
JavaOne™

The Status of JSR 292 & InvokeDynamic

- John Rose

  Oracle JVM Architect

  September 24, 2013

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.

JSR 292 Update: Agenda

- Security
- API
- Implementation
- Performance

 |

Security

 |

Caller sensitivity

- A few dozen JDK methods are ***caller sensitive***
  - This means their caller's identity is an implicit argument
  - This implicit argument cannot be forged or spoofed
- Many methods use this identity to make security-sensitive decisions

```
class MyUntrustedApplet {
  public static void main(String... args) throws Throwable {
    System.setSecurityManager(new SecurityManager());
    Class.forName("sun.misc.Unsafe");   // caller ID = MyUntrustedApplet

}
```

 |

Caller sensitivity

- A few dozen JDK methods are ***caller sensitive***
  - This means their caller's identity is an implicit argument
  - This implicit argument cannot be forged or spoofed
- Many methods use this identity to make security-sensitive decisions

```
import java.lang.invoke.*;
class MyUntrustedApplet {
  public static void main(String... args) throws Throwable {
    System.setSecurityManager(new SecurityManager());
    Class.forName("sun.misc.Unsafe");   // caller ID = MyUntrustedApplet
    MethodHandle MH_forName = MethodHandles.lookup()
      .findStatic(Class.class, "forName",
            MethodType.methodType(Class.class, String.class));
    MH_forName.invokeWithArguments("sun.misc.Unsafe");  // caller ID = ??
  }
}
```

 |

Caller sensitivity update

- Explicitly mark these methods with @CallerSensitive
- Process them with special care when making method handles
- Enforce existing design rule:

MH invocation ≡ ***bytecode behavior***

- Result: Every “lookup” sees a distinct version of Class.forName, etc.

 |

Unstructured caller sensitivity: removed

- sun.reflect.Reflection.getCallerClass(4)
- Four? Really?
- Change #1: Parameter must be 1.
- Change #2: Methods to be marked and processed with care.

```
// NOTE: be very careful if you change the stack depth of this
// routine. The depth of the "getCallerClass" call is hardwired so
// that the compiler can have an easier time if this gets inlined.
private void doSecurityCheck(Object obj) throws IllegalAccessException {
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class caller = Reflection.getCallerClass(4);
```

 |

Unstructured caller sensitivity: ***restored***

- Security updates in JDK 7 sanitized and limited this method
- Broke certain customers: dynamic languages, loggers
- Hook restored in JDK 7, but is removed from JDK 8
- WIP to create a safer-saner replacement not based on magic ints

More precise capability checks: **Private access**

- MethodHandles.publicLookup vs. MethodHandles.lookup
  - One is “minimally trusted”, one is “full power”
  - The latter has ***private access***
- For a given lookup class C, a ***private access*** lookup can:
  - access private fields, methods, and constructors of C
  - create method handles which invoke caller sensitive methods
  - create method handles which emulate invokespecial instructions
  - avoid package access checks for classes accessible to C
  - create delegated lookup objects for nestmates of C

More precise capability checks: The flipside

- Less-trusted lookup objects have reduced capabilities
- The least trusted lookup, MHs.publicLookup():
  - can access only public fields, methods, and constructors
  - cannot create method handles to caller sensitive methods
  - cannot create method handles which emulate invokespecial
  - must incur package access checks for target method package
  - cannot create delegated lookup objects of greater power

Aligned security manager checks

- Calls to SecurityManger.checkPackageAccess aligned with bytecode
- A MH could be expressed as as constant pool constant...
- if and only if it can be expressed as a native bytecode behavior...
- if and only if the MH can be reflected using the Lookup API
- Practical outcome: SM checks are irrelevant to private access lookup

API

 |

New API: MethodHandleInfo

- A way to “crack” any CONSTANT_MethodHandle constant
- Also works on Lookup-derived MHs that “could have been” constants
- Capability based: You can only “crack” your own MHs

New API: MethodHandles.collectArguments

- A missing form of function composition
- Allows any sequence of N arguments to be filtered down to one
- Existing APIs allow filtering of ***all*** arguments, or ***one***, but not ***some***.

Clarifications

- Concept of “bytecode behavior” emphasized and expanded
- Concept of “private access” emphasized and expanded

Nits and gnats...

- Initialization order aligned with bytecode behavior (getstatic, etc.)
- Access checking aligned with bytecode behavior
- Certain exceptions clarified and defined
  - High arity corner cases
  - Wrong-arity calls to a spreader

 |

Implementation

 |

Implementation refresh: motivations

- security issues (see above)
- NoClassDefFound bug
- maintainability of assembly code
- method handle creation requires a visit to JNI code (for each MH)

implementation refresh: 2012/07/24
7023639: JSR 292 ... needs a fast path

- move calling logic to Java code (IR = "Lambda Forms")
- generate adapter bytecodes on the fly, using ASM, loaded as "anonymous classes"
- some adapters statically defined (maybe more later)
- erase reference types in adapters
- written in JDK 8, back ported to JDK 7u40
- composition uses low-level bytecodes and Java method calls

implementation refresh: upsides

- no assembly code (well, less than 100 lines)
- C++ LOC -13028 +5414; Java LOC -3510 +6973
- debugger can show internal method handle slots and call frames
- better framework for optimization...

(kill -SIGSECURITY $$)

implementation refresh: downsides

- interpretation needs a tune-down (LambdaForm.interpretWithArguments)
- "epic stack traces" (awkward use of nested IR constructs)
- storage leaks at scale (IR generation does not converge)
- JIT needs tuning for new code shapes

implementation refresh: downsides

- interpretation needs a tune-down (LambdaForm.interpretWithArguments)
- "epic stack traces" (awkward use of nested IR constructs)
- storage leaks at scale (IR generation does not converge)
- JIT needs tuning for new code shapes

Performance

 |

Current performance work

- improve classic JIT optimizations: inlining, escape analysis
  - retune inlining heuristics after implementation refresh
- iterating on JDK 8 Closures
  - bootstrap method performance, constant call sites
- iterating on Nashorn JavaScript implementation

Current performance work: some details

- fast path for MH.asType and generic MH.invoke
- more caching of direct method handle constants
- carefully targeted eager rendering to bytecodes (e.g., after BSM)
  - reduces visits to the LF interpreter
- better processing of repeated lambda forms
  - detect repeats and reuse generated bytecodes
  - avoid repeats by caching
- flattening of “epic backtraces”
- current compiler team push is through this CY


JavaOne™