

static final Method getInfo,
        getName,
        getDeclaringClass,
        getMethodType,
        getModifiers,
        getReferenceKind;

static {
   try {
        getInfo = …

        Class<?> MHInfoClass = getInfo.getReturnType();

        getName = MHInfoClass.getDeclaredMethod(“getName");
        getName.setAccessible(true);
        …

   } catch(Throwable t) { … }
}

public static String getName(Object info) throws Throwable {
        return (String) getName.invoke(info);
}
```

# Roundtripping through the introspector

```
MethodHandle handle = lookup().findStatic(Integer.class, "bitCount",
         methodType(int.class, int.class));

Object info = Inspector.getInfo(handle);

switch(Inspector.getReferenceKind(info)) {

case 5: /* REF_invokeVirtual */
         return lookup().findVirtual(
                  Inspector.getDeclaringClass(info),
                  Inspector.getName(info),
                  Inspector.getMethodType(info).dropParameterTypes(0, 1));

case 6: /* REF_invokeStatic */
         return = lookup().findStatic(
                  Inspector.getDeclaringClass(info),
                  Inspector.getName(info),
                  Inspector.getMethodType(info));
}
```

# Roundtripping through the introspector

```
MethodHandle handle = lookup().findStatic(Integer.class, "bitCount",
         methodType(int.class, int.class));

Object info = Inspector.getInfo(handle);

switch(Inspector.getReferenceKind(info)) {

case 5: /* REF_invokeVirtual */
         return lookup().findVirtual(
                  Inspector.getDeclaringClass(info),
                  Inspector.getName(info),
                  Inspector.getMethodType(info).dropParameterTypes(0, 1));

case 6: /* REF_invokeStatic */
         return = lookup().findStatic(
                  Inspector.getDeclaringClass(info),
                  Inspector.getName(info),
                  Inspector.getMethodType(info));
}
```

# Introspection limitations

- Practical limitations of MethodHandle introspection
  - MethodHandles on private or final methods returned by findVirtual() may appear to have originated in findSpecial()
  - No way to avoid access checks for MethodHandles.Lookup. Roundtripping requires a sufficiently privileged MethodHandles.Lookup object
  - MethodHandles::getInfo returns null for non-LDC MethodHandles

- JSR 292 has a weakness: the inability to “crack” a CONSTANT_MethodHandle
  - Prior to JSR 292, it was always possible to determine the symbolic reference that defined an LDC constant

- Limited introspection restores this ability to “crack” a CONSTANT_MethodHandle
  - Must be MethodHandles defined in LDC instructions, or
  - Created by equivalent MethodHandles.Lookup.find* & unreflect* methods

# Introspection implementation

- MethodHandles::getInfo(MethodHandle) filters out non-public methods in java.lang.invoke
  – Private implementation methods shouldn't be introspectable by accident

- Each MethodHandle subclass that is introspectable implements a private getInfo() method
  – This returns the appropriate info for that MethodHandle subclass

- ConstructorHandle::getInfo():

```
@Override
MethodHandleInfo getInfo() {
        return new MethodHandleInfoImpl(definingClass,
                name, type, modifiers,
                MethodHandleInfo.REF_newInvokeSpecial);
}
```

# Why not full reflection?

- Full MethodHandle reflection is a hard problem
  - Security issues
  - Behavioral equivalence
  - Implementation restrictions

"*Computational physics imposes a basic ground rule for MH reflection. There is (provably) no canonical form for any reasonably general kind of function. Therefore, it is a fool's errand (as we all know!) to attempt to define a cross-platform deterministic (hence normalized) form of reflected method handles*."

– John Rose, "Re: [jsr-292-eg] MH Introspection", 02/21/2012

# Security implications

```
// Lookup a 'String decrypt(String password, byte[] message)' function
  MethodHandle decrypt = MethodHandles.lookup().findStatic(
         MyEncryption.class,
         "decrypt",
         methodType(String.class, String.class, byte[].class));
  return decrypt.bindTo("myPassword");
```

- Full MethodHandle reflection would reveal bound parameters
  - Anyone who had the bound handle now can find my password
- Not an insurmountable problem
  - Security checks
  - "transient" bound parameters
- Expensive in terms of specification, implementation, potential security bugs
  - Burden on every user of 292 to think about the security of their bound parameters

# Behavioral equivalence

- Equivalent MethodHandles should have the same behaviour
  - Return the same result
  - Have the same user noticeable side effects
  - Throw the same exceptions
- Allows swapping MethodHandles without noticing a difference
- None of this requires a canonical representation

```
MethodType type = methodType(int.class);
MethodHandle bind = lookup().bind("Hello", "length", type);
MethodHandle length = lookup().findVirtual(String.class, "length", type);
MethodHandle bindTo = length.bindTo("Hello");
MethodHandle insert = insertArguments(length, 0, "Hello");
```

# Behavioral equivalence

```
MethodHandle target = …
MethodHandle fallback = …
MethodHandle test = constant(boolean.class, true);
return guardWithTest(test, target, fallback);
```

```
SwitchPoint swp = new SwitchPoint();
SwitchPoint.invalidateAll(new SwitchPoint[] {swp});
return swp.guardWithTest(target, fallback);
```

```
// handle has type (int, int)int
MethodType type = methodType(int.class, byte.class, byte.class);
MethodHandle asType = handle.asType(type);
MethodHandle explicitCast = explicitCastArguments(handle, type);
```

# Implementation restrictions

```
String.class.getDeclaredMethods()[3].invoke("hello", …);
```

# Implementation restrictions

```
String.class.getDeclaredMethods()[3].invoke("hello", …);
```

"*... The elements in the array returned are not sorted and are not in any particular order.*"
-- Class::getDeclaredMethods() javadoc

# Conclusion

- JSR 335 (“Project Lambda”) is building on this
  - Limited MethodHandle reflection will be available

- But it won’t be full MH reflection (at least not yet)

# Legal Notices

IBM and the IBM logo are trademarks or registered trademarks of IBM Corporation, in the United States, other countries or both.

Java and all Java-based marks, among others, are trademarks or registered trademarks of Sun Microsystems in the United States, other countries or both.

Other company, product and service names may be trademarks or service marks of others.

THE INFORMATION DISCUSSED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, SUCH INFORMATION. ANY INFORMATION CONCERNING IBM'S PRODUCT PLANS OR STRATEGY IS SUBJECT TO CHANGE BY IBM WITHOUT NOTICE.