Invokedynamic: Deep Dive

Vladimir Ivanov, @iwan0www
HotSpot JVM Compiler
Oracle

vladimir.x.ivanov@oracle.com
14.03.2015

Agenda

- Introduction
- Public API
- Internals

Introduction

JSR 292 History

Supporting Dynamically Typed Languages on the Java Platform

- 2011, July 28: Released as part of Java 7
- 2010: API refinement (e.g., BootstrapMethods)
- 2009: API refinement (e.g., CONSTANT_MethodHandle)
- 2008: API with Method Handles (Early Draft Review)
- 2007: Expert Group reboot
- 2006: JSR 292 Expert Group formed
- 2005: Initial design sketch

What's invokedynamic?

"In summary, invokedynamic…

- is a natural general purpose primitive
  - Not tied to semantics of a specific programming language
  - Flexible building block for a variety of method invocation semantics
- enables relatively simple and efficient method dispatch."

Gilad Bracha

Java Language Architect,
Sun Microsystems, JAOO, 2005

What's invokedynamic?

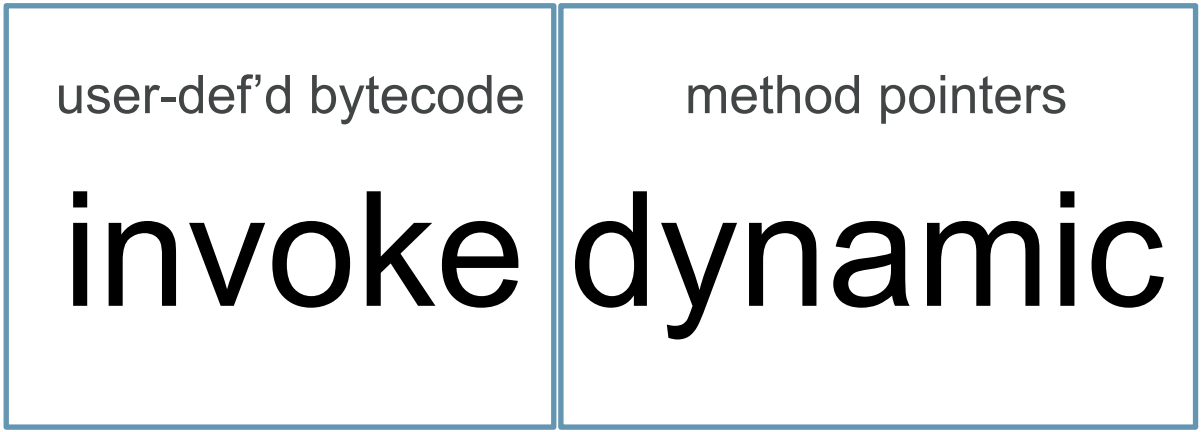
"In summary, invokedynamic…

- is a natural general purpose primitive
  - Not tied to semantics of a specific programming language
  - Flexible building block for a variety of method invocation semantics
- enables relatively simple and efficient method dispatch."

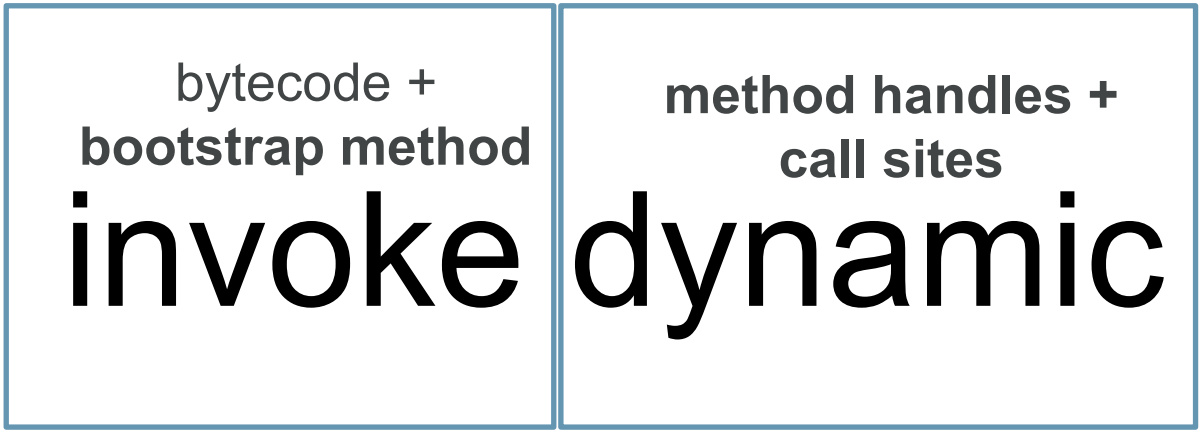
Gilad Bracha

Java Language Architect,
Sun Microsystems, JAOO, 2005

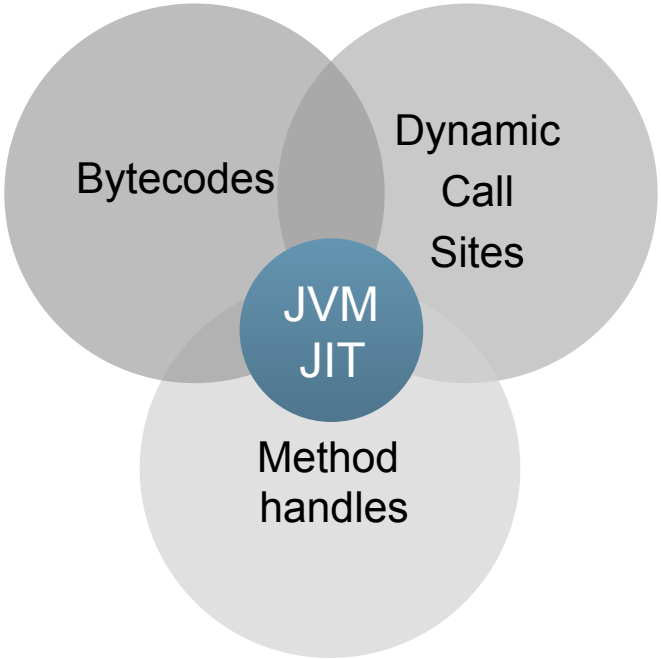
What's invokedynamic?



What's invokedynamic?



Architecture



Public API

bytecode + java.lang.invoke

invoke*

bytecode + bootstrap method

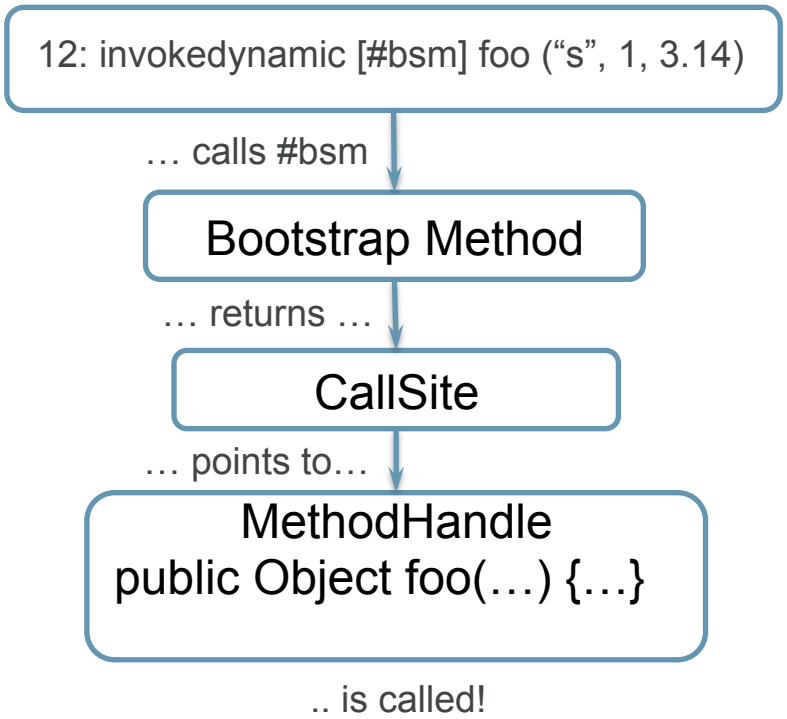
Invokedynamic

Comparison

invokestatic	invokespecial	invokevirtual	invokeinterface	invokedynamic
no receiver	1 receiver	1 receiver (class)	1 receiver (interface)	**no receiver**
no dispatch	no dispatch	single dispatch (via table)	single dispatch (via search)	**custom dispatch**

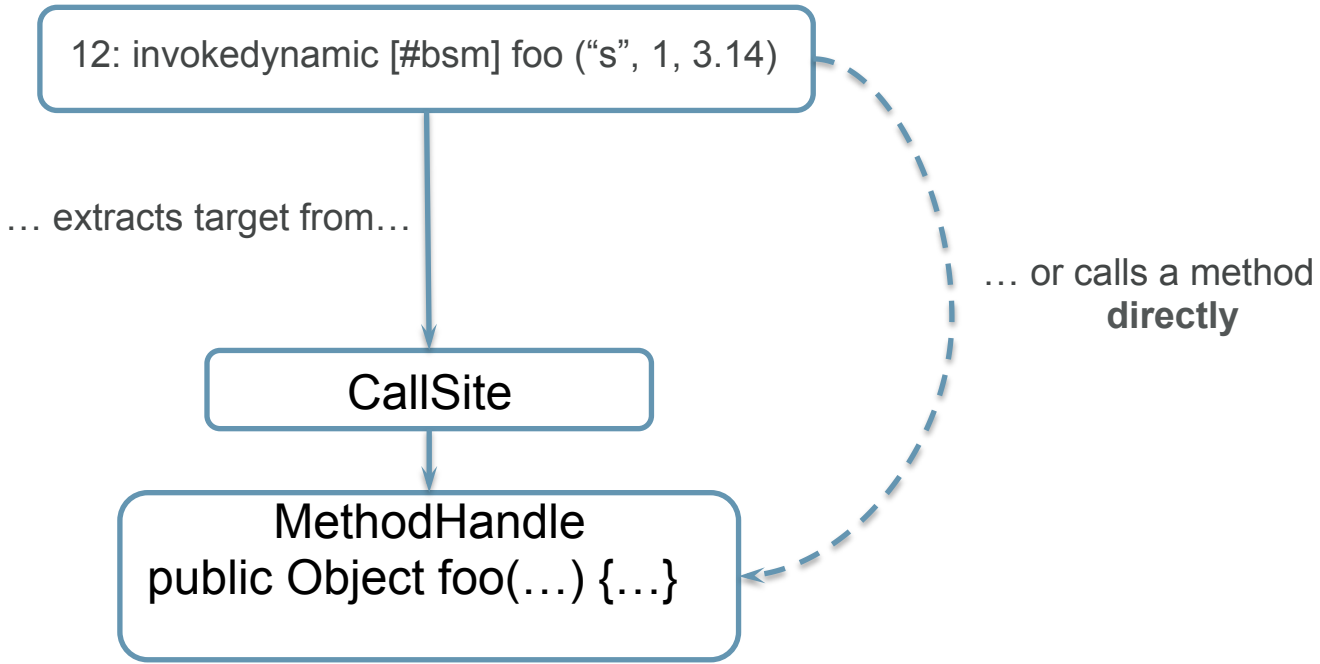
Invokedynamic Basics

1st Invocation



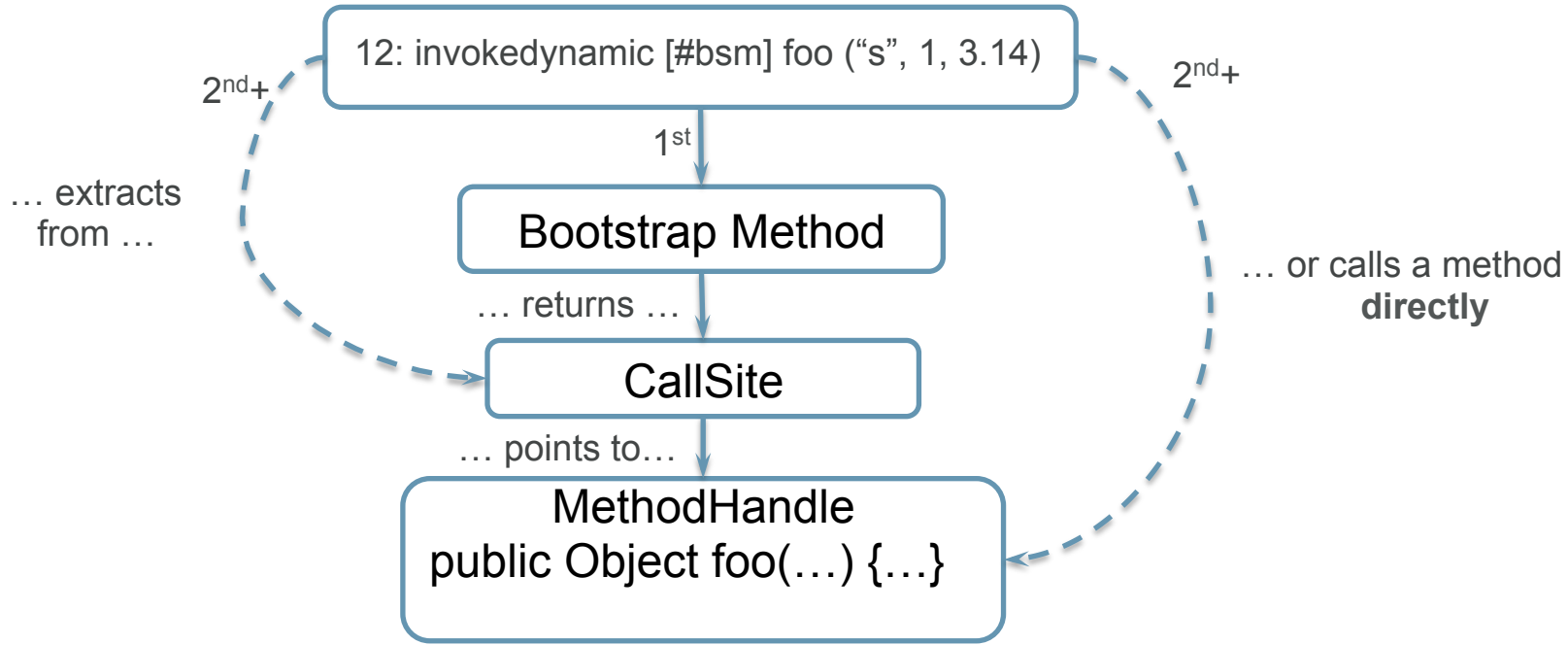
Invokedynamic Basics

2nd+ Invocation



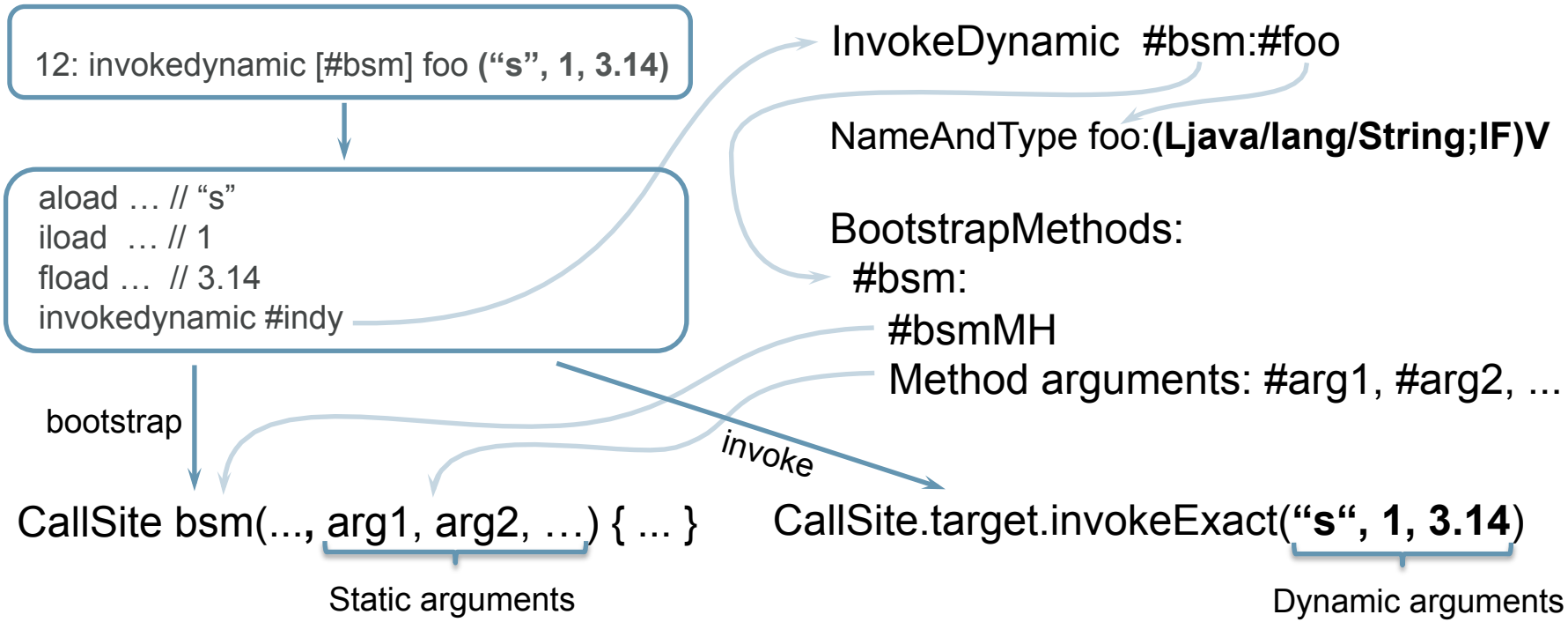
Invokedynamic Basics

Summary



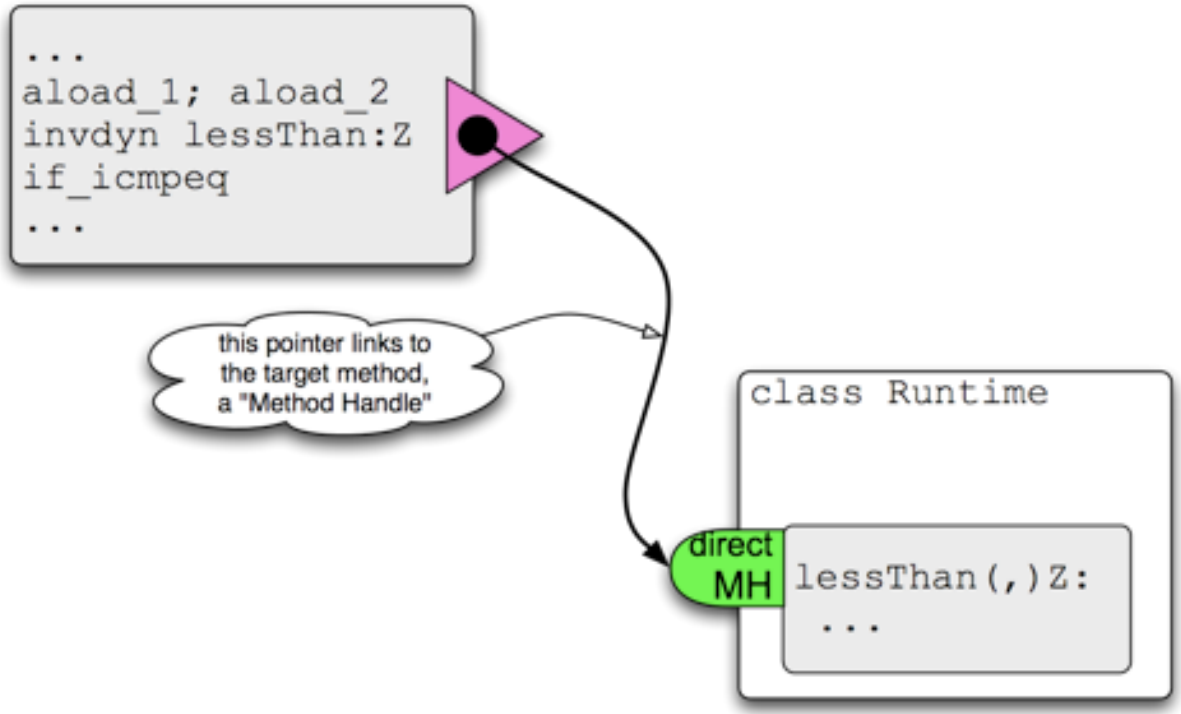
Invokedynamic Basics

Static vs Dynamic Arguments



Invokedynamic Basics

2nd+ Invocation



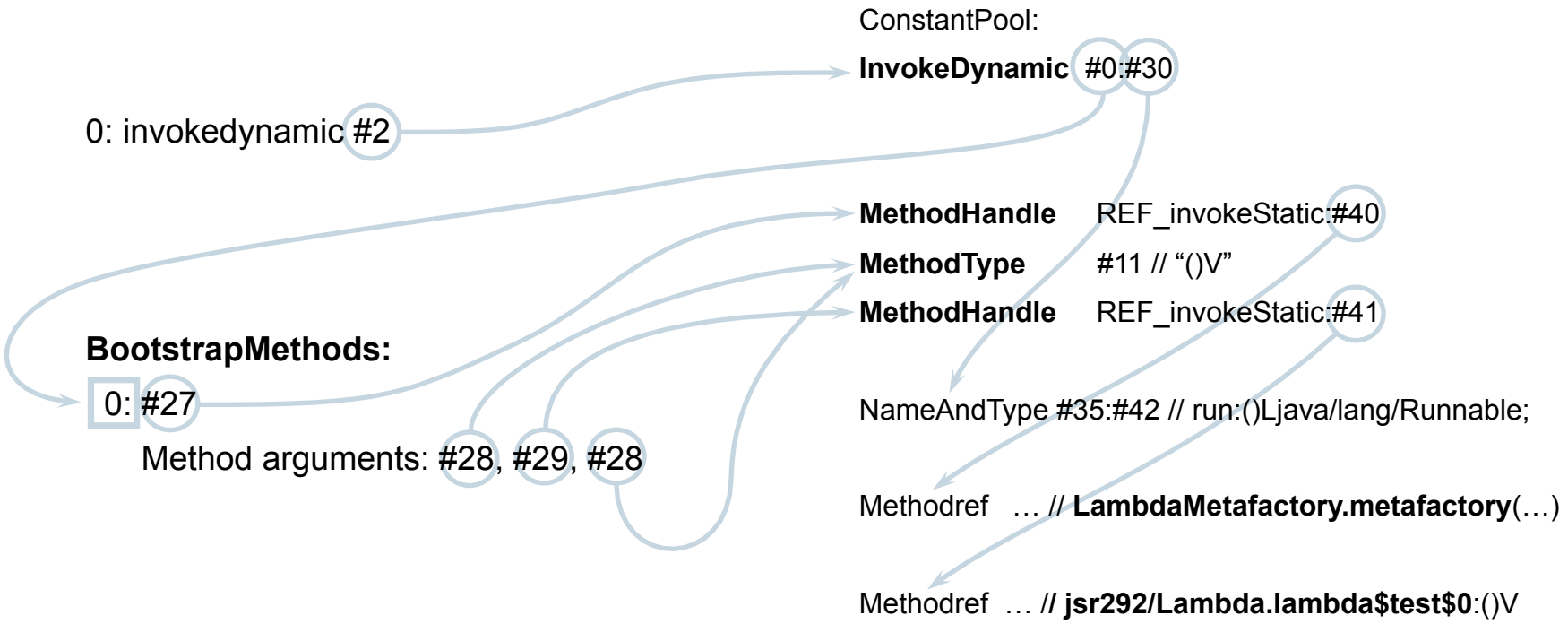
Example: Lambda Expression

Java & Bytecode

```
Runnable r = () -> System.out.println("run");
r.run();

Bytecode:
  0: invokedynamic #2,  0  // InvokeDynamic #0:run:()L...Runnable;
...
  7: invokeinterface #4,  1 // InterfaceMethod …Runnable.run:()V
```

Example: Lambda Expression

Linkage



Constant Pool Entries

JVMS-4.4.10. The CONSTANT_InvokeDynamic_info Structure

```
CONSTANT_InvokeDynamic_info {
    u1 tag                               = 18
    u2 bootstrap_method_attr_index : index into BootstrapMethods array
    u2 name_and_type_index         : CONSTANT_NameAndType_info
}
```

Constant Pool Entries

JVMS-4.4.8. The CONSTANT_MethodHandle_info Structure

```
CONSTANT_MethodHandle_info {
    u1 tag                 = 15
    u1 reference_kind      : [1..9] (JVMS-5.4.3.5)
    u2 reference_index     : (CONSTANT_Fieldref_info ||
}                              CONSTANT_Methodref_info ||
                               CONSTANT_InterfaceMethodref_info)
```

Constant Pool Entries

JVMS-4.4.9. The CONSTANT_MethodType_info Structure

```
CONSTANT_MethodType_info {
    u1 tag                 = 16
    u2 descriptor_index : CONSTANT_Utf8_info
}
```

Example: Lambda Expression

Bootstrap Method

```
j.l.i.LambdaMetafactory {
public static CallSite metafactory(  MethodHandles.Lookup caller,
                                     String invokedName,
                                     MethodType invokedType,
                                     MethodType samMethodType,
                                     MethodHandle implMethod,
                                     MethodType instantiatedMethodType)
    throws LambdaConversionException {
  AbstractValidatingLambdaMetafactory mf;
  mf = new InnerClassLambdaMetafactory(…);
  mf.validateMetafactoryArgs();
  return mf.buildCallSite();
}
```

symbolic info

static args

Example: Lambda Expression

Linked CallSite

```
CallSite buildCallSite() throws LambdaConversionException {
…
try {
    Object inst = ctrs[0].newInstance();
    return new ConstantCallSite(MethodHandles.constant(samBase, inst));
…
```

Example: Lambda Expression

Resolved case

Bytecode:

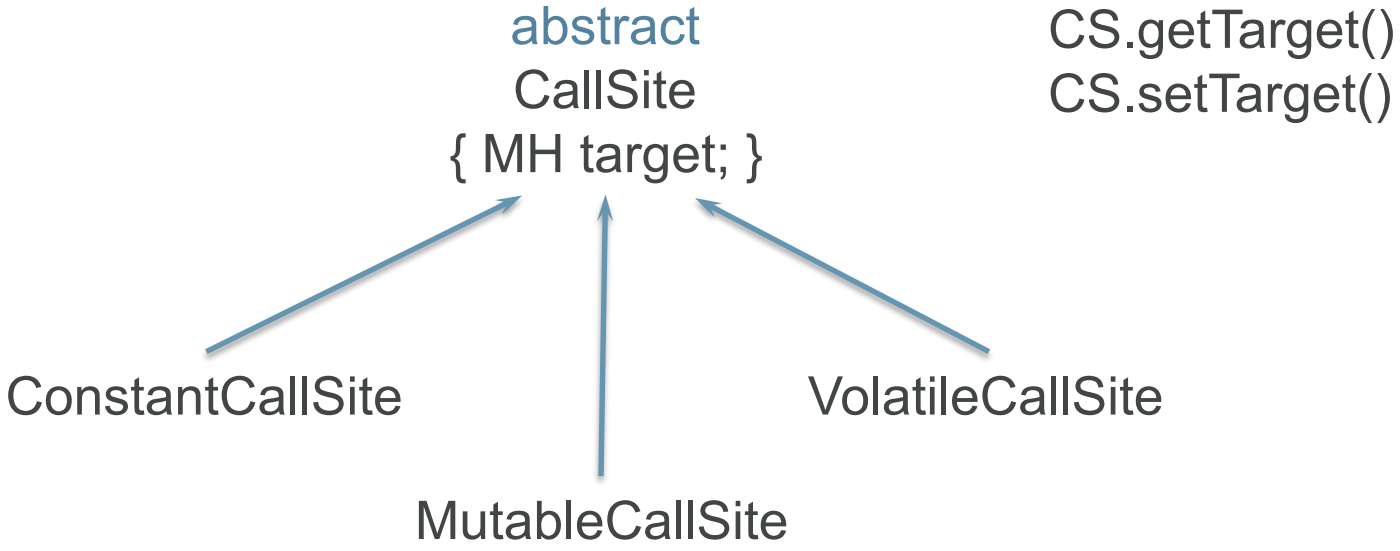
```
  0: invokedynamic #2,  0  // InvokeDynamic #0:run:()L...Runnable;
...
  7: invokeinterface #4,  1 // InterfaceMethod …Runnable.run:()V
```

Compilation log:

```
@ 0  …LambdaForm$MH027/…::linkToTargetMethod_000 (8 bytes)   force inline by annotation
   @ 4  …LambdaForm$MH026/…::identity_006_L (8 bytes)   force inline by annotation

@ 7   jsr292.Lambda$$Lambda$1/...::run (4 bytes)   inline (hot)
   @ 0   jsr292.Lambda::lambda$test$0 (9 bytes)   inline (hot)
```

CallSite

Hierarchy



… and user-defined call site types …

CallSite

State change

- CS.setTarget()
- Calls into VM:
  - MHN.setCallSiteTargetNormal()/setCallSiteTargetVolatile()
  - MHN_setCallSiteTargetNormal/MHN_setCallSiteTargetVolatile
- Why? For JIT purposes
  - both C1 & C2 optimistically inline through CallSites
  - it is recorded as a nmethod dependency (skipped for ConstantCS)
    - DepType::call_site_target_value
  - affected nmethods should be invalidated when CS.target changes

*dynamic

Method Handles

Method Handles

java.lang.invoke

- MethodHandle
  - *“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.”* javadoc
  - immutable, no visible state
- MethodType
  - arguments and return type descriptor for a MethodHandle

Method Handles

java.lang.invoke

- MethodHandles.Lookup
  - constructs MHs for methods, fields, …
  - e.g. findVirtual(), findGetter(), …
- MethodHandles
  - numerous MH adaptations and MH combinators
  - e.g. guardWithTest(), catchException(), filterReturnValue(), …

Method Handles

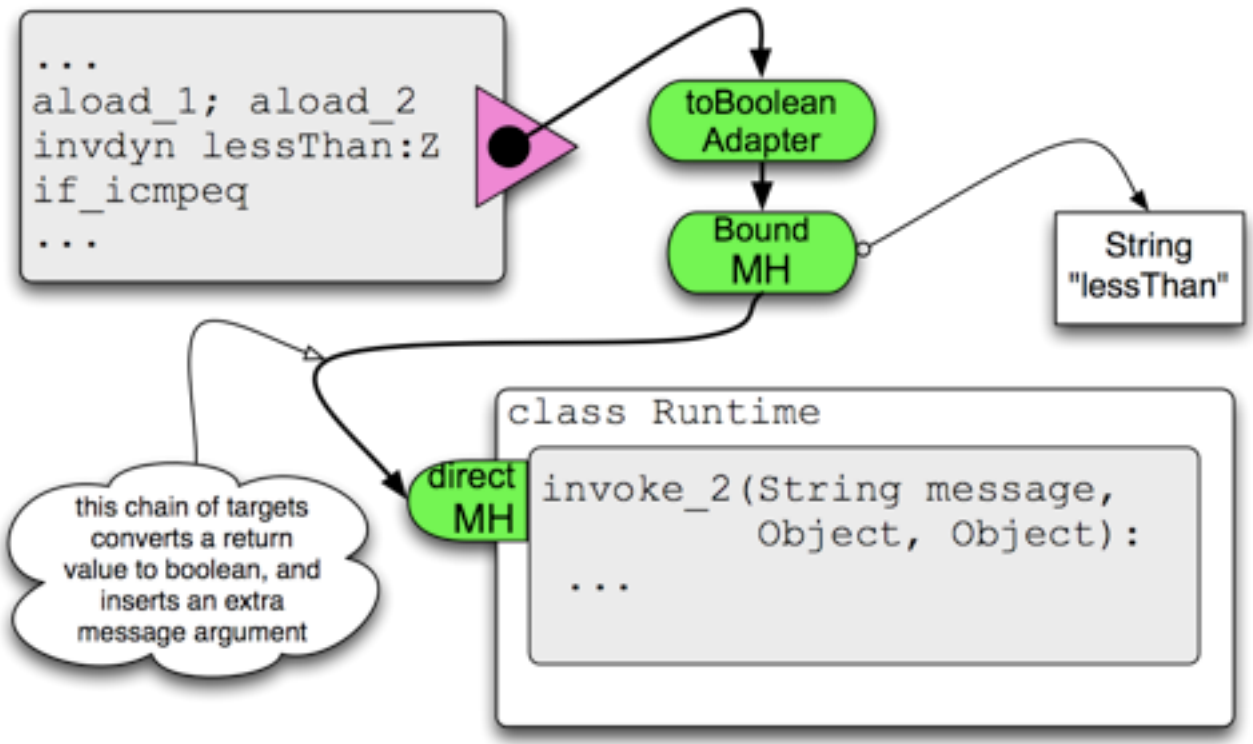
Lifecycle

- Construction (direct MHs)
  - reflective factory API: MethodHandles.Lookup
  - ldc of CONSTANT_MethodHandle
  - special factories: identity, invoker
- Transformation/adaptation (bound or adapter MHs)
  - bindTo, insertArguments, guardWithTest, etc.
  - asType, filterArguments, etc.
- Linkage (invokedynamic call site or a constant)
  - BSM or store in static final field
- Invocation (exact or inexact)

Method Handles

Usage

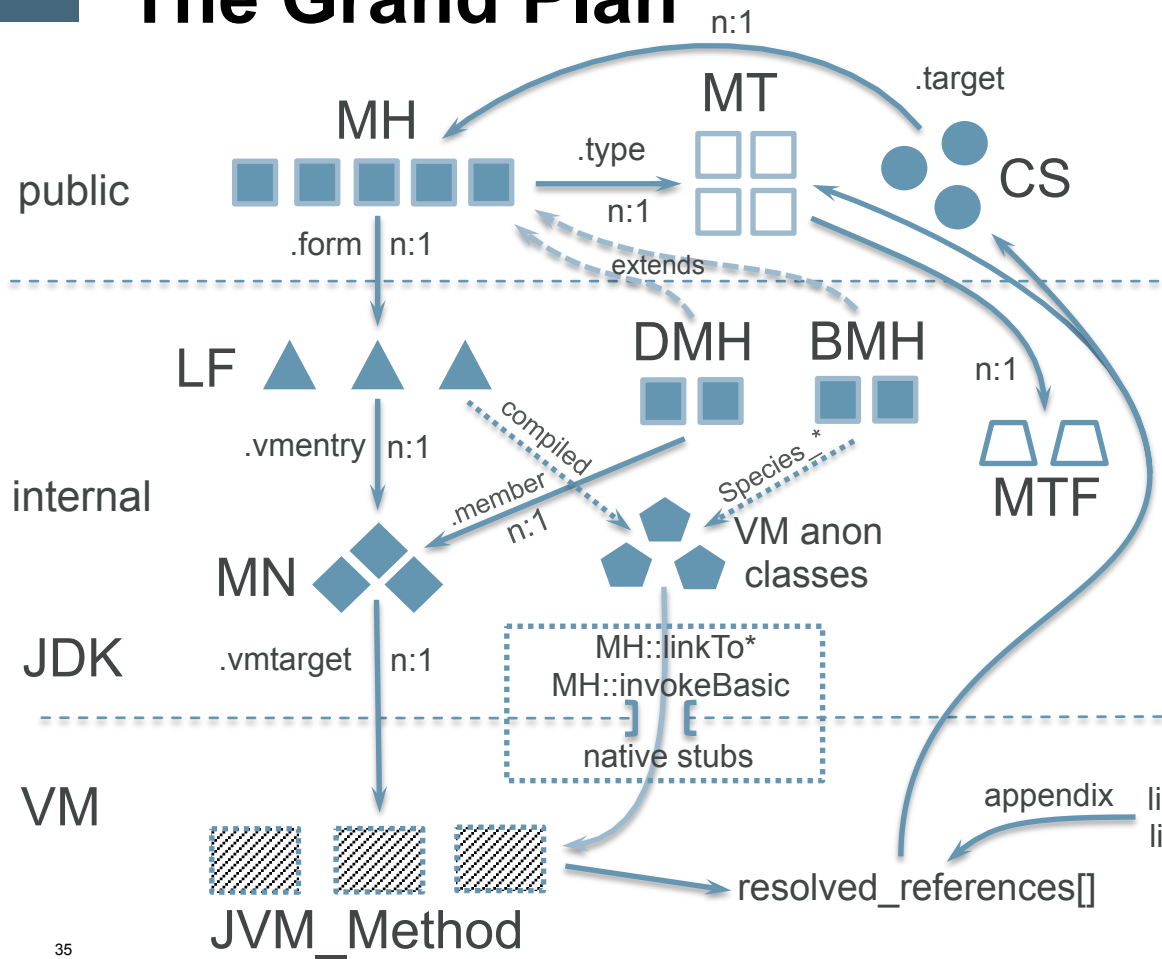
```
MethodHandles.Lookup LOOKUP = MethodHandles.lookup();

MethodHandle CONCAT =
        LOOKUP.findVirtual(
                String.class,
                "concat",
                methodType(String.class, String.class));

assertEquals("xy", (String) CONCAT.invokeExact("x", "y"));

MethodHandle CONCAT_x = CONCAT.bindTo("x");
assertEquals("xy", CONCAT_x.invoke("y"));
```



Internals

java.lang.invoke + VM

The Grand Plan



MH = MethodHandle
MT = MethodType
CS = CallSite

LF = LambdaForm
DMH = DirectMethodHandle
BMH = BoundMethodHandle
MN = MemberName
MTF = MethodTypeForm

MHN = MethodHandleNative

MHN

Upcalls

VM calls

appendix

linkCallSite
linkMethod
etc

MH::init/resolve
setCallSiteTarget*
etc

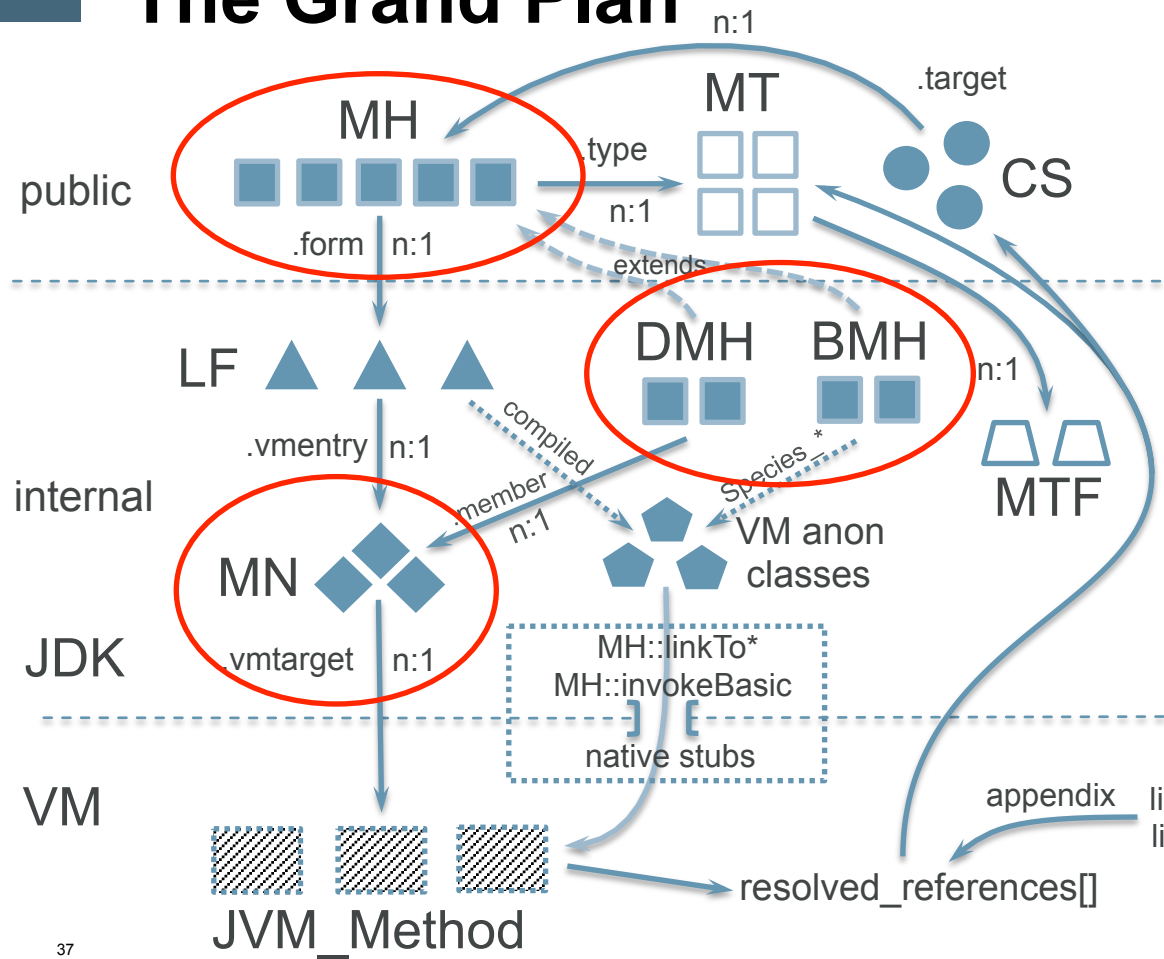
methodHandles*.cpp

ORACLE

Method Handles

Internals

The Grand Plan



MH = MethodHandle
MT = MethodType
CS = CallSite

LF = LambdaForm
DMH = DirectMethodHandle
BMH = BoundMethodHandle
MN = MemberName
MTF = MethodTypeForm

MHN = MethodHandleNative

MHN

Upcalls

VM calls

appendix

linkCallSite
linkMethod
etc

MH::init/resolve
setCallSiteTarget*
etc

methodHandles*.cpp

ORACLE

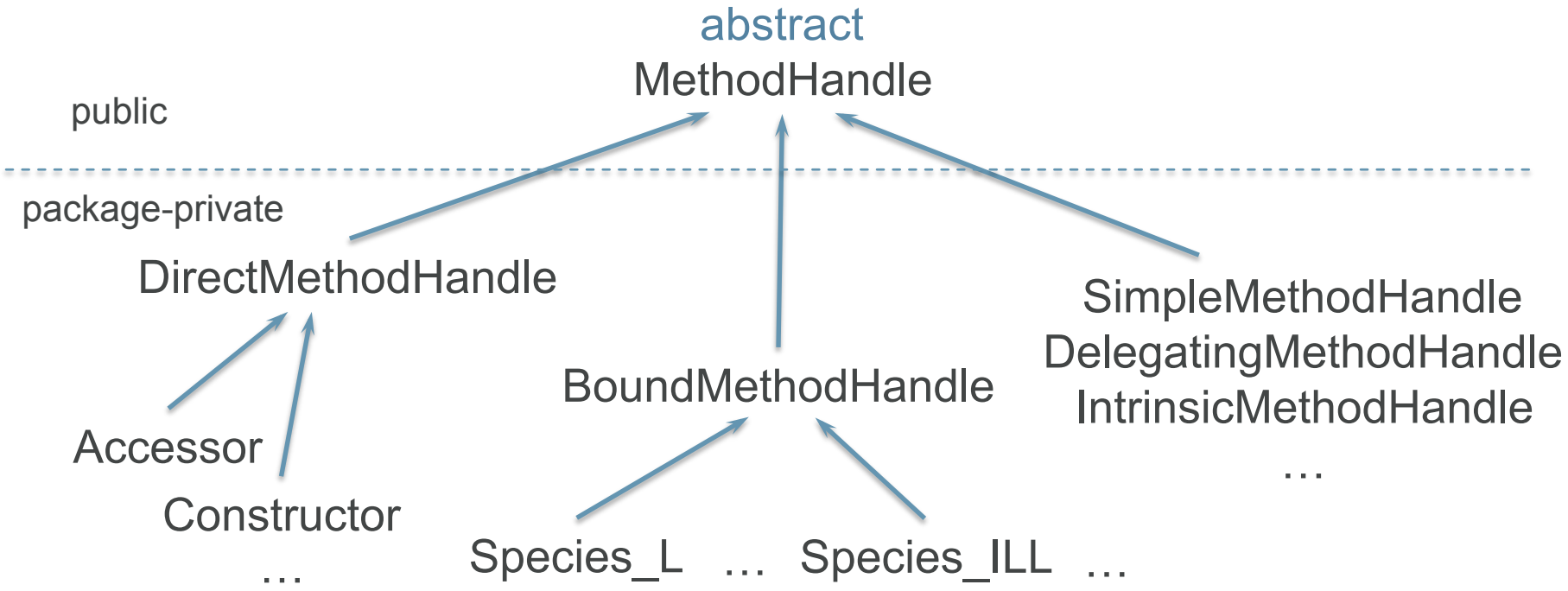
Method Handles

Lifecycle

- Construction (direct MHs)
  - reflective factory API: MethodHandles.Lookup
  - ldc: CONSTANT_MethodHandle
  - special factories: identity, invoker
- Transformation/adaptation (bound or adapter MHs)
  - bindTo, insertArguments, guardWithTest, etc.
  - asType, filterArguments, etc.
- Linkage (invokedynamic call site or a constant)
  - BSM or store in static final field
- Invocation (exact or inexact)

MethodHandle

Hierarchy



Method Handles

Lifecycle: Construction

- MH is linked to a concrete method
  - LambdaForm instance, which describes “behavior”
  - additional MemberName for direct MH (DMH)
- All access checks w.r.t. Lookup object
  - see MHs::checkAccess, VerifyAccess::isMemberAccessible, MHs::checkSecurityManager
- **NO** access/security checks during further usage!
  - the key to fast invocation!

BoundMethodHandle

Example

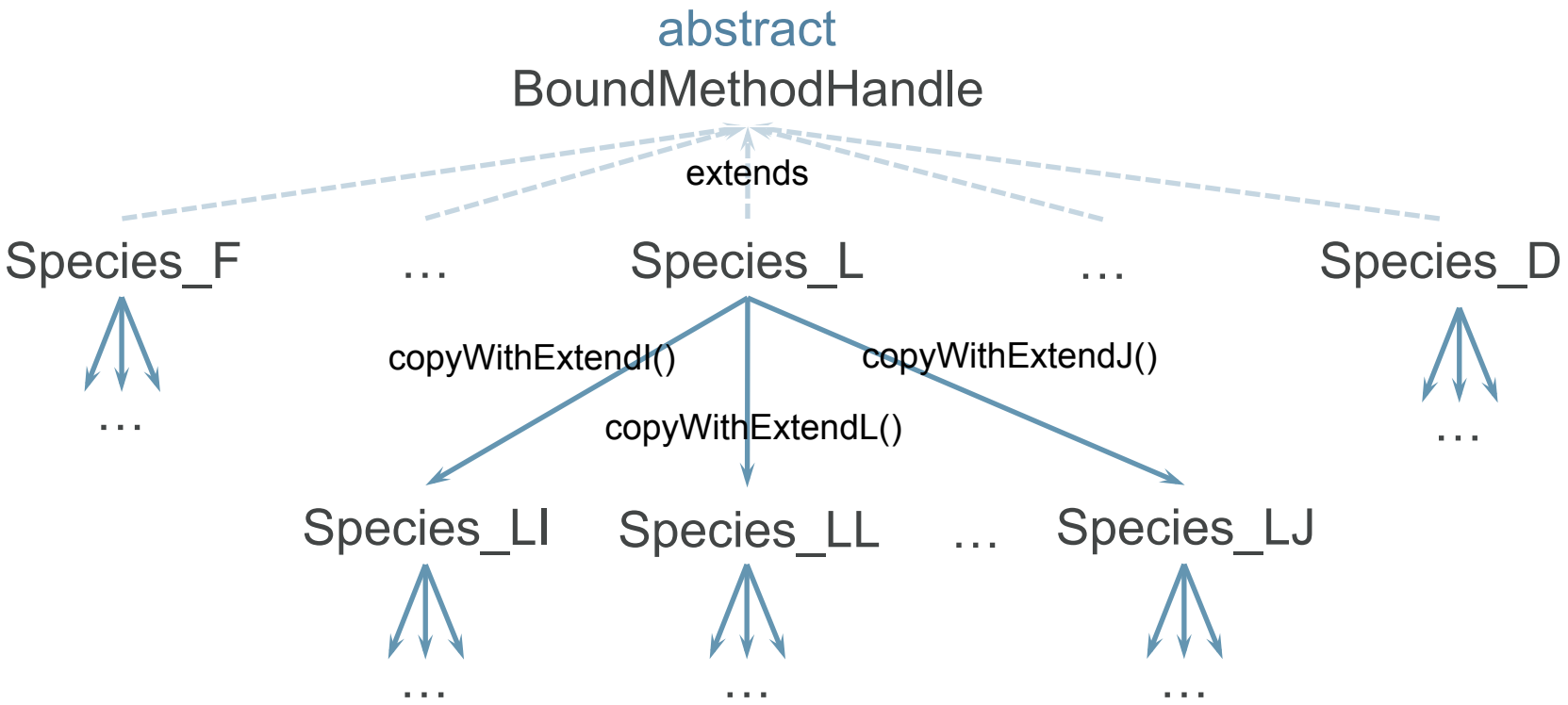
abstract
BoundMethodHandle

```
class Species_LLIJ extends BoundMethodHandle {
    private final Object argL0;
    private final Object argL1;
    private final int    argI2;
    private final long   argJ3;
```

BoundMethodHandle

Breeding

BMH.copyWithExtend*()



ORACLE

DirectMethodHandle

- MethodHandle
  - *"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."*

javadoc

DirectMethodHandle

```
/** The flavor of method handle which implements
  * a constant reference to a class member. */
class DirectMethodHandle extends MethodHandle {
    final MemberName member;

/** This subclass handles non-static field references. */
static class Accessor extends DirectMethodHandle {
    final Class<?> fieldType;
    final int      fieldOffset;


/** This subclass handles constructor references. */
static class Constructor extends DirectMethodHandle {
    final MemberName initMethod;
    final Class<?>   instanceClass;
```

Example: Lookup.findStatic

DMH construction

```
public
MethodHandle findStatic(Class<?> refc, String name, MethodType type)
    MemberName method =
            resolveOrFail(REF_invokeStatic, refc, name, type);
    return getDirectMethod(REF_invokeStatic, refc, method,
                           findBoundCallerClass(method));
}
```

Example: Lookup.findStatic

Resolve MemberName

```
private static final MemberName.Factory IMPL_NAMES =
        MemberName.getFactory();

MemberName resolveOrFail(byte refKind, Class<?> refc,
                         String name, MethodType type)
    // do this before attempting to resolve
    checkSymbolicClass(refc);
    name.getClass();  // NPE
    type.getClass();  // NPE
    checkMethodName(refKind, name);  // NPE check on name
    return IMPL_NAMES.resolveOrFail(refKind,
            new MemberName(refc, name, type, refKind),
            lookupClassOrNull(),
            NoSuchMethodException.class);
}
```

Example: Lookup.findStatic

DMH construction: access checks

```
/** Common code for all methods;
 * do not call directly except from immediately above. */
private MethodHandle getDirectMethodCommon(
                                byte refKind,
                                Class<?> refc,
                                MemberName method,
                                boolean checkSecurity,
                                boolean doRestrict,
                                Class<?> callerClass)
    checkMethod(refKind, refc, method);
    // Optionally check with the security manager;
    // this isn't needed for unreflect* calls.
    if (checkSecurity)
        checkSecurityManager(refc, method);
```

MemberName

Method/field/constructor pointer both VM/JDK understand.

```
/*non-public*/ final class MemberName implements Member, Cloneable {
    private Class<?> clazz;       // class in which the method is defined
    private String   name;        // may be null if not yet materialized
    private Object   type;        // may be null if not yet materialized
    private int      flags;       // modifier bits; see reflect.Modifier
    //@Injected JVM_Method* vmtarget;
    //@Injected int         vmindex;
    private Object   resolution;  // if null, this guy is resolved
```

MemberName

Lifecycle

- Initialization
  - make a symbolic reference (clazz, name, type, flags)
- Resolution
  - prepare an instance for actual usage (e.g. invocation)
  - compute vmtarget & vmindex

MemberName

Initialization

- Fills in symbolic info
  - happens in Java: MN.init()
  - sometimes, call into VM:
    - e.g. java.lang.reflect.Method/Field => MemberName

Stack trace:

1. MethodHandles::init_MemberName
2. MHN_init_Mem
3. MHN.init(MemberName self, Object ref)

MemberName

Resolution

- Computes vmtarget/vmindex from symbolic reference
- MN resolution happens in VM w/o access checks
  - access checks happen before resolution step on symbolic ref and later when DMH is constructed
    - see MHs.checkSymbolicClass, VerifyAccess.isClassAccessible, MHs.checkMethod, VerifyAccess.isMemberAccessible

Stack trace:

1. MethodHandles::resolve_MemberName
2. MHN_resolve_Mem
3. MHN.resolve()
4. MN.resolve()

MemberName

Class Redefinition Support

- MemberNameTable
  - MemberNameTable::adjust_method_entries(…)
- Will be moved to Java
  - see https://bugs.openjdk.java.net/browse/JDK-8013267

Method Handles

Lifecycle: Invocation

- 2 ways to invoke a MH:
  - indy insn in bytecode
  - MH.invoke/.invokeExact methods

- invoke/invokeExact are “signature polymorphic”
  - method signature is defined by call site signature in bytecode
  - e.g (int)mh.invoke(1, 1L, new Object()) => MH.invoke(int, long, Object)int

Signature Polymorphism

```
public final native @PolymorphicSignature
Object invokeExact(Object... args) throws Throwable;
```

"… a signature polymorphic method is one which can operate with any of a wide range of call signatures and return types.", javadoc

"… [JVM] will successfully link any such call, regardless of its symbolic type descriptor.", javadoc

Signature polymorphic method by JLS (JLS-§15.12.3):

native MethodHandle::*(Object[])Object method

Method Handles

Lifecycle: Invocation

- “Exact” invocation (indy or MH.invokeExact())
  - type check (MH type == invoker type), then call
  - if fails, throw WrongMethodTypeException
- “Inexact” / “generic” invocation (MH.invoke())
  - type check, then call
  - otherwise, type conversion attempt: target.asType(callSiteType)
  - if fails, throw WrongMethodTypeException

asType conversion

MH::asType(MT) => MH

$$oldType: (T^1_1, \ldots, T^1_n): T^0_R$$

$$newType: (T^0_1, \ldots, T^0_n): T^1_R$$

$T^0 \Rightarrow T^1$	primitive	ref	void
primitive	method invocation conversion (JLS-5.3)	assignment[1] (JLS-5.2)	discarded
ref	unbox, then primitive => primitive	cast to T1	discarded
void	zero value	null value	discarded

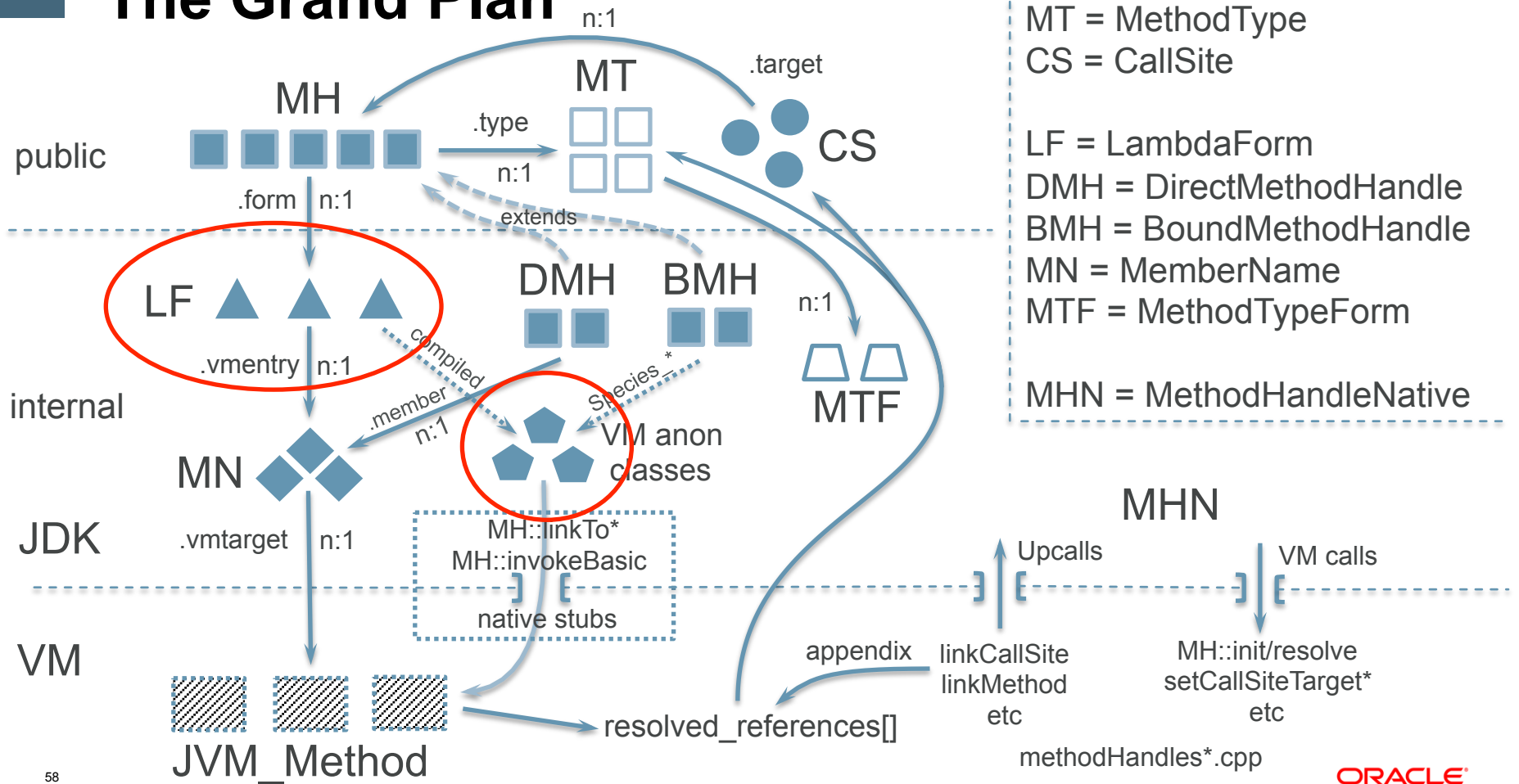
http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html#asType-java.lang.invoke.MethodType-

1: spec says casting (JLS-5.5) which mandates the same behavior for the prim/ref case

LambdaForms

IR for Method Handles

The Grand Plan



ORACLE

LambdaForm

“The symbolic, non-executable form of a method handle's invocation semantics.”, javadoc

Linear array of Names

- first arguments, then expressions

```
class LambdaForm {
    final int arity;
    final int result;
    final boolean forceInline;
    final MethodHandle customized;
    @Stable final Name[] names;
    final String debugName;
    MemberName vmentry;
    private boolean isCompiled;
```

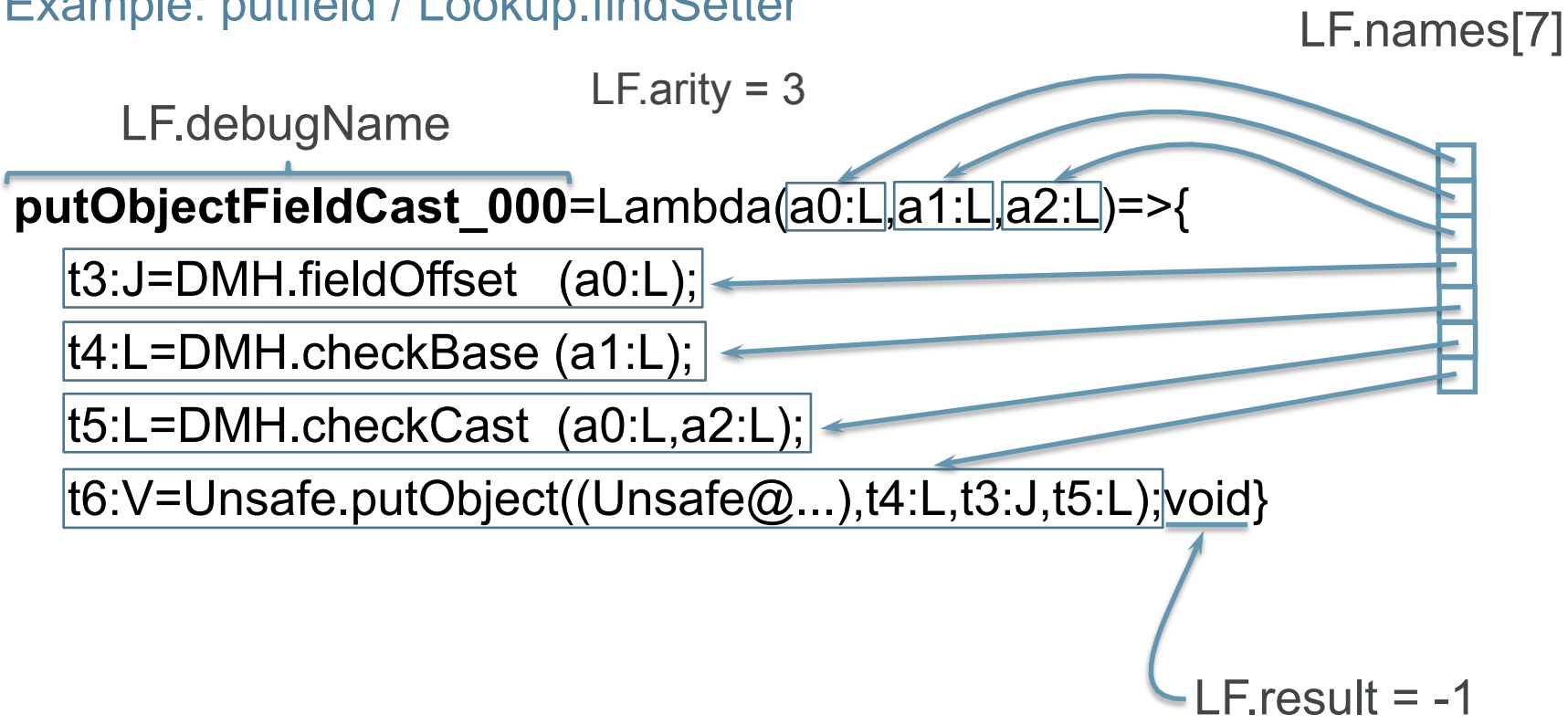
LambdaForm

Example: putfield / Lookup.findSetter

```
putObjectFieldCast_000=Lambda(a0:L,a1:L,a2:L)=>{
    t3:J=DMH.fieldOffset   (a0:L);
    t4:L=DMH.checkBase (a1:L);
    t5:L=DMH.checkCast  (a0:L,a2:L);
    t6:V=Unsafe.putObject((Unsafe@...),t4:L,t3:J,t5:L);void}
```

LambdaForm

Example: putfield / Lookup.findSetter



LambdaForm

Expression

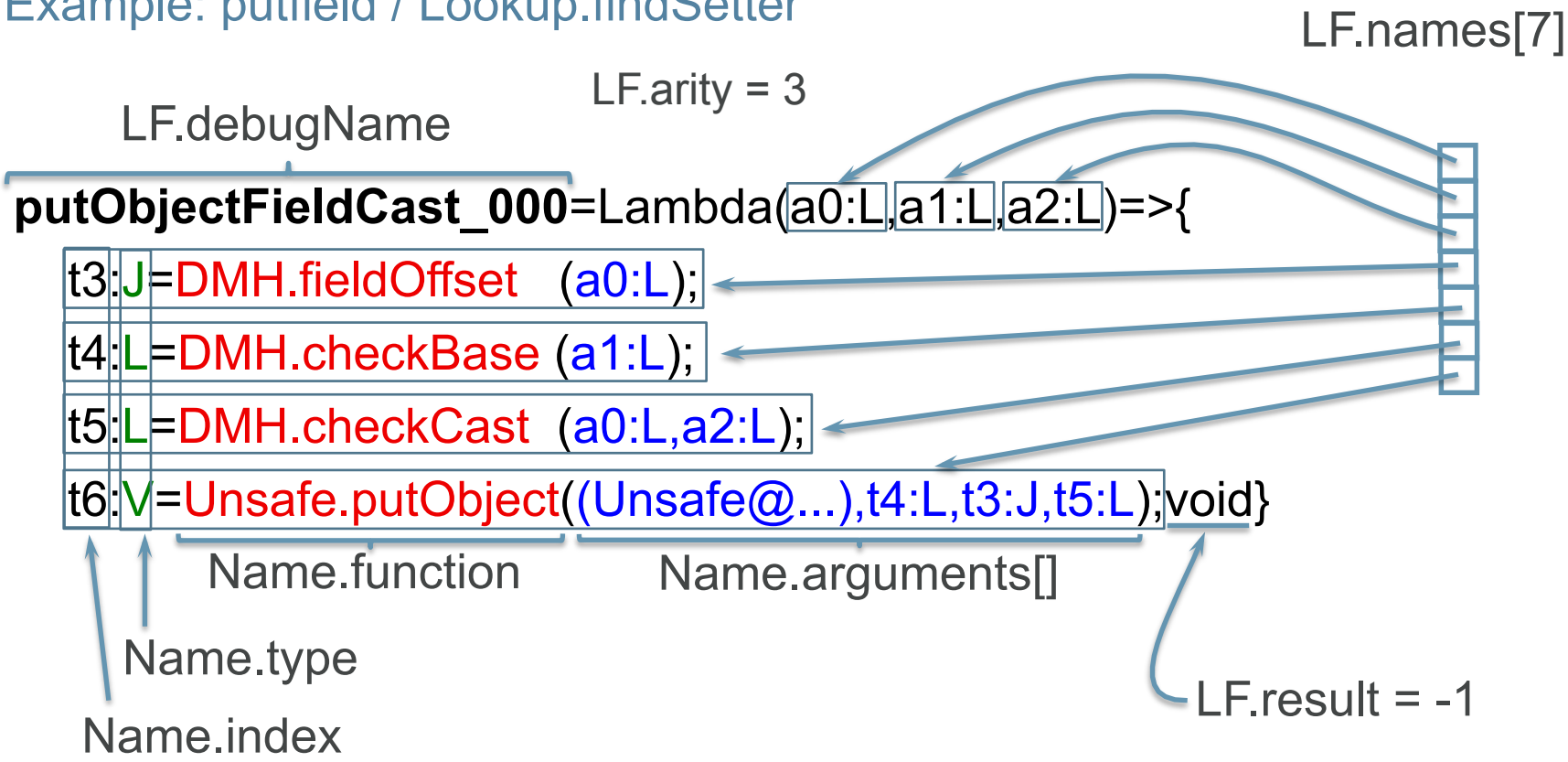
```
static final class Name {
    final BasicType type;
    private short index;
    final NamedFunction function;
    final Object constraint;
    @Stable final Object[] arguments;
```

```
static class NamedFunction {
    final MemberName member;
    @Stable MethodHandle resolvedHandle;
    @Stable MethodHandle invoker;
```

- An expression is a NamedFunction with arguments
  - named function is a symbolic reference on Boot Class Path
  - argument array contains (previous) Names and/or Objects
- Weakly typed (5 basic types: I, J, F, D, L)
- No symbolic names (just local Name pointers)
- No control flow (except early exit), so trivially SSA

LambdaForm

Example: putfield / Lookup.findSetter



LambdaForm

Execution

- Entry point: LambdaForm::**vmentry** : MemberName

- Can point to:
  - LambdaForm Interpreter
    - generated entrypoint: LFI::interpret_*
      - see LambdaForm::getPreparedForm()
    - calls LF::interpretWithArguments
  - compiled bytecode
    - by InvokerBytecodeGenerator

LambdaForm

Interpreter

```
@Hidden
@DontInline
/** Interpretively invoke this form on the given arguments. */
Object interpretWithArguments(Object... argumentValues) throws Throwable {
    checkInvocationCounter();
    Object[] values = Arrays.copyOf(argumentValues, names.length);
    for (int i = argumentValues.length; i < values.length; i++) {
        values[i] = interpretName(names[i], values);
    }
    Object rv = (result < 0) ? null : values[result];
    return rv;
}
```

LambdaForm

Compiled Form

a0:L a1:L a2:L

static void **putObjectFieldCast_000**(Object, Object, Object);

t3:J
```
0: aload_0
1: invokestatic  #16
4: lstore_3
```
// DMH.fieldOffset:...

t4:L
```
5: aload_1
6: invokestatic  #20
9: astore        5
```
// DMH.checkBase:...

t5:L
```
11: aload_0
12: aload_2
13: invokestatic  #24
16: astore        6
```
// DMH.checkCast:...

t6:V
```
18: ldc           #26
20: checkcast     #28
23: aload         5
25: lload_3
26: aload         6
28: invokevirtual #32
31: return
```
// String CONSTANT_PLACEHOLDER_0 <<sun.misc.Unsafe@3830f1c0>>
// class sun/misc/Unsafe
// Unsafe.putObject:(Object;Object;)V

RuntimeVisibleAnnotations:

LambdaForm$Hidden, LambdaForm$Compiled, ForceInline

```
putObjectFieldCast_000=Lambda(a0:L,a1:L,a2:L)=>{
  t3:J=DMH.fieldOffset   (a0:L);
  t4:L=DMH.checkBase (a1:L);
  t5:L=DMH.checkCast  (a0:L,a2:L);
  t6:V=Unsafe.putObject((Unsafe@...),t4:L,t3:J,t5:L);void}
```

VM Anonymous Classes

```
public final class Unsafe {

    public native Class<?> defineAnonymousClass(Class<?> hostClass,
                                                byte[] data,
                                                Object[] cpPatches);
```

"*Define a class but do not make it known to the class loader or system dictionary.*"
javadoc

Corollary: can be unloaded once it's not used.

hostClass: context for linkage, access control, protection domain, and class loader

cpPatches: non-null patches replace corresponding CP entries in loaded class

VM Anonymous Classes

Constant Pool Patching

static void **putObjectFieldCast_000**(…);
...

t6:V

```
18: ldc          #26
    // String “CONSTANT_PLACEHOLDER_0“

20: checkcast    #28
    // class sun/misc/Unsafe

23: aload        5
25: lload_3
26: aload        6
28: invokevirtual #32 // Unsafe.putObject(...)
31: return
```

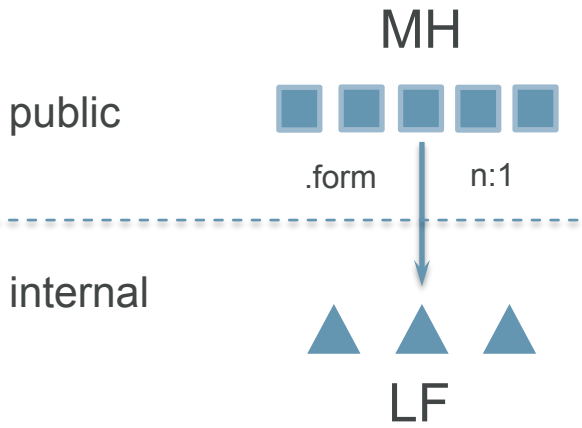
```
putObjectFieldCast_000=Lambda(a0:L,a1:L,a2:L)=>{
  t3:J=DMH.fieldOffset   (a0:L);
  t4:L=DMH.checkBase (a1:L);
  t5:L=DMH.checkCast  (a0:L,a2:L);
  t6:V=Unsafe.putObject((Unsafe@...),t4:L,t3:J,t5:L);void}
```

```
Unsafe.defineAnonymousClass(
                LambdaForm.class,
                byte[] {…},
                Object[] { …, unsafeObj, …})
                                  #26
```

```
ConstantPool
#26 = Utf8 “...”
...
resolved_references[] { ..., unsafeObj , ...}
```

MHs vs LFs

*"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."*

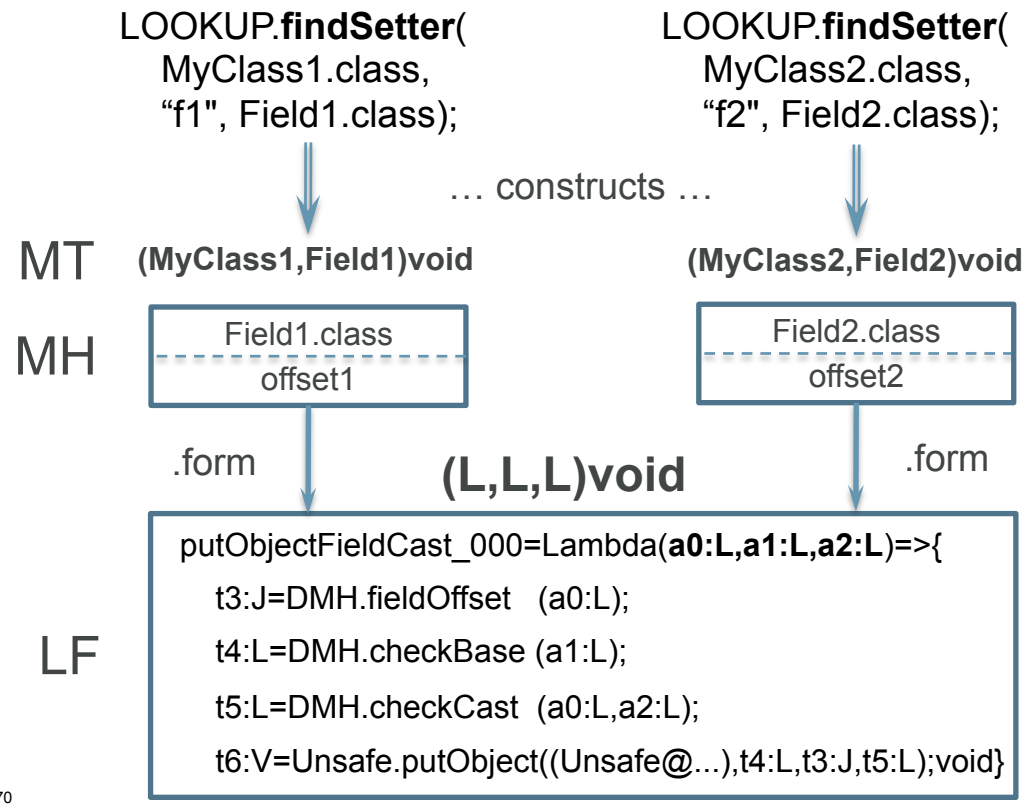


*" [A **lambda form** is a] … symbolic, non-executable form of a method handle's invocation semantics."*

javadoc

MHs vs LFs

Example: putfield / Lookup.findSetter




```
static class Accessor
        extends DirectMethodHandle {
    final Class<?> fieldType;
    final int      fieldOffset;
```

MHs vs LFs

Sharing vs Customization

- Shared LambdaForm (n to 1)
  - many MethodHandles share a single LambdaForm instance
- Customized LambdaForm (1 to 1)
  - LambdaForm instance is customized for some particular MethodHandle

Sharing vs Customization

Example: MHs.guardWithTest()

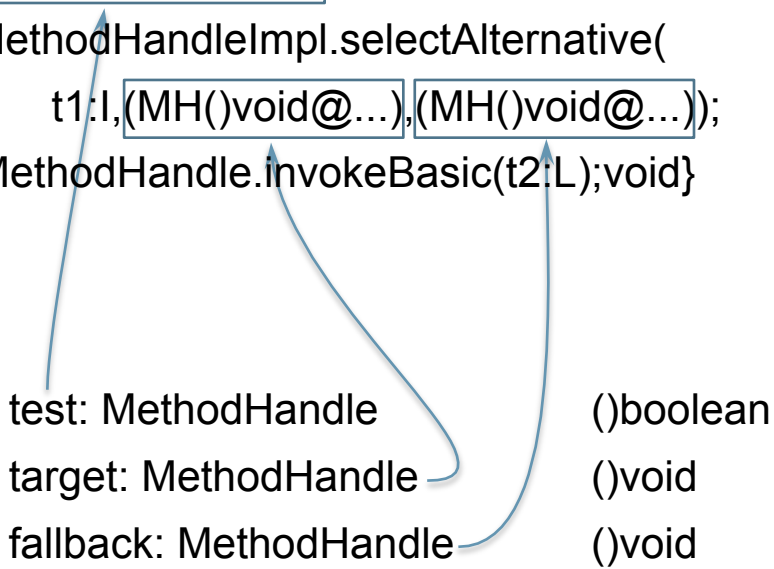
```
public static
MethodHandle guardWithTest(MethodHandle test,
                           MethodHandle target,
                           MethodHandle fallback) {
```

```
T guard(A... a, B... b) {
    if (test(a...))
        return target(a..., b...);
    else
        return fallback(a..., b...);
}
```

Sharing vs Customization

Example: Customized version

```
guard_000=Lambda(a0:L)=>{
    t1:I =(MH()boolean@...);
    t2:L=MethodHandleImpl.selectAlternative(
                t1:I,(MH()void@...),(MH()void@...));
    t3:V=MethodHandle.invokeBasic(t2:L);void}
```



Sharing vs Customization

Example: Customized version

```
guard_000=Lambda(a0:L)=>{
    t1:I =(MH()boolean@...);
    t2:L=MethodHandleImpl.selectAlternative(
                t1:I,(MH()void@...),(MH()void@...));
    t3:V=MethodHandle.invokeBasic(t2:L);void}
```

test: MethodHandle

target: MethodHandle

fallback: MethodHandle

static void guard_000(Object);

```
ldc          ... // <<MethodHandle()boolean>>
checkcast    ... // class MethodHandle
invokevirtual ... // MethodHandle.invokeBasic:()I
```

t1:I

```
…
if_icmpne    …
ldc          ... // <<MethodHandle()void>>
checkcast    ... // class MethodHandle
astore_2
aload_2
invokevirtual ... // MethodHandle.invokeBasic:()V
goto         …

ldc          ... // <<MethodHandle()void>>
checkcast    ... // class MethodHandle
astore_2
aload_2
invokevirtual ... // MethodHandle.invokeBasic:()V
```

t2:L + t3:V

return

Sharing vs Customization

Example: Shareable version

```
guard_000=Lambda(a0:L)=>{
    t3:L=BMH$Species_L3.argL0(a0:L);   // test
    t4:L=BMH$Species_L3.argL1(a0:L);   // target
    t5:L=BMH$Species_L3.argL2(a0:L);   // fallback
    t6:I =MethodHandle.invokeBasic(t3:L);
    t7:L=MethodHandleImpl.selectAlternative(t6:I,t4:L,t5:L);
    t8:V=MethodHandle.invokeBasic(t7:L);void}
```

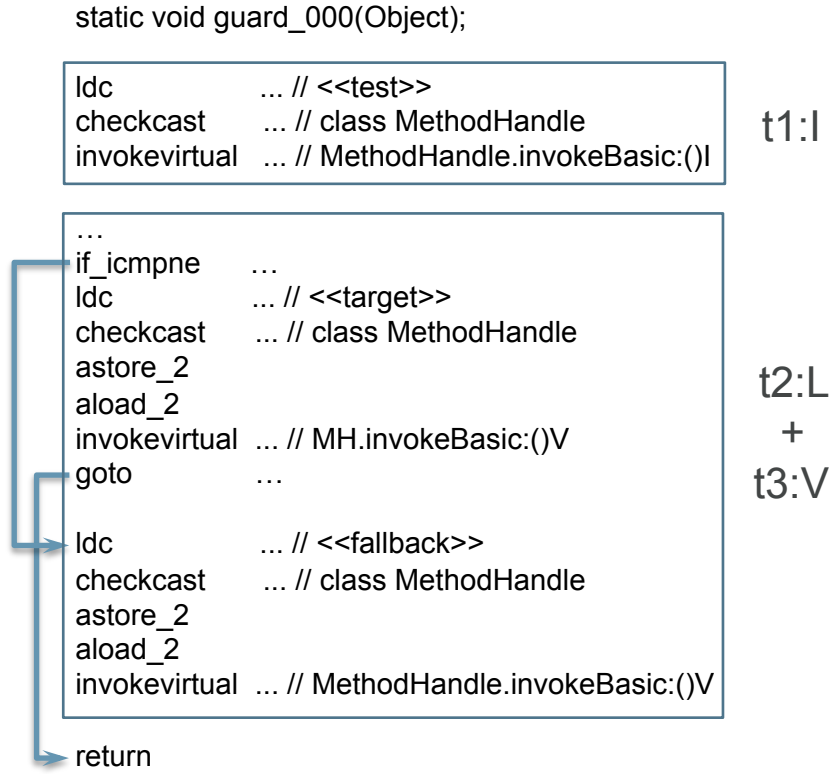
LF Sharing

- Pros:
  - improved application startup & warmup times
  - reduced dynamic footprint
- Cons:
  - profile pollution (peak performance suffers)
  - less optimized machine code when non-inlined MH calls

Profile Pollution

Customized version

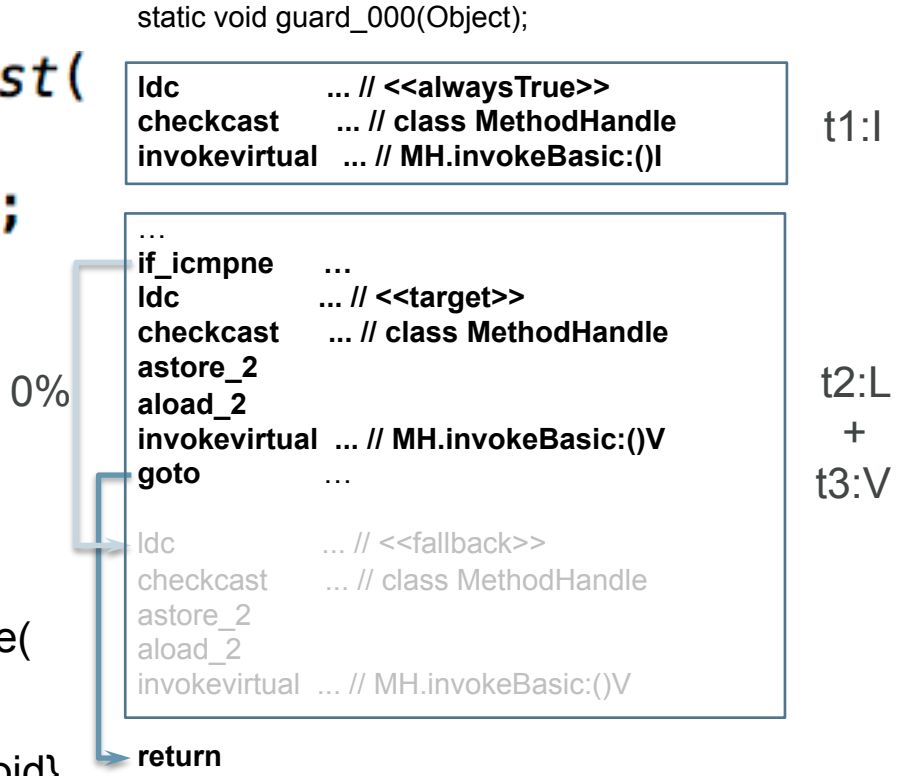
```
guard_000=Lambda(a0:L)=>{
    t1:I =<<test>>;
    t2:L=MethodHandleImpl.selectAlternative(
                t1:I,<<target>>,<<fallback>>);
    t3:V=MethodHandle.invokeBasic(t2:L);void}
```



Profile Pollution

Customized version

```
MethodHandles.guardWithTest(
        alwaysTrue,
        target, fallback);
```

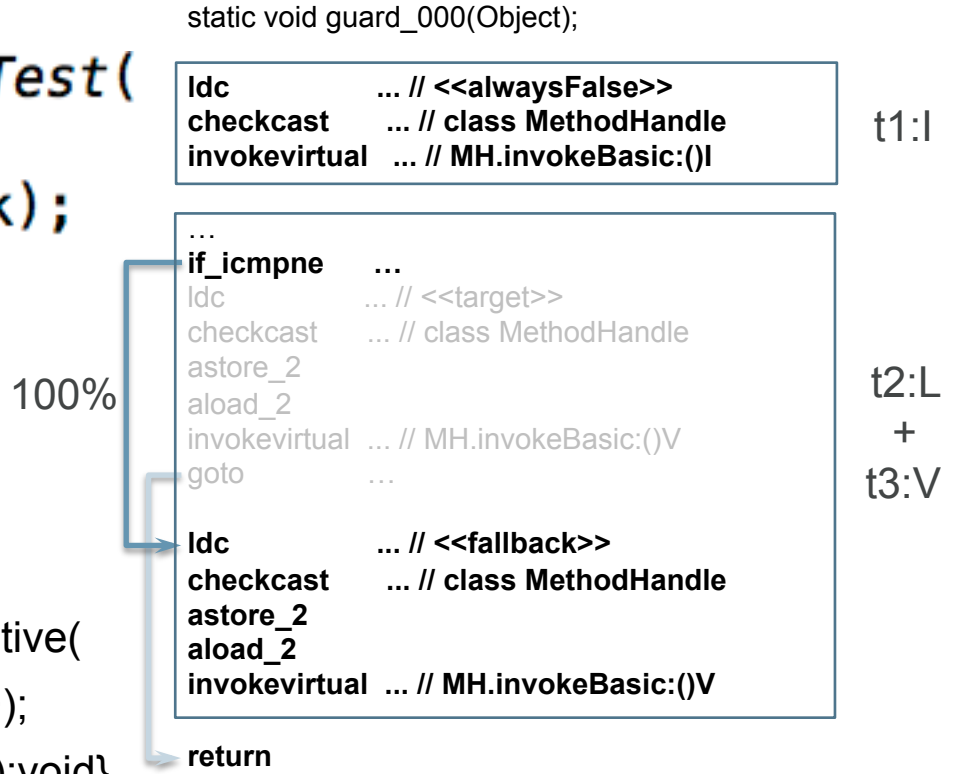



```
guard_000=Lambda(a0:L)=>{
   t1:I =<<alwaysTrue>>;
   t2:L=MethodHandleImpl.selectAlternative(
            t1:I,<<target>>,<<fallback>>);
   t3:V=MethodHandle.invokeBasic(t2:L);void}
```

Profile Pollution

Customized version

```
MethodHandles.guardWithTest(
        alwaysFalse,
        target, fallback);
```




```
guard_000=Lambda(a0:L)=>{
   t1:I =<<alwaysFalse>>;
   t2:L=MethodHandleImpl.selectAlternative(
            t1:I,<<target>>,<<fallback>>);
   t3:V=MethodHandle.invokeBasic(t2:L);void}
```

Profile Pollution

Shared version

```
MethodHandles.guardWithTest(
        alwaysTrue,
        target, fallback);

MethodHandles.guardWithTest(
        alwaysFalse,
        target, fallback);
```

?%

```
static void guard_000(Object);

aload_0
checkcast      … // BoundMethodHandle$Species_L3
getfield       ... // BMH$Species_L3.argL0:Object
invokevirtual  ... // MH.invokeBasic:()I
…
if_icmpne      …
aload_2
checkcast      ... // class MethodHandle
astore_5
aload_5
invokevirtual  ... // MH.invokeBasic:()V
goto           …

aload_3
checkcast      ... // class MethodHandle
astore_5
aload_5
invokevirtual  ... // MH.invokeBasic:()V

return
```

Non-Inlined Call

```
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodHandle test1 = lookup.findStatic(GWT.class, "test1", methodType(boolean.class));
MethodHandle    f1 = lookup.findStatic(GWT.class, "f1",    methodType(void.class));
MethodHandle    f2 = lookup.findStatic(GWT.class, "f2",    methodType(void.class));

MethodHandle   gwt = MethodHandles.guardWithTest(test1, f1, f2);
while (true) { gwt.invokeExact(); }
```

@ 105 **jsr292.GWT::run1** (7 bytes) inline (hot)

@ 3 java.lang.invoke.LambdaForm$MH000/789451787::invokeExact_MT (13 bytes) inline (hot)

@ 2 java.lang.invoke.Invokers::checkExactType (30 bytes) inline (hot)

@ 11 java.lang.invoke.MethodHandle::type (5 bytes) accessor

@ 9 java.lang.invoke.MethodHandle::invokeBasic()V (0 bytes) **receiver not constant**

Non-Inlined Call

Customized version

155 34 java.lang.invoke.LambdaForm$MH002/1510467688::guard (56 bytes)

@ 5 java.lang.invoke.LambdaForm$DMH005/1581781576::invokeStatic__I (13 bytes) inline (hot)

@ 1 java.lang.invoke.DirectMethodHandle::internalMemberName (8 bytes) inline (hot)

@ 9 **jsr292.GWT::test1** (12 bytes) inline (hot)

@ 31 java.lang.invoke.LambdaForm$DMH005/1581781576::invokeStatic__V (13 bytes) inline (hot)

@ 1 java.lang.invoke.DirectMethodHandle::internalMemberName (8 bytes) inline (hot)

@ 9 **jsr292.GWT::f1** (2 bytes) inline (hot)

@ 52 java.lang.invoke.LambdaForm$DMH005/1581781576::invokeStatic__V (13 bytes) inline (hot)

@ 1 java.lang.invoke.DirectMethodHandle::internalMemberName (8 bytes) inline (hot)

@ 9 **jsr292.GWT::f2** (2 bytes) inline (hot)

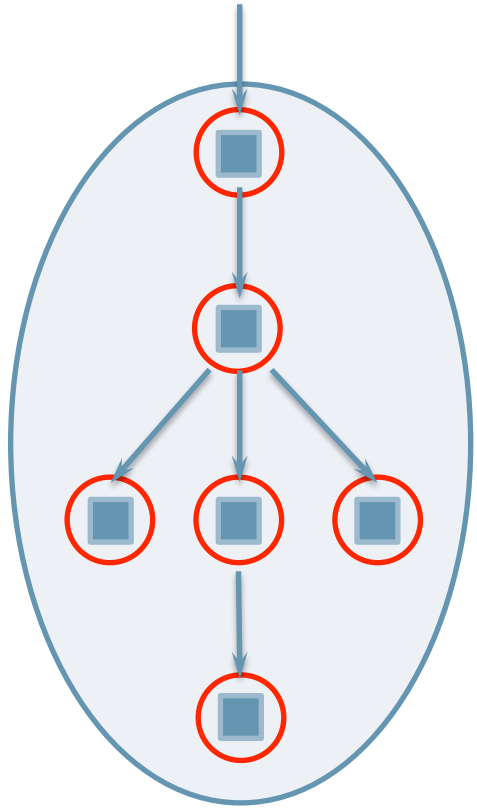
Non-Inlined Call

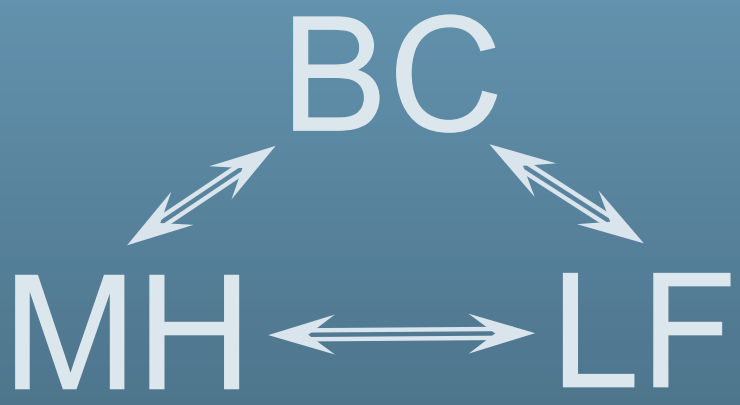
Shared version

```
192   64          java.lang.invoke.LambdaForm$MH/1252585652::guard (68 bytes)
  @ 24   java.lang.invoke.MethodHandle::invokeBasic()I (0 bytes)   receiver not constant
  @ 47   java.lang.invoke.MethodHandle::invokeBasic()V (0 bytes)   receiver not constant
  @ 64   java.lang.invoke.MethodHandle::invokeBasic()V (0 bytes)   receiver not constant
```

Non-Inlined Call

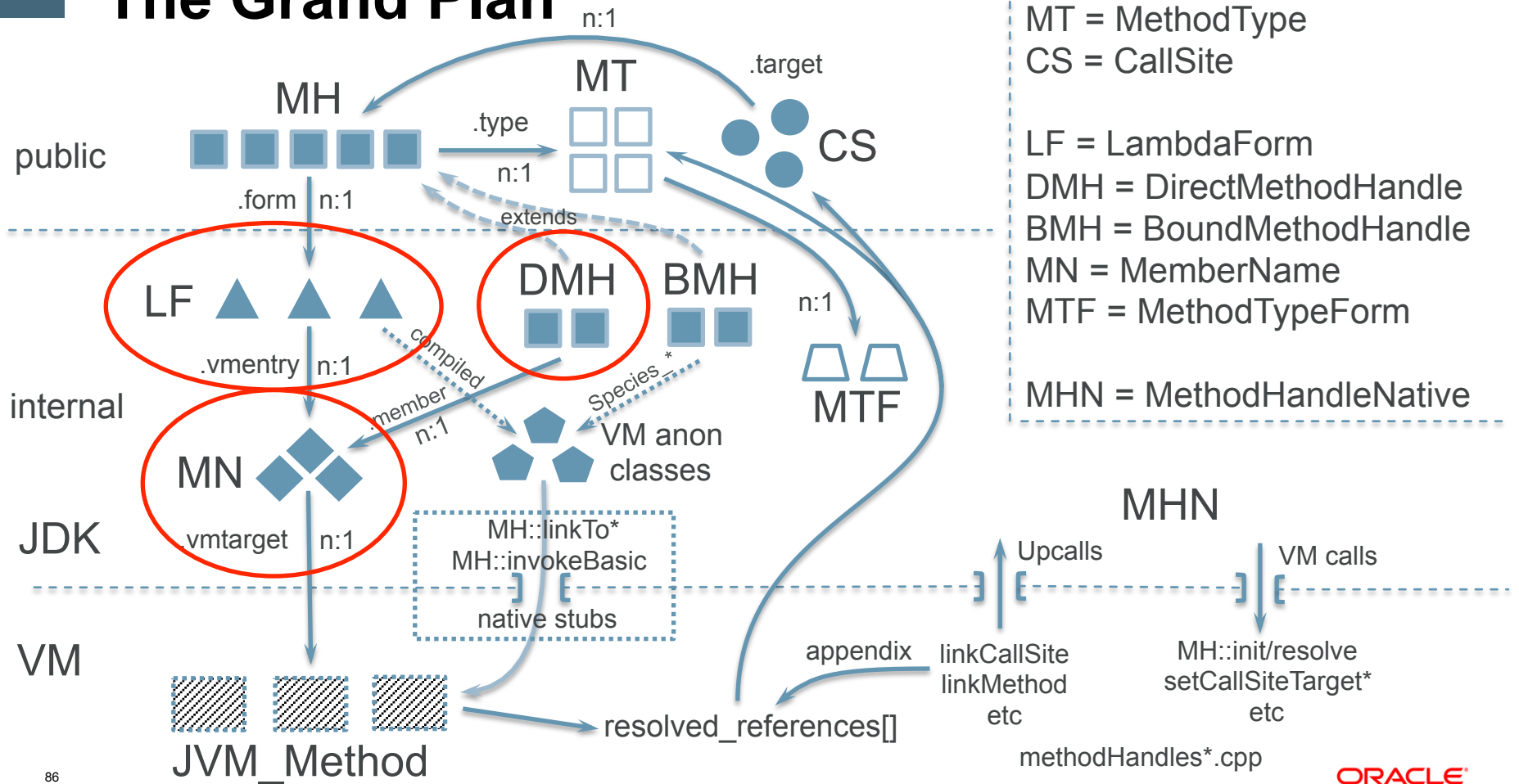
- Customized LambdaForm
  - single compiled method per MH chain / root LF
- Shared LambdaForm
  - compiled method per MH/LF





Interactions

The Grand Plan



ORACLE

Calls between LF, MH, and Bytecode

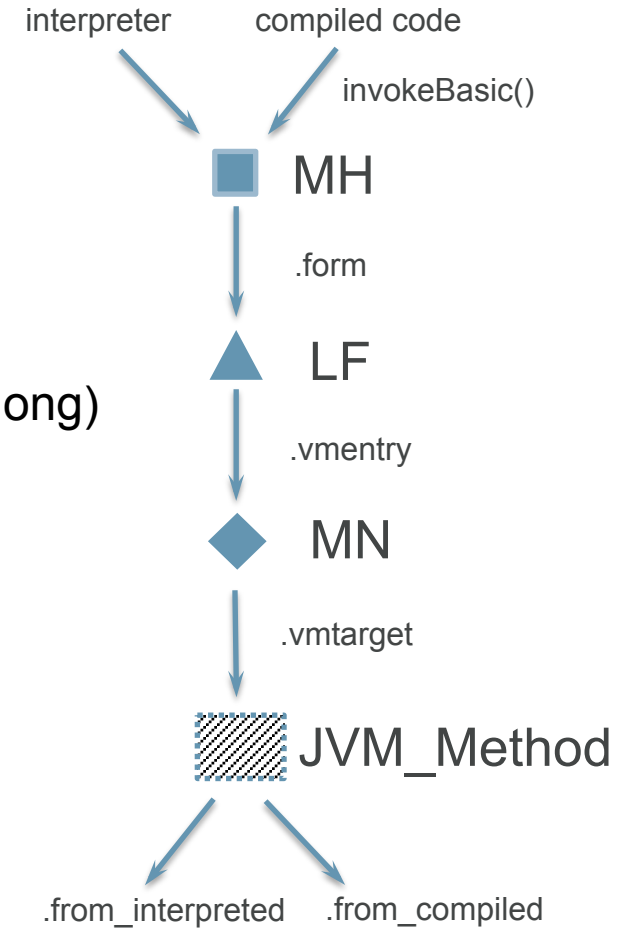
1. MH => LF
   - jump to mh.form
2. LF => MH
   - NamedFunction.resolvedHandle.invokeBasic(…)
3. LF => BC (Java method)
   - DMH “linkers” (MH.linkTo*)
4. BC => MH
   - LF adapters for invokedynamic & “invokehandle”

MH.invokeBasic(…)

1-2. MH <=> LF

- Unchecked version of MH.invokeExact(…)
  - Weakly-typed (basic types)
  - NB! inherently unsafe (e.g. allows Object <=> long)
- Action: MH.form.vmentry.vmtarget

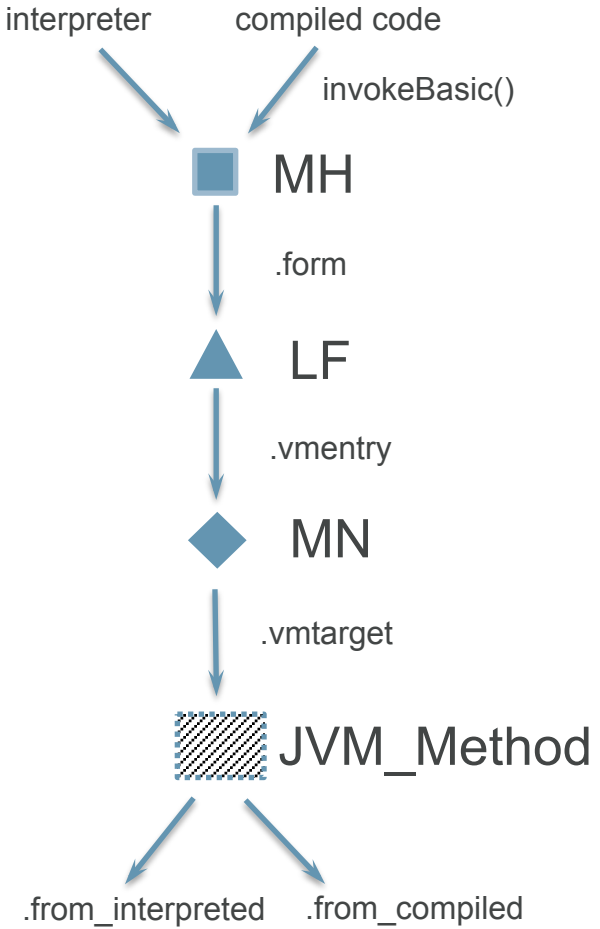
- Stack trace:
  1. MethodHandles::jump_from_method_handle(…)
  2. MethodHandles::jump_to_lambda_form(…)
  3. MethodHandles::generate_method_handle_dispatch(…)
  4. gen_special_dispatch()
  5. SharedRuntime::generate_native_wrapper(…)



MH.invokeBasic(L)L

1-2. MH <=> LF

```
# this:     rsi:rsi   = 'java/lang/invoke/MethodHandle'
# parm0:    rdx:rdx   = 'java/lang/Object'

 <+0>: mov    0x14(%rsi),%ebx     ; (1) MH.form
 <+3>: shl    $0x3,%rbx           ;
 <+7>: mov    0x28(%rbx),%ebx     ; (2) LF.vmentry
<+10>: shl    $0x3,%rbx           ;
<+14>: mov    0x18(%rbx),%rbx     ; (3) MH.vmtarget
<+18>: test   %rbx,%rbx           ;
<+21>: je     <+30>               ; (4) JVM_Method == NULL?
<+27>: jmpq   *0x48(%rbx)         ;
<+30>: jmpq   <throw_AbstractMethodError_stub>
```



ORACLE

Direct Method Handles

3. LF => BC

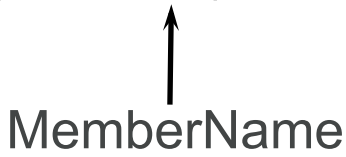
- Capability for using one Java method
  - Or field or constructor
  - Implements CONSTANT_MethodHandle constants
- Carries an internal JVM cookie “MemberName”
- Performs needed checks or conversions
- Has internal weakly-typed jump to its member-name
  - For methods and constructors, uses a “linker intrinsic”
  - For fields (static & instance), uses sun.misc.Unsafe

Direct Method Handles

Example: invokestatic

MT: (Object)Object

DMH.invokeStatic_000_L_L=Lambda(a0:L,a1:L)=>{
    t2:L=DirectMethodHandle.internalMemberName(a0:L);
    t3:L=MethodHandle.**linkToStatic**(a1:L,t2:L); t3:L}



Direct Method Handle “Linkers”

3. LF => BC

- Weakly-typed invocation of arbitrary member-names
  - MH::linkToStatic, MH::linkToVirtual, MH::linkToInterface, MH::linkToSpecial
- Oddity: The member-name is the trailing argument
  - Forces caller to perform argument shuffling
  - Trailing argument can be used and transparently dropped
  - Enables compiled fast paths w/o special JVM handling

MH.linkTo*(…, MemberName)

- Invokes w/ arguments a method described by MemberName
- Action:
  - pop trailing argument (MemberName)
  - load Method* from MemberName.vmtarget
  - perform method selection

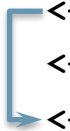
- Stack trace:
  1. MethodHandles::generate_method_handle_dispatch(…)
  2. gen_special_dispatch()
  3. SharedRuntime::generate_native_wrapper(…)

MH.linkToStatic(ILLL)L

```
# parm0:    rsi       = int
# parm1:    rdx:rdx   = 'java/lang/Object'
# parm2:    rcx:rcx   = 'java/lang/Object'
# parm3:    r8:r8     = 'java/lang/invoke/MemberName'

<+0>:  mov    0x18(%r8),%rbx
<+4>:  test   %rbx,%rbx
<+7>:  je     <+16>
<+13>: jmpq   *0x48(%rbx)
<+16>: jmpq   <throw_AbstractMethodError_stub>
```

MH.linkToSpecial(LL)I

```
# parm0:    rsi:rsi   = 'java/lang/Object'
# parm1:    rdx:rdx   = 'java/lang/invoke/MemberName'

 <+0>: cmp    (%rsi),%rax
 <+3>: mov    0x18(%rdx),%rbx
 <+7>: test   %rbx,%rbx
<+10>: je     <+19>
<+16>: jmpq   *0x48(%rbx)
<+19>: jmpq   <throw_AbstractMethodError_stub>
```



MH.linkToVirtual(LLIL)L

```
# parm0:    rsi:rsi   = 'java/lang/Object'
# parm1:    rdx:rdx   = 'java/lang/Object'
# parm2:    rcx       = int
# parm3:    r8:r8     = 'java/lang/invoke/MemberName'

 <+0>: mov    0x8(%rsi),%r10d
 <+4>: shl    $0x3,%r10
 <+8>: mov    0x10(%r8),%r11
<+12>: mov    0x1c8(%r10,%r11,8),%rbx
<+20>: test   %rbx,%rbx
<+23>: je     <+32>
<+29>: jmpq   *0x48(%rbx)
<+32>: jmpq   <throw_AbstractMethodError_stub>
```

MH.linkToInterface(LL)L

```
# parm0:    rsi:rsi   = 'Object'
# parm1:    rdx:rdx   = 'MemberName'

 <+0>: mov    0x8(%rsi),%r10d
 <+4>: shl    $0x3,%r10
 <+8>: mov    0x20(%rdx),%eax
<+10>: shl    $0x3,%rax
<+15>: mov    0x48(%rax),%rax
<+19>: mov    0x10(%rdx),%rbx
<+23>: mov    0x128(%r10),%r11d
<+30>: lea    0x1c8(%r10,%r11,8),%r11
<+38>: lea    (%r10,%rbx,8),%r10
<+42>: mov    (%r11),%rbx
<+45>: cmp    %rbx,%rax
<+48>: je     <+71>
```

```
<+50>: 0x...f12: test   %rbx,%rbx
<+53>: 0x...f15: je     <+96>
<+59>: 0x...f1b: add    $0x10,%r11
<+63>: 0x...f1f: mov    (%r11),%rbx
<+66>: 0x...f22: cmp    %rbx,%rax
<+69>: 0x...f25: jne    <+50>
<+71>: 0x...f27: mov    0x8(%r11),%r11d
<+75>: 0x...f2b: mov    (%r10,%r11,1),%rbx
<+79>: 0x...f2f: test   %rbx,%rbx
<+82>: 0x...f32: je     <+91>
<+88>: 0x...f38: jmpq   *0x48(%rbx)
<+91>: 0x...f3b: jmpq   <throw_AME_stub>
<+96>: 0x...f40: jmpq   <throw_ICCE_stub>
```

Bytecode Call Sites

4. BC => MH

- Two kinds:
  - invokedynamic
    - bytecode instruction
  - “invokehandle”
    - MH.invokeExact(), MH.invoke()

Bytecode Call Sites

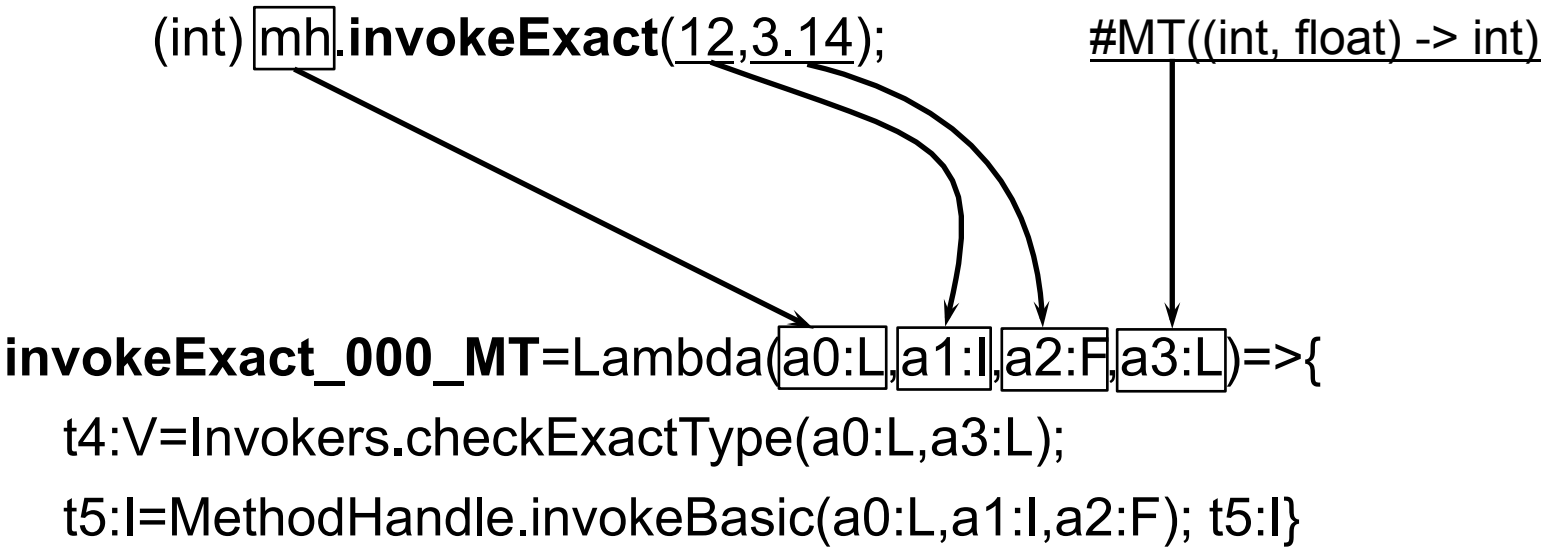
Context

- Linked contextual argument
  - invokedynamic: linked CallSite instance
    - Invocation must insert and invoke the call site target.
  - invokehandle: resolved MethodType value
    - Invocation must reify the MT enough to check it

- Formalized in the JVM via **“appendix”** args
  - Linking indy or MH.invoke makes an up-call to the JDK
  - JDK computes a LF and appendix argument
  - JVM records both and uses them for all calls

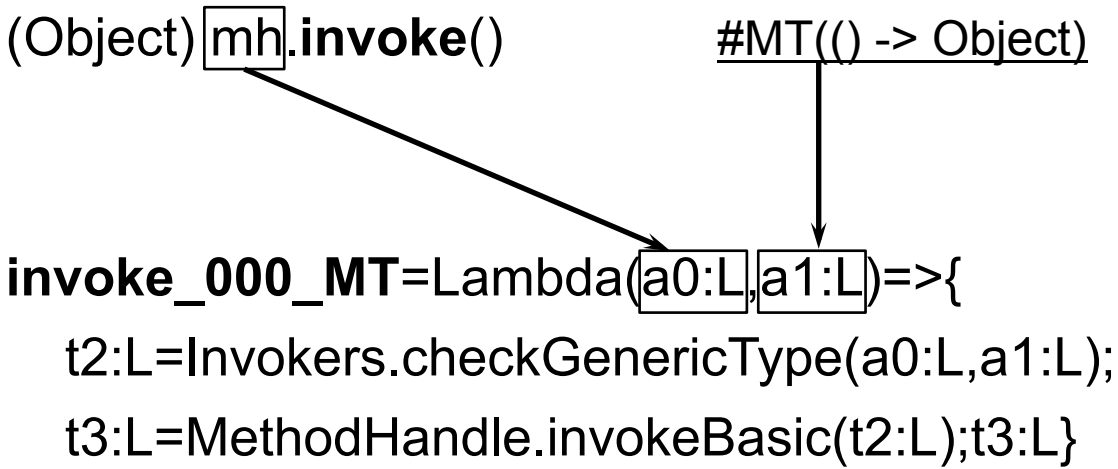
Linking “invokehandle”

Example: MH.invokeExact



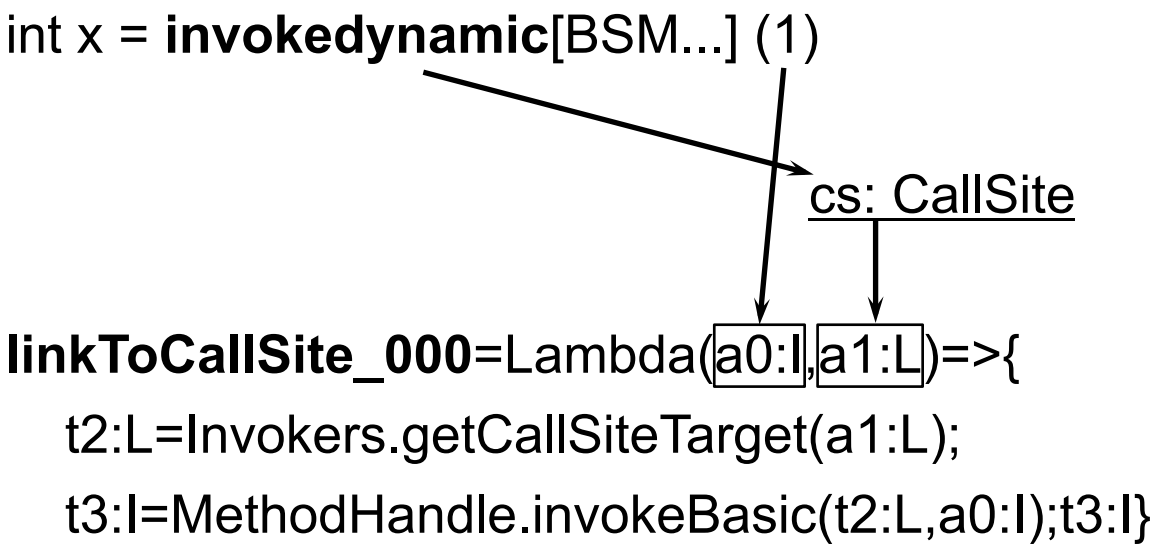
Linking “invokehandle”

Example: MH.invoke



Linking invokedynamic

Example: invokedynamic



Linking invokedynamic

Single CP entry, multiple call sites

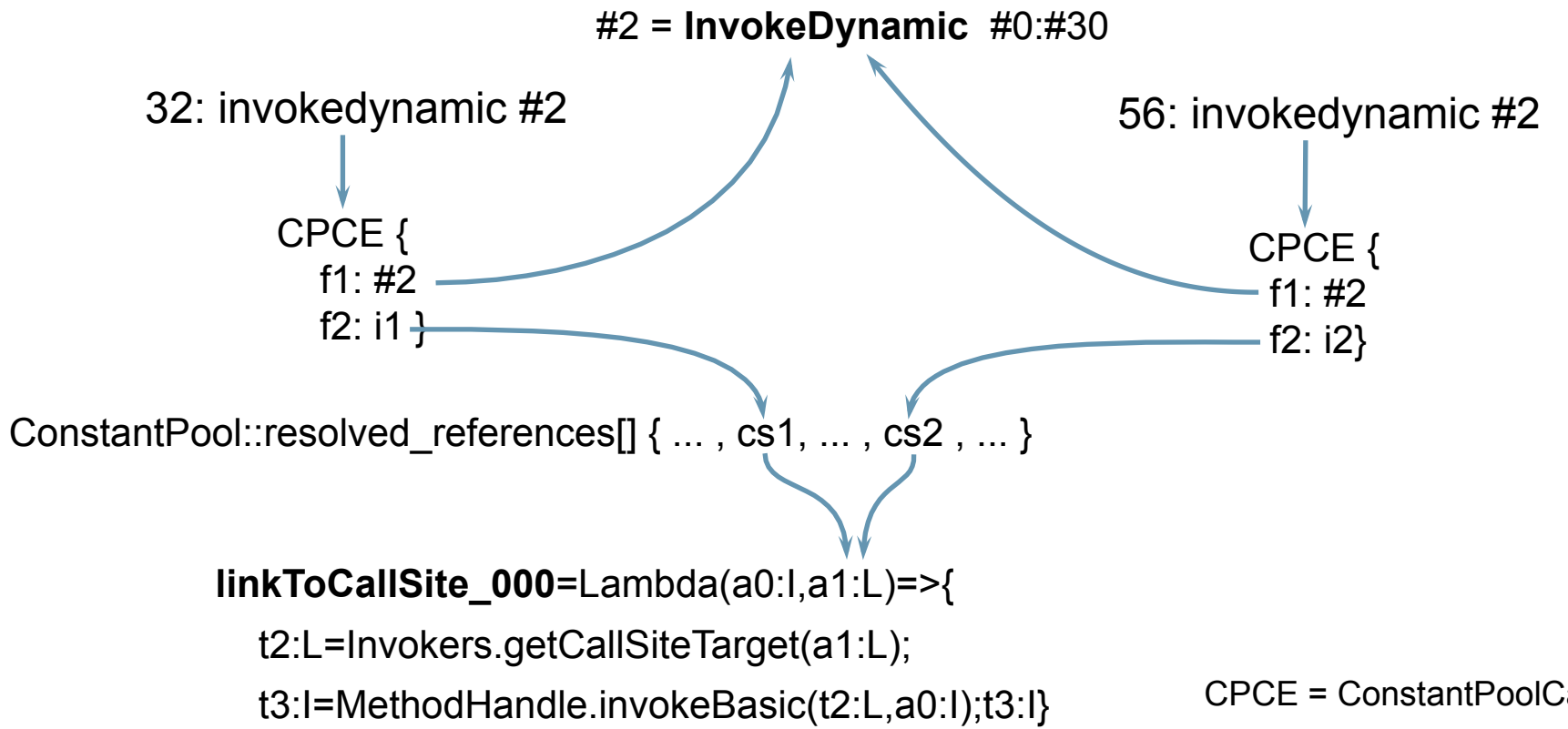
- Problem:

  multiple indy call sites can point to the same CONSTANT_InvokeDynamic

- How to distinguish them and feed different appendix arguments?

Linking invokedynamic

Single CP entry, multiple call sites

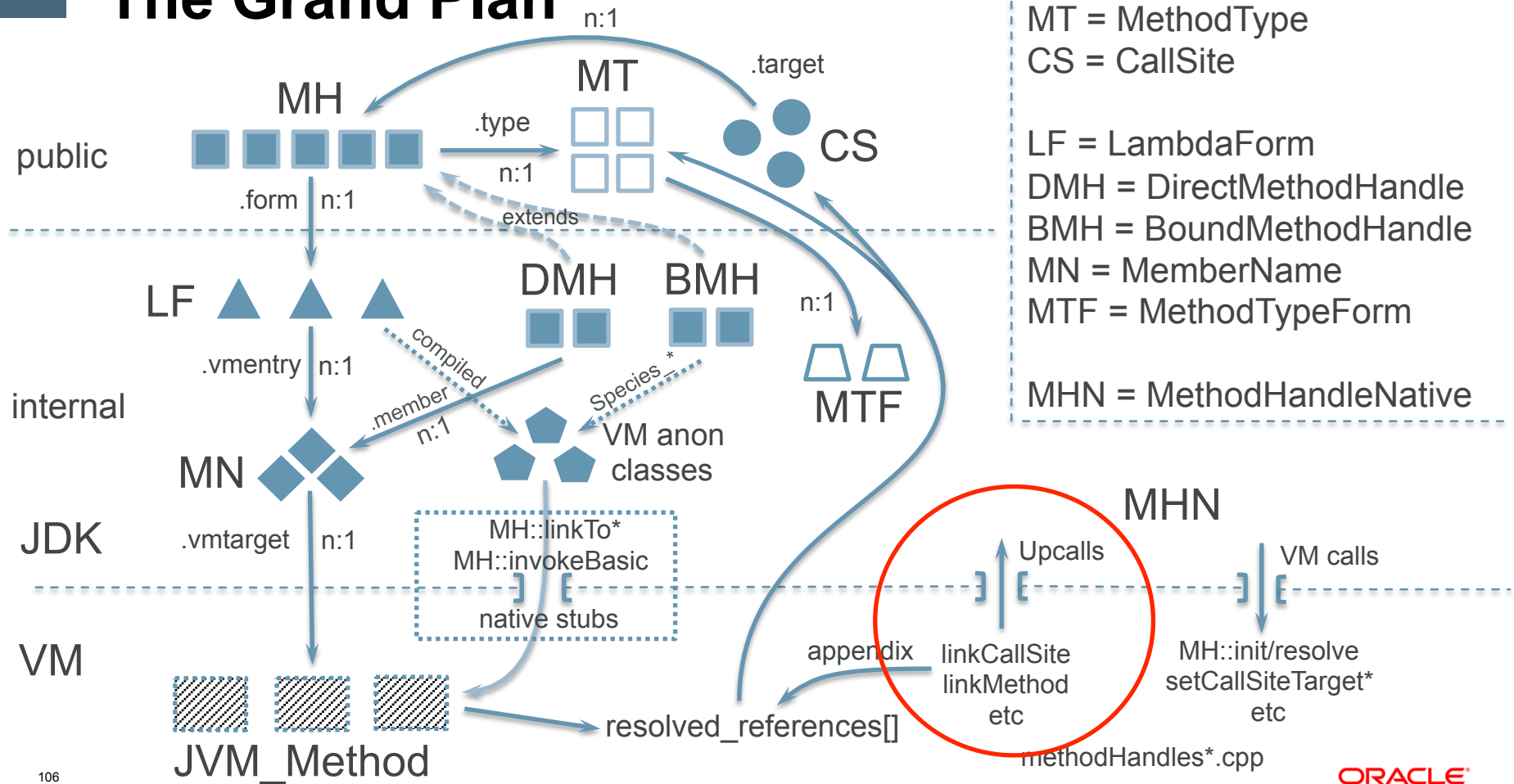


CPCE = ConstantPoolCacheEntry

Upcalls

VM => Java

The Grand Plan



ORACLE

Upcalls

VM => JDK

- MHN.linkCallSite
  - indy call site linkage
- MHN.linkMethod
  - MH.invoke/.invokeExact linkage
- MHN.linkMethodHandleConstant
  - CONSTANT_MethodHandle resolution
- MHN.findMethodHandleType
  - CONSTANT_MethodType resolution

MHN.linkCallSite

Stack trace:

1. MH.invoke (on BSM)
2. CallSite.makeSite
3. MHN.linkCallSiteImpl
4. **MHN.linkCallSite**
5. SystemDictionary::find_dynamic_call_site_invoker
6. LinkResolver::resolve_dynamic_call
7. LinkResolver::resolve_invokedynamic

MHN.linkMethod

Stack trace:

1. Invokers.invokeHandleForm (cf. MethodHandles.exactInvoker)
2. Invokers.methodHandleInvokeLinkerMethod
3. MHN.linkMethodImpl
4. **MHN.linkMethod**
5. find_method_handle_invoker
6. LinkResolver::lookup_polymorphic_method
7. LinkResolver::resolve_invokehandle / LinkResolver::resolve_method

MHN.linkMethodHandleConstant

Stack trace:

1. MHs.getDirectMethodCommon (cf. Lookup.findVirtual)
2. MHs.getDirectMethodNoSecurityManager
3. MHs.getDirectMethodForConstant
4. Lookup.linkMethodHandleConstant
5. **MHN.linkMethodHandleConstant**
6. SystemDictionary::link_method_handle_constant
7. ConstantPool::resolve_constant_at_impl
8. ConstantPool::resolve_constant_at

MHN.findMethodHandleType

Stack trace:

1. MethodType.makeImpl (cf. MethodType.methodType)
2. **MHN.findMethodHandleType**
3. SystemDictionary::find_method_handle_type
4. ConstantPool::resolve_constant_at_impl
5. ConstantPool::resolve_constant_at

Annotations

Annotation	Description	Examples
@LF.Hidden	omit the frame in stack traces	LambdaFormInterpreter (LFI.invoke_*)
@LF.Compiled	mark bytecode-compiled LambdaForm	j.l.i.LambdaForm$MH::*, …$DMH::*, ...$BMH::*, …
@ForceInline	always inline (for JIT)	MHI.castReference
@DontInline	never inline (for JIT)	LFI, Invokers.maybeCustomize
@Stable	mark effectively final field or array elements (for JIT)	LF.names[], LF.Name.arguments[], NamedFunction.resolvedHandle

Useful Diagnostic Options

- -XX:+ShowHiddenFrames (diagnostic)
- -XX:+PrintMethodHandleStubs (diagnostic)
- -XX:+TraceMethodHandles (develop)
- -XX:+TraceInvokeDynamic (develop)
- -Djava.lang.invoke.MethodHandle.DUMP_CLASS_FILES=true
- -Djava.lang.invoke.MethodHandle.TRACE_METHOD_LINKAGE=true

Materials

- Notes: http://cr.openjdk.java.net/~vlivanov/talks/2015-Indy_Deep_Dive.pdf
- https://wiki.openjdk.java.net/display/HotSpot/Method+handles+and+invokedynamic
- https://wiki.openjdk.java.net/display/HotSpot/Bound+method+handles
- https://wiki.openjdk.java.net/display/HotSpot/Direct+method+handles
- https://wiki.openjdk.java.net/display/HotSpot/Method+handle+invocation
- "Deconstructing MethodHandles" by Paul Sandoz
  - https://wiki.openjdk.java.net/display/HotSpot/Deconstructing+MethodHandles
- "Lambda Forms" by John Rose, JVMLS'12
  - http://cr.openjdk.java.net/~jrose/pres/201207-LF-Tutorial.pdf
- "J9's MethodHandle Compilation Pipeline" by Dan Heidinga, Jfocus VM Summit'15
  - http://www.jfokus.se/jfokus15/preso/J9%20MethodHandle%20Compilation%20Pipeline.pdf
- HotSpot: methodHandles*.hpp/.cpp
- JVM Specification 8: http://docs.oracle.com/javase/specs/jvms/se8/html/index.html

Safe Harbor Statement

The preceding 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.

MAKE THE FUTURE JAVA

ORACLE