Method Handles Everywhere!

Charles Oliver Nutter
@headius

Method Handles

- What are method handles?
- Why do we need them?
- What's new for method handles in Java 9?
- InvokeBinder API
- What other crazy things can we do with them?

Method Handles?

History

- Way back in 2006...the JRuby team joined Sun Microsystems
- Renewed interest in alternative languages on JVM
  - Especially dynamic languages (Ruby, Groovy, Python, ...)
- "Invoke Dynamic" JSR had stalled
- Time for a reboot!

What Did We Need?

- We needed to be able to call methods dynamically
  - With very different class structures, call semantics, etc
- We needed to dynamically assign fields and constants
  - Object shapes determined at runtime
- We needed it to be fast
  - JVM must optimize **as if it were regular Java code**

JSR-292

- InvokeDynamic JSR
- Part 1: a new bytecode "invokedynamic"
  - User-defined bytecode based on method handles
- Part 2: the MethodHandle API (java.lang.invoke)
  - All the plumbing we need to hook up invokedynamic

MethodHandle API

- `MethodHandles.Lookup` provides access to fields, methods
- `MethodType` defines a method signature with params and return
- Methods, fields, and array accesses are "direct" method handles
- `MethodHandles` provides handle wrappers, adapters
  - reorder args, test conditions, ...
- Direct handles plus wrappers form a "method handle tree"

Like Reflection?

- Use cases for reflection are similar
  - Exposing or accessing object state
  - Inspecting and working with unfamiliar classes
  - Metaprogramming
- Method handles do all this but with less overhead and better JIT
- Also starting to replace/expose "Unsafe" operations

MethodHandle Basics

Acquire a MethodHandles.Lookup

```
// This has access to all private fields and methods
// visible from the current class.
MethodHandles.Lookup lookup = MethodHandles.lookup();

// This only has access to public state.
MethodHandles.Lookup publicLookup = MethodHandles.publicLookup();
```

Look up a method with a MethodType

```
MethodType getprop = MethodType.methodType(String.class, String.class);
MethodType listAdd = MethodType.methodType(int.class, Object.class);
MethodType newHash = MethodType.methodType(HashMap.class, int.class, float.class);

MethodHandle getProperty = lookup.findStatic(System.class, "getProperty", getprop);
MethodHandle add = lookup.findVirtual(List.class, "add", listAdd);
MethodHandle hash = lookup.findConstructor(HashMap.class, newHash);
```

Look up a field

```
MethodHandle sysOut =
        lookup.findStaticGetter(System.class, "out", PrintStream.class);
```

```
MethodHandles.Lookup lookup = MethodHandles.lookup();

MethodType getprop = MethodType.methodType(String.class, String.class);
MethodType listAdd = MethodType.methodType(int.class, Object.class);
MethodType newHash = MethodType.methodType(HashMap.class, int.class, float.class);

MethodHandle gp = lookup.findStatic(System.class, "getProperty", getprop);
MethodHandle add = lookup.findVirtual(List.class, "add", listAdd);
MethodHandle hash = lookup.findConstructor(HashMap.class, newHash);

MethodHandle sysOut =
         lookup.findStaticGetter(System.class, "out", PrintStream.class);
```

Wrap handle with other adaptations

```
MethodHandle getHome =
        MethodHandles.insertArguments(getprop, 0, "java.home");

MethodHandle hashFactor1 =
        MethodHandles.insertArguments(hash, 1, 1.0f);
```

Invoke the handle

```
String home = getProperty.invokeWithArguments("java.home");

// use handle already bound to "java.home"
home = getHome.invoke();
```

Invoking a Handle

- Reflection's invoke does a lot for you
  - Type checks and conversion
  - Boxing and unboxing
- MethodHandle.invokeWithArguments: closest to reflection invoke
- MethodHandle.invoke: fewer features, some type conversions
- MethodHandle.invokeExact: raw invoke; types must match exactly

Java as Handles

Hello, Handles!

```
public class Hello {
    public static void main(String[] args) {
        PrintStream stream = System.out;
        String name = args[0];
        stream.println("Hello, " + name);
    }
}
```

Hello, Handles!

```
public class Hello {
    public static void main(String[] args) {
        PrintStream stream = System.out;
        String name = args[0];
        stream.println("Hello, " + name);
    }
}
```

```
MethodHandle streamH =
        lookup.findStaticGetter(System.class, "out", PrintStream.class);
```

Hello, Handles!

```
public class Hello {
    public static void main(String[] args) {
        PrintStream stream = System.out;
        String name = args[0];
        stream.println("Hello, " + name);
    }
}
```

```
MethodHandle nameH =
        MethodHandles.arrayElementGetter(String[].class);
```

Hello, Handles!

```
public class Hello {
    public static void main(String[] args) {
        PrintStream stream = System.out;
        String name = args[0];
        stream.println("Hello, " + name);
    }
}
```

```
MethodHandle printlnH =
        lookup.findVirtual(PrintStream.class, "println",
                           MethodType.methodType(void.class, String.class));
```

Hello, Handles!

```
public class Hello {
    public static void main(String[] args) {
        PrintStream stream = System.out;
        String name = args[0];
        stream.println("Hello, " + name);
    }
}
```

```
MethodHandle concatH =
        lookup.findVirtual(String.class, "concat",
                           MethodType.methodType(String.class, String.class));

concatH = concatH.bindTo("Hello, ");
```

Combining Multiple Handles

- Many handle adapters can wrap two or more other handles
  - Method handle "combinators"
- These combinators allow more complex adaptations
- JIT still sees through and optimizes to native code

if/then/else

```
private static final HashMap cache = new HashMap();

if (cache.containsKey(cacheKey)) {
    return cache.get(cacheKey);
} else {
    Object data = db.loadRow(cacheKey);
    cache.put(cacheKey, data);
    return data;
}

miniCache =
        MethodHandles.guardWithTest(cond, then, els);
```

```
        miniCache =
                MethodHandles.guardWithTest(cond, then, els);

MethodHandle cond =
        lookup.findVirtual(Map.class, "containsKey",
                           methodType(boolean.class, Object.class));
cond = cond.bindTo(cache);

MethodHandle then =
        lookup.findVirtual(Map.class, "get",
                           methodType(Object.class, Object.class));
then = then.bindTo(cache);

MethodHandle els =
        lookup.findStatic(MiniCache.class, "cacheFromDB",
                          methodType(Object.class, Map.class, String.class));
els = els.bindTo(cache)

public static Object cacheFromDB(Map cache, String key) {cache.put(...)}
```

try/catch

```
MethodHandle tryCatch =
        MethodHandles.catchException(body, handler);
```

```
MethodHandle body = cacheFromDB; // protect our DB access
```

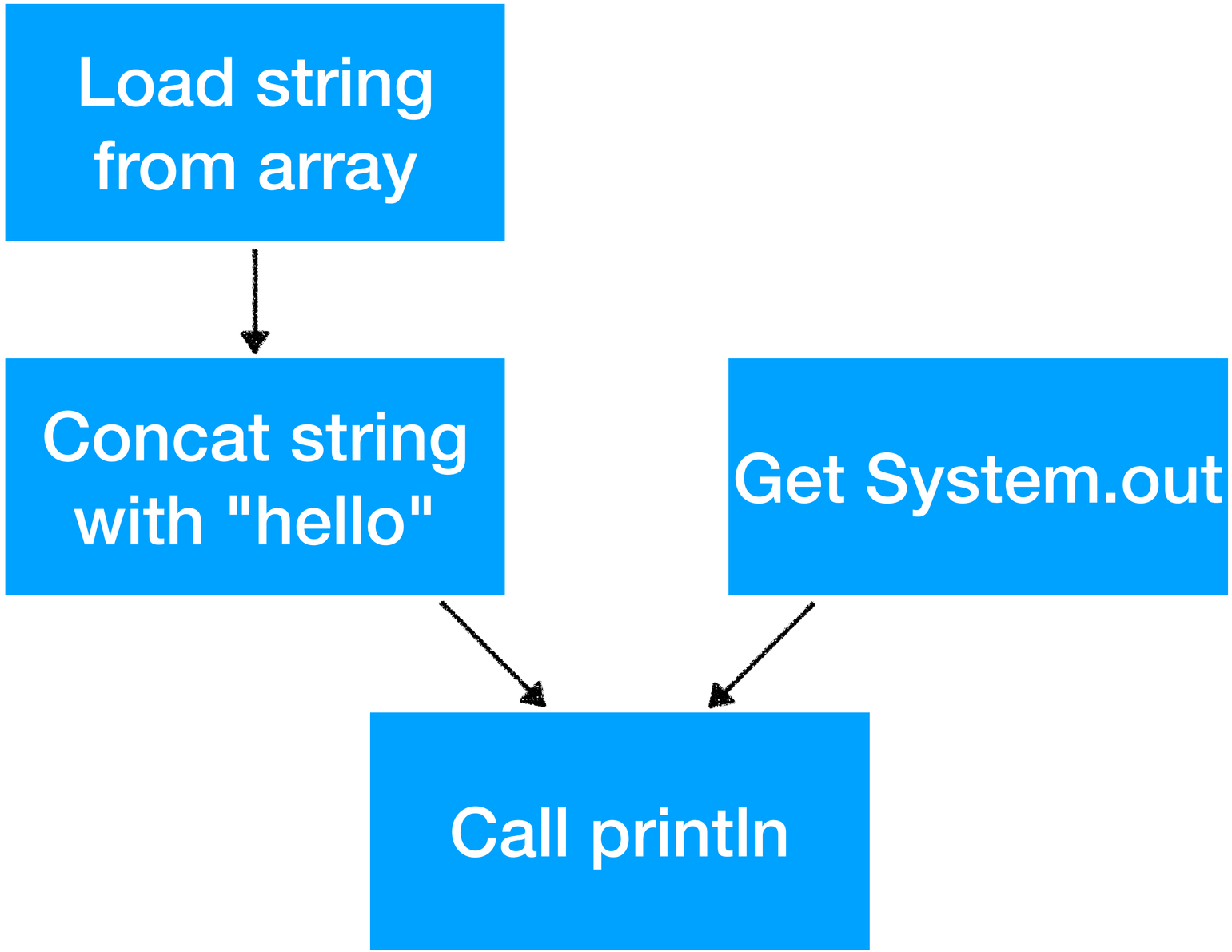
```
MethodHandle handler =
        lookup.findStatic(MiniCache.class, "cacheDefault",
                          methodType(Object.class, Map.class, String.class));
```

```
// Provide a default value when there's an error
public static Object cacheDefault(Throwable err, Map cache, String key) {
    String value = "error for key '" + key + "':" + err.getMessage();
    cache.put(key, value);
    return value;
}
```

More Adaptations

- insert/dropArguments - insert constants, drop unneeded args
- permuteArguments - reorder args
- filterArguments/Return - pass value to filter, replace with result
- foldArguments - pass all arguments to function, prepend resulting value

All Together Now


Load string from array
Concat string with "hello"
Get System.out
Call println

```
MethodHandle streamH =
        lookup.findStaticGetter(System.class, "out", PrintStream.class);

MethodHandle nameH =
        MethodHandles.arrayElementGetter(String[].class);
nameH = MethodHandles.insertArguments(nameH, 1, 0);

MethodHandle concatH =
        lookup.findVirtual(String.class, "concat",
                           MethodType.methodType(String.class, String.class));
concatH = concatH.bindTo("Hello, ");

MethodHandle printlnH =
        lookup.findVirtual(PrintStream.class, "println",
                           MethodType.methodType(void.class, String.class));
printlnH =
        MethodHandles.foldArguments(printlnH,
                                    MethodHandles.dropArguments(streamH, 0, String.class));

MethodHandle helloH = MethodHandles.filterArguments(printlnH, 0, concatH);
helloH = MethodHandles.filterArguments(helloH, 0, nameH);

helloH.invoke(args);
```

Java 9

Missing Features

- try/finally
- various loop forms
- volatile and atomic field/array accesses
- array construction

try/finally

- Like javac, finally block must be duplicated
  - Normal path saves return value, calls finally on the way out
  - Exceptional path calls finally, re-raises exception
- In Java 7 or 8 handles, have to do this duplication manually

```
public Object tryFinally(MethodHandle target, MethodHandle post) throws Throwable {
    try {
        return target.invoke();
    } finally {
        post.invoke();
    }
}
```

```
MethodHandle exceptionHandler = Binder
        .from(target.type().insertParameterTypes(0, Throwable.class).changeReturnType(void.class))
        .drop(0)
        .invoke(post);

MethodHandle rethrow = Binder
        .from(target.type().insertParameterTypes(0, Throwable.class))
        .fold(exceptionHandler)
        .drop(1, target.type().parameterCount())
        .throwException();

target = MethodHandles.catchException(target, Throwable.class, rethrow);

// if target returns a value, we must return it regardless of post
MethodHandle realPost = post;
if (target.type().returnType() != void.class) {
    // modify post to ignore return value
    MethodHandle newPost = Binder
            .from(target.type().insertParameterTypes(0, target.type().returnType()).changeReturnType(void.class))
            .drop(0)
            .invoke(post);

    // fold post into an identity chain that only returns the value
    realPost = Binder
            .from(target.type().insertParameterTypes(0, target.type().returnType()))
            .fold(newPost)
            .drop(1, target.type().parameterCount())
            .identity();
}

return MethodHandles.foldArguments(realPost, target);
```

Ouch!

Java 9

```
MethodHandle logger =
        MethodHandles.tryFinally(cacheFromDB, logCacheUpdate);
```

Loops

Java 9

```
MethodHandle whileLoop =
        MethodHandles.whileLoop(init, cond, body);
```

```
MethodHandle doWhileLoop =
        MethodHandles.doWhileLoop(init, cond, body);
```

```
MethodHandle countedLoop =
        MethodHandles.countedLoop(count, init, body);
```

```
MethodHandle iteratedLoop =
        MethodHandles.iteratedLoop(iterator, init, body);
```

```
MethodHandle complexLoop =
        MethodHandles.loop(...)
```

Java 9

VarHandles

VarHandle

Java 9

- Utilities for accessing fields and arrays
- Volatile and atomic accesses
- byte array/buffer "views"
  - Treat a byte[] like int[] or long[]
- Convertible to a MethodHandle

Java 9

Acquire a VarHandle

```
sysOut = lookup.findStaticVarHandle(System.class, "out", PrintStream.class);
otherField = lookup.findVarHandle(...);

byteView = MethodHandles.byteArrayViewVarHandle(int[].class, ByteOrder.BIG_ENDIAN);
bbView = MethodHandles.byteBufferViewVarHandle(long[].class, ByteOrder.BIG_ENDIAN);

strArray = MethodHandles.arrayElementVarHandle(String[].class);
```

Java 9

Atomic and Volatile Accesses

```
String[] names = loadNames();

boolean updated = strArray.compareAndSet(names, 5, "Charles", "Chris");

strArray.setVolatile(names, 5, "Chris");
```

Java 9

View bytes as wide values

```
byte[] data = io.read();

long count = data.length / 4;

for (int i = 0; i < count; i++) {
    long wideValue = vh.get(data, i);

    processValue(wideValue);
}
```

Java 9

Mix VarHandle into MethodHandle tree

```
VarHandle logField =
    lookup.findStaticVarHandle(MyLogger.class, "logEnabled", boolean.class);

MethodHandle logCheckVolatile =
        logCheck.toMethodHandle(VarHandle.AccessMode.GET_VOLATILE);

MethodHandle conditionalLogger =
        MethodHandles.guardWithTest(logCheckVolatile, doLog, dontLog);
```

InvokeBinder

```
MethodHandle streamH =
        lookup.findStaticGetter(System.class, "out", PrintStream.class);

MethodHandle nameH =
        MethodHandles.arrayElementGetter(String[].class);
nameH = MethodHandles.insertArguments(nameH, 1, 0);

MethodHandle concatH =
        lookup.findVirtual(String.class, "concat",
                           MethodType.methodType(String.class, String.class));
concatH = concatH.bindTo("Hello, ");

MethodHandle printlnH =
        lookup.findVirtual(PrintStream.class, "println",
                           MethodType.methodType(void.class, String.class));
printlnH =
        MethodHandles.foldArguments(printlnH,
                                    MethodHandles.dropArguments(streamH, 0, String.class));

MethodHandle helloH = MethodHandles.filterArguments(printlnH, 0, concatH);
helloH = MethodHandles.filterArguments(helloH, 0, nameH);

helloH.invoke(args);
```

InvokeBinder

- Method handles get composed in reverse
  - Start with method, wrap with if/then, try/catch, permute args, etc
  - We write code going forward
- Repetitive: same argument/method types over and over
- InvokeBinder helps compose complex handle trees naturally
  - Start with incoming args, adapt in **forward** direction

```
MethodHandle streamH =
        lookup.findStaticGetter(System.class, "out", PrintStream.class);

MethodHandle nameH =
        MethodHandles.arrayElementGetter(String[].class);
nameH = MethodHandles.insertArguments(nameH, 1, 0);

MethodHandle concatH =
        lookup.findVirtual(String.class, "concat",
                           MethodType.methodType(String.class, String.class));
concatH = concatH.bindTo("Hello, ");

MethodHandle printlnH =
        lookup.findVirtual(PrintStream.class, "println",
                           MethodType.methodType(void.class, String.class));
printlnH =
        MethodHandles.foldArguments(printlnH,
                                    MethodHandles.dropArguments(streamH, 0, String.class));

MethodHandle helloH = MethodHandles.filterArguments(printlnH, 0, concatH);
helloH = MethodHandles.filterArguments(helloH, 0, nameH);

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

Basic Use

- Binder.from(type) produces a new Binder (which is immutable)
- Binder#drop, insert, permute, etc act in call-forward direction
- Binder#fold, filter, catchException, etc take additional handles
- Endpoints: invoke*, set/getField, arraySet/Get, constant, identity, nop, branch (guardWithTest)
- Utilities for larger constructs

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                      .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                      .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                      .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                      .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                      .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                      .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

```
MethodHandle helloH = Binder
        .from(void.class, String[].class)
        .filter(0, String.class, b -> b.append(0)
                                       .arrayGet())
        .filter(0, String.class, b -> b.prepend("Hello, ")
                                       .invokeVirtualQuiet(lookup, "concat"))
        .fold(PrintStream.class, b -> b.dropAll()
                                       .getStaticQuiet(lookup, System.class, "out"))
        .invokeVirtualQuiet(lookup, "println");

helloH.invoke(args);
```

Something Crazy?

Turing Complete?

- We can modify state with field and array accessors
- We can call methods
- We can insert conditional logic
- We can execute controlled loops
- We can represent a language entirely with method handles!

Ruby "Compiler"

- Visit our way through JRuby's abstract syntax tree
- Each syntactic element translated into a method handle
- Local variable state stored in an Object[]
- Simple numeric operations, conditions, loops

```java
public static void main(String[] args) throws Throwable {
    Ruby runtime = Ruby.newInstance();
    String src = args[0];
    int loopCount = 1;

    if (args[0].equals("--loop")) {
        src = args[2];
        loopCount = Integer.parseInt(args[1]);
    }

    Node top = (Node) runtime.parseFromMain("main", new ReaderInputStream(new StringReader(src)));

    System.out.println("AST:\n" + top);

    HandleCompiler hc = new HandleCompiler(runtime);

    MethodHandle handle = hc.compile(top);

    Object result = handle.invoke();

    for (int i = 0; i < loopCount; i++) {
        handle.invoke();
    }

    System.out.println("result: " + result);
}
```

```
MethodHandle compile(Node node) {
    return node.accept(this);
}
```

```
public <T> T accept(NodeVisitor<T> iVisitor) {
    return iVisitor.visitCallNode(this);
}
```

```
public MethodHandle visitFalseNode(FalseNode iVisited) {
    return Binder.from(Object.class, Object[].class)
            .drop(0)
            .constant(Boolean.FALSE);
}

public MethodHandle visitTrueNode(TrueNode iVisited) {
    return Binder.from(Object.class, Object[].class)
            .drop(0)
            .constant(Boolean.TRUE);
}
```

```
public MethodHandle visitFixnumNode(FixnumNode iVisited) {
    return Binder.from(Object.class, Object[].class)
            .drop(0)
            .constant(Long.valueOf(iVisited.getValue()));
}
```

```java
@Override
public MethodHandle visitLocalVarNode(LocalVarNode iVisited) {
    return Binder.from(Object.class, Object[].class)
            .append(iVisited.getIndex())
            .arrayGet();
}

@Override
public MethodHandle visitLocalAsgnNode(LocalAsgnNode iVisited) {
    return Binder.from(Object.class, Object[].class)
            .fold(compile(iVisited.getValueNode()))
            .append(iVisited.getIndex())
            .permute(1, 2, 0)
            .foldVoid(b->b.arraySet())
            .drop(0, 2)
            .identity();
}
```

```
public MethodHandle visitIfNode(IfNode iVisited) {
    MethodHandle cond = compile(iVisited.getCondition());
    MethodHandle then = compile(iVisited.getThenBody());
    MethodHandle els = compile(iVisited.getElseBody());

    cond = MethodHandles.filterReturnValue(cond, TRUTHY);

    return MethodHandles.guardWithTest(cond, then, els);
}
```

```
public MethodHandle visitWhileNode(WhileNode iVisited) {
    MethodHandle pred = compile(iVisited.getConditionNode());
    MethodHandle body = compile(iVisited.getBodyNode());
    boolean atStart = iVisited.evaluateAtStart();

    pred = Binder.from(boolean.class, Object.class, Object[].class)
            .drop(0)
            .invoke(MethodHandles.filterReturnValue(pred, TRUTHY));

    body = Binder.from(Object.class, Object.class, Object[].class)
            .drop(0)
            .invoke(body);

    if (atStart) {
        return MethodHandles.whileLoop(null, pred, body);
    } else {
        return MethodHandles.doWhileLoop(null, pred, body);
    }
}
```

```
public MethodHandle visitRootNode(RootNode iVisited) {
    // compile body of the script
    MethodHandle child = compile(iVisited.getBodyNode());

    // create an array to hold our variables
    StaticScope scope = iVisited.getStaticScope();
    int varCount = scope.getNumberOfVariables();

    MethodHandle combiner = Binder.from(Object[].class)
            .append(varCount)
            .arrayConstruct();

    return MethodHandles.foldArguments(child, combiner);
}
```

Does It Work?

```
$ java \
       -cp target/jruby-handle-compiler-1.0-SNAPSHOT.jar
       com.headius.jruby.HandleCompiler
       "a = 1; while a < 1_000_000; a += 1; end; a"
AST:
(RootNode 0, (BlockNode 0, (LocalAsgnNode:a 0, (FixnumNode 0)),
(WhileNode 0, (CallNode:< 0, (LocalVarNode:a 0), (ArrayNode 0,
(FixnumNode 0)), null), (LocalAsgnNode:a 0, (CallNode:+ 0,
(LocalVarNode:a 0), (ArrayNode 0, (FixnumNode 0)), null))),
(LocalVarNode:a 0)))
result: 1000000
```

Does It Work Well?

```
Compiled method (c1)   18446 2640       2       java.lang.invoke.LambdaForm$MH/1430710100::identity_L (89 bytes)
 total in heap  [0x0000000114fad090,0x0000000114fae118] = 4232
 relocation     [0x0000000114fad200,0x0000000114fad2b8] = 184
 main code      [0x0000000114fad2c0,0x0000000114fad800] = 1344
 stub code      [0x0000000114fad800,0x0000000114fad8e8] = 232
 oops           [0x0000000114fad8e8,0x0000000114fad960] = 120
 metadata       [0x0000000114fad960,0x0000000114fad9d8] = 120
 scopes data    [0x0000000114fad9d8,0x0000000114fadd80] = 936
 scopes pcs     [0x0000000114fadd80,0x0000000114fae090] = 784
 dependencies   [0x0000000114fae090,0x0000000114fae098] = 8
 nul chk table  [0x0000000114fae098,0x0000000114fae118] = 128
--------------------------------------------------------------------------------
java/lang/invoke/LambdaForm$MH.identity_L(Ljava/lang/Object;)V  [0x0000000114fad2c0, 0x0000000114fad8e8]  1576 bytes
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} {0x00000001327846e0} 'identity_L' '(Ljava/lang/Object;)V' in 'java/lang/invoke/LambdaForm$MH'
  # parm0:    rsi:rsi   = 'java/lang/Object'
  #           [sp+0xe0]  (sp of caller)
  0x0000000114fad2c0: mov    %eax,-0x14000(%rsp)
  0x0000000114fad2c7: push   %rbp
  0x0000000114fad2c8: sub    $0xd0,%rsp
  0x0000000114fad2cf: movabs $0x1316f89f0,%rdx
  0x0000000114fad2d9: mov    0x10(%rdx),%edi
  0x0000000114fad2dc: add    $0x8,%edi
  0x0000000114fad2df: mov    %edi,0x10(%rdx)
  0x0000000114fad2e2: and    $0x3ff8,%edi
  0x0000000114fad2e8: cmp    $0x0,%edi
  0x0000000114fad2eb: je     0x0000000114fad611  ;*ldc {reexecute=0 rethrow=0 return_oop=0}
                                                 ; - java.lang.invoke.LambdaForm$MH/1430710100::identity_L@0
  0x0000000114fad2f1: movabs $0x6c0596740,%rsi  ;   {oop('java/lang/Class'{0x00000006c0596740} 'java/lang/Object')}
  0x0000000114fad2fb: mov    $0x3,%edx          ;*invokestatic newArray {reexecute=0 rethrow=0 return_oop=0}
                                                ; - java.lang.reflect.Array::newInstance@2 (line 78)
                                                ; - java.lang.invoke.DirectMethodHandle$Holder::invokeStatic@11
                                                ; - java.lang.invoke.LambdaForm$BMH/266906347::reinvoke@26
                                                ; - java.lang.invoke.LambdaForm$MH/1430710100::identity_L@20

  0x0000000114fad300: nopl   0x0(%rax)
  0x0000000114fad307: callq  0x000000011c54e4d8  ; ImmutableOopMap{}
                                                ;*invokestatic newArray {reexecute=0 rethrow=0 return_oop=0}
                                                ; - java.lang.reflect.Array::newInstance@2 (line 78)
                                                ; - java.lang.invoke.DirectMethodHandle$Holder::invokeStatic@11
                                                ; - java.lang.invoke.LambdaForm$BMH/266906347::reinvoke@26
                                                ; - java.lang.invoke.LambdaForm$MH/1430710100::identity_L@20
                                                ;   {static_call}
  0x0000000114fad30c: cmp    $0x0,%rax
  0x0000000114fad310: je     0x0000000114fad358  ;*ifnull {reexecute=0 rethrow=0 return_oop=0}
                                                ; - java.lang.Class::cast@1 (line 3577)
                                                ; - java.lang.invoke.DirectMethodHandle$Holder::invokeSpecial@11
                                                ; - java.lang.invoke.LambdaForm$BMH/1668627309::reinvoke@21
                                                ; - java.lang.invoke.LambdaForm$BMH/266906347::reinvoke@44
                                                ; - java.lang.invoke.LambdaForm$MH/1430710100::identity_L@20
```

We have made a native compiler without generating any bytecode

Results

- "Compiled" code really does optimize to native
- It takes a little while to get there
  - Method handles optimize internally first
  - JIT picks them up later
- Maybe just a toy, but an interesting toy!

Less Crazy?

Better Streams

- Streams API (Java 8) has some issues
  - Most JVMs won't inline everything
  - Same call paths used for all lambdas, types of collections
- Method handles can inline and specialize better
  - Reimplement streams as handles!
  - Coming soon!

```
static final HandleStream MAP_COLLECT_FILTER = StreamBinder.from(Set.class, List.class)
                .map((x) -> process(x))
                .filter((x) -> filter(x));

public static void blah() {
    Set<String> x = MAP_COLLECT_FILTER.collect(myList, HandleCollectors::toSet());
}
```

Wrapping Up

- Method handles are powerful
- InvokeBinder makes them easier to use
- Many new use cases to be explored
- Play with them and see what you can do!

Thank You!

- Charles Oliver Nutter
- headius@headius.com
- @headius

- https://github.com/headius/invokebinder