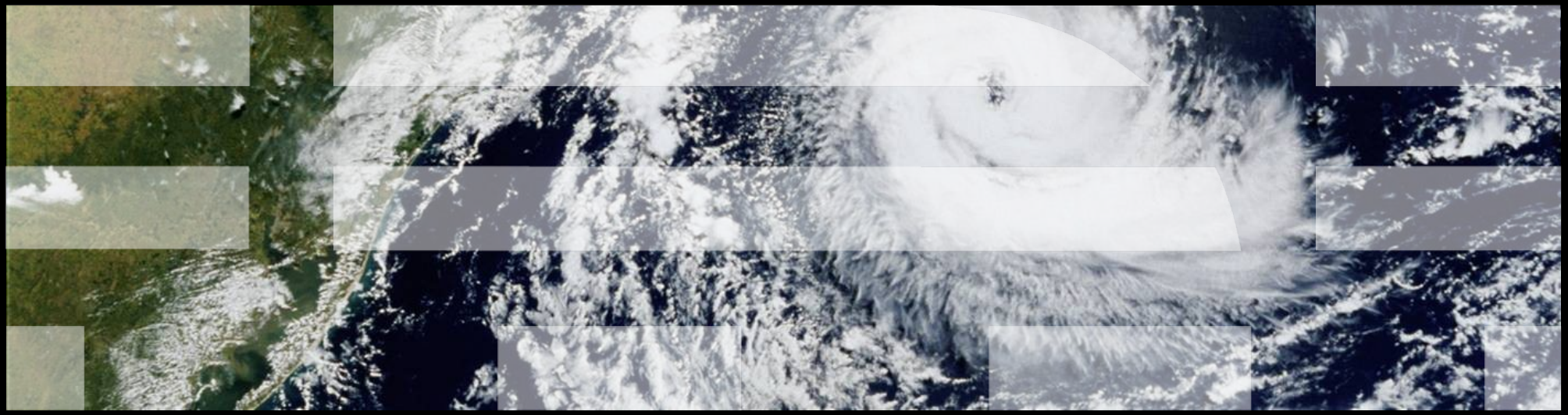
# MethodHandle implemention tips and tricks

Dan Heidinga
J9 VM Software Developer
daniel_heidinga@ca.ibm.com



# MethodHandles: a 30 sec introduction

"A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution."

-- java.lang.invoke.MethodHandle javadoc

# MethodHandles: a 30 sec introduction

"A method handle is a **typed**, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution."

-- java.lang.invoke.MethodHandle javadoc

- **typed** – strongly typed function pointer. Type is carried by an instance of MethodType.

# MethodHandles: a 30 sec introduction

"A method handle is a typed, **directly executable** reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution."

-- java.lang.invoke.MethodHandle javadoc

- typed – strongly typed function pointer. Type is carried by an instance of MethodType.
- **directly executable** – 3 execution modes:
  - invokeExact – invoking signature must exactly match the handle's type
  - invoke – (formally invokeGeneric) – is equivalent to handle.asType(invokingSignature).invokeExact()
  - invokeWithArguments – allows "reflection-style" invocation using arguments collected into arrays

# MethodHandles: a 30 sec introduction

"A method handle is a typed, directly executable reference to an underlying **method, constructor, field, or similar low-level operation**, with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution."

-- java.lang.invoke.MethodHandle javadoc

- typed – strongly typed function pointer. Type is carried by an instance of MethodType.
- directly executable – 3 execution modes:
  - invokeExact – invoking signature must exactly match the handle's type
  - invoke – (formally invokeGeneric) – is equivalent to handle.asType(invokingSignature).invokeExact()
  - invokeWithArguments – allows "reflection-style" invocation using arguments collected into arrays
- **method, constructor, field, or similar low-level operation** – emulates existing java bytecode set: invoke*, put/getField, put/getStatic

# MethodHandles: a 30 sec introduction

"A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. These **transformations** are quite general, and include such patterns as conversion, insertion, deletion, and substitution."

-- java.lang.invoke.MethodHandle javadoc

- typed – strongly typed function pointer. Type is carried by an instance of MethodType.
- directly executable – 3 execution modes:
  - invokeExact – invoking signature must exactly match the handle's type
  - invoke – (formally invokeGeneric) – is equivalent to handle.asType(invokingSignature).invokeExact()
  - invokeWithArguments – allows "reflection-style" invocation using arguments collected into arrays
- method, constructor, field, or similar low-level operation – emulates existing java bytecode set: invoke*, put/getField, put/getStatic
- **transformations** - argument insertion and deletion, argument and return filtering, catching and throwing exceptions, collecting and spreading, permuting, casting, decision trees, etc.

# Talk Purpose

- Let you peek behind the curtain and see how MethodHandles have been implemented
  - It's a major change to the JVM
- Provide you with a solid initial performance model for MethodHandle related code
  - Guidance on what is fast
  - How to write your own fast code

# Design of a JSR 292 implementation

- How do you add something like this to the JVM?
  - Implement it in Java as much as possible and let the JIT do what it does best
  - Find the smallest set of things that need to be done primitively
  - Use these as building blocks to make the rest of the system work
  - Rebalance the line between primitive and Java as required

- Add interpreter support to determine if the feature is feasible, useful, etc.
  - Usually easy to prototype by hacking the interpreter
  - Sufficient to validate if the design will work and to find corner cases

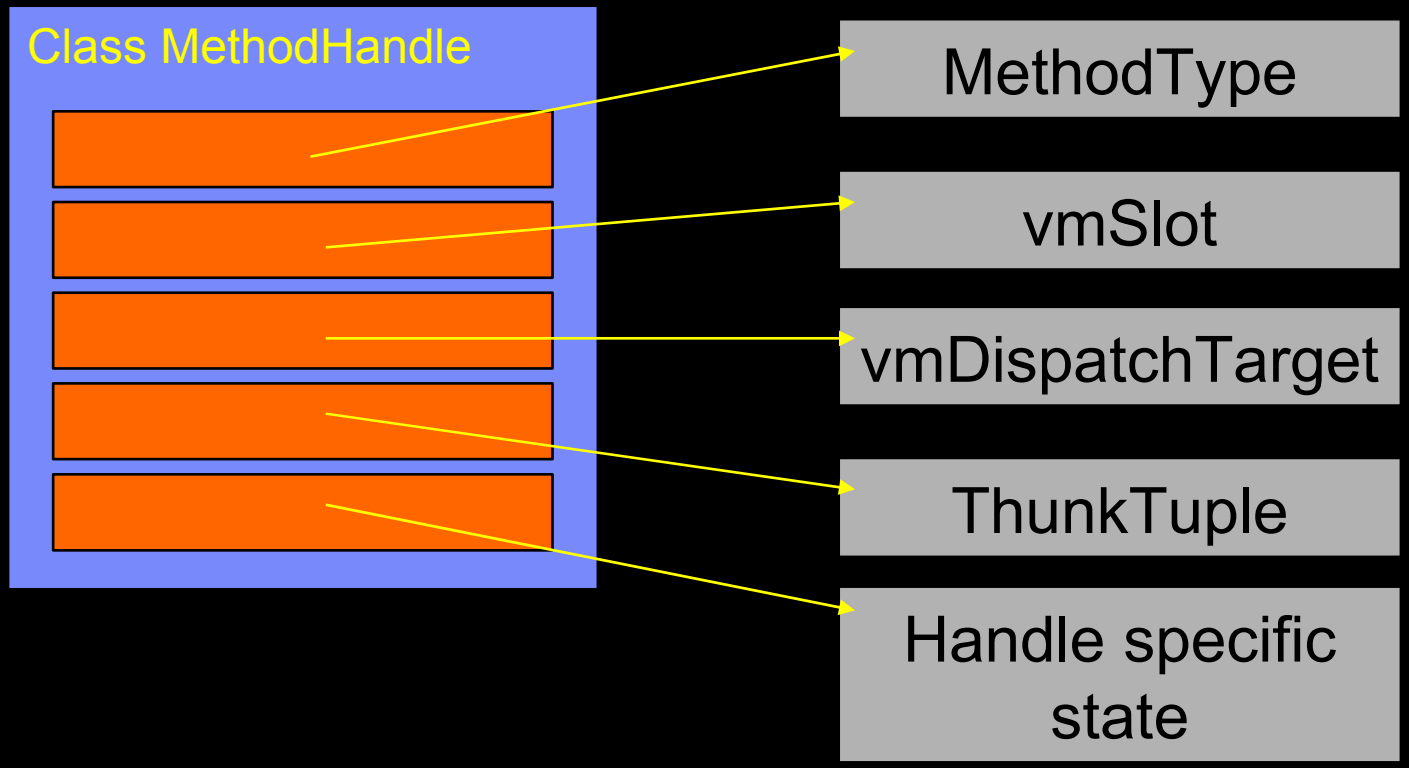
- Add JIT support to make it fast
  - JIT design is informed by the lessons learned doing the interpreter support
  - Modifies the interpreter support as required to aid the JIT design

# Primitive vs. “Java” MethodHandles

Primitive handles require VM support
- Invoking methods
- Getting and setting fields
- Constructors
- Collecting arguments
- Type conversion
- Return filtering
- Bound parameters
- DynamicInvokers
- Special MethodHandle.{invokeExact, invoke} handles

- Everything else can be built on top of this:
  - Primitive handles on helper methods. Mostly API in the MethodHandles factory class:
    arrayElement{Getter,Setter}, identity, constant, throwException
  - Primitive handles bound to helper classes.
    - These helper classes hold the necessary state for the handle's operation

# Primitive MethodHandle layout



# Primitive MethodHandle interpreter implementation

- **MH.type**: The strongly typed signature descriptor for this MethodHandle.
  - Describes the arguments for this handle instance
- **MH.vmSlot**: Stores vm-specific references used by the dispatchTarget
  - Virtual function table indexes
  - Field offsets
  - Method addresses
- **MH.vmDispatchTarget**: interpreter entry point for the handle
  - Performs the necessary stack operations for this handle
- Primitive handles avoid building frames in the interpreter where possible
  - MethodHandle.{invoke, invokeExact} don't get frames
  - Modifes the stack and jumps to the next primitive handle in the chain
  - Frames for eventual target methods do get created

# Primitive MethodHandle JIT implementation

- **MH.thunkTuple**: a shared object that holds the JIT entrypoint for a handle instance
  - Shared between multiple handles when possible to conserve code cache
  - Without sharing, 5 million handle stress test --> code cache filled with duplicate code

- ThunkArchetypes
  - Templates implemented in Java that describe the operation of a handle subclass
  - JIT specializes the template for the MethodType
  - Specialized thunkArchetype JIT entry point is stored in the ThunkTuple
  - Jump from archetype to archetype as appropriate
- ThunkTable: holds the thunkTuples and manages the sharing
  - Sharing is done on a per-MethodHandle subclass basis
  - Criteria for sharing is very different between different handle subclasses
    - Signature only (e.g. FieldHandle)
    - Signature + data (e.g. AsTypeHandle)

- Don't affect Security-related stack walks and can't be seen in stack traces

# FilterReturnHandle and continuation based JVMs

- J9's interpreter is continuation based
  - Each bytecode dispatches directly to the next one
  - Requires a very different mindset than “call-return” semantics

- MethodHandles.filterReturnValue() creates a handle that modifies the return value
  - Need to get the value returned by the previous handle invocation
  - Usual technique: call back into the interpreter
    - Expensive!  Interpreter state needs to be preserved
    - FilterReturn will be very common

# FilterReturnHandle and continuation based JVMs

- Solution: placeholder frames
  - Insert an “extra” frame on the stack
    - Arguments are described for the GC using a normal Java method
    - The bytecodes for the frame are NOT from the method
    - impdep1: a reserved bytecode provides the “next continuation”

- impdep1
  - Remove the placeholder frame
  - Redispatch as required

```
Exception in thread "main" java.lang.RuntimeException
        at Example.<init>(Example.java:8)
        at MethodHandle.constructorPlaceHolder()
        at MethodHandle.returnFilterPlaceHolder()
        at Example.main(Example.java:14)
```

# FilterReturnHandle JIT implementation

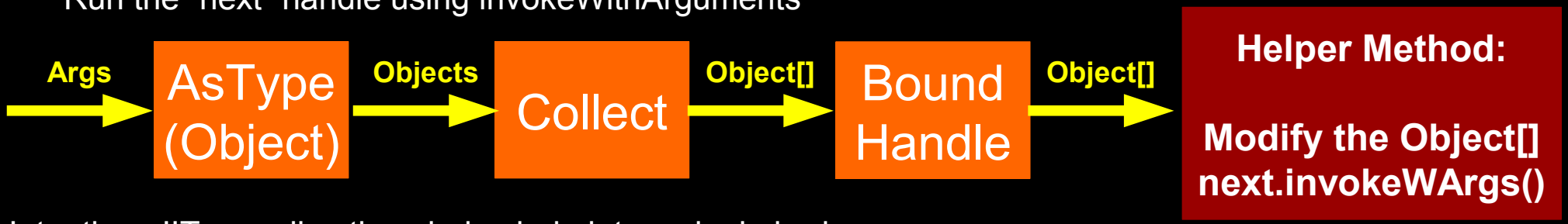
- This is easy for the JIT
  - No placeholder frames required
  - It's just regular Java code
- None of the natives in the thunkArchetype have real implementations
  - Can't be invoked by the interpreter
  - Recognized by the JIT and converted to the appropriate code

```
private final int
invokeExact_thunkArchetype_I(int argPlaceholder) throws Throwable
{
    return ILGenMacros.invokeExact_I(
            filter,
            fixReturnType(
                    ILGenMacros.invokeExact_I(
                            next,
                            argPlaceholder)
                    )
            );
}
```

# “Java” MethodHandles

- Primarily to implement the complicated methods in the MethodHandles factory
  - Argument modification: spread, permute, drop, insert, filter, & fold
  - CatchException & guardWithTest

- Employ a simple pattern:
  - Convert the arguments to Object
  - Collect the Objects into an array
  - Modify the array as required
  - Run the “next” handle using invokeWithArguments




- Intention: JIT compiles the whole chain into a single body
  - Without boxing/unboxing
  - Without collecting
  - Minimal conversions

- All the standard optimizations will kick in

# Custom LoopHandle example

```
class LoopHandle {
    final MethodHandle loopStarter;

    LoopHandle(MethodHandle starter) {
        loopStarter = starter;
    }

    final Object helper(Object[] args) throws Throwable {
        MethodHandle target = (MethodHandle) loopStarter.invokeWithArguments(args);
        while(target != null) {
            target = (MethodHandle) target.invokeWithArguments(args);
        }
        return null;
    }

    public static MethodHandle createLoopHandle(MethodHandle starter,
            MethodType loopType) throws Throwable {

        MethodType helperType = MethodType.methodType(Object.class, Object[].class);
        return lookup()
                .bind(new LoopHandle(starter), "helper", helperType)
                .asCollector(Object[].class, loopType.parameterCount())
                .asType(loopType);
    }
}
```

# LoopHandle example

```
public class Summit {
    int i = 0;
    MethodHandle next;

    public static void main(String[] args) throws Throwable {
        Summit s = new Summit();
        MethodType handleType = methodType(MethodHandle.class, Object[].class);
        MethodHandle handle = lookup().bind(s, "printArg", handleType);
        s.next = handle;

        MethodHandle looper =
            LoopHandle.createLoopHandle(handle, genericMethodType(4));
        looper.invoke("Hello", "JVM", "Lang", "Summit");
    }

    public MethodHandle printArg(Object... args) {
        System.out.println(i + ": " + args[i]);
        if (++i == args.length) {
            return null;
        }
        return next;
    }
}
```

```
--- Example Output ---
C:\intrels>java Summit
0: Hello
1: JVM
2: Lang
3: Summit
```

# LoopHandle example implications

- Custom “user” handles are possible
  - Simple pattern that's easy to implement
  - Performance characteristics will be similar to standard java.lang.invoke API handles

- Security stack  walks
  - Many methods walk the stack to determine the caller for security purposes
  - Beware: the helper class will be visible to the stack walks

- Best practice:
  - Lean on handles provided by the standard API
    - They will receive the most attention from JSR 292 implementors
  - Don't feel handcuffed by the standard API
    - If necessary, use this pattern to create the custom behaviour you need

# Performance hints

- Caveat: These are true today.  They may not be true tomorrow.

- Use invokeExact where possible
  - Provides the fastest path to run a handle
  - invoke is as fast when MethodTypes exactly match
    - Otherwise it needs to do expensive validation

- Put asType() operations in out-of-line paths to ensure that the main-line can use invokeExact
  - Only worth the effort if the asType() actually ends up only on the infrequent path
  - mh.asType(type).invokeExact() is equivalent to mh.invoke()

- invokedynamic is a lazily resolved invokeExact
  - ConstantCallSite is “free”
  - Mutable and Volatile both incur an additional dynamicInvoker() handle invocation

# 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 Oracle 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.