





Introducing Java 7

MOVING JAVA FORWARD



ORACLE®

John R. Rose, Da Vinci Machine Project, JSR 292 Lead



Introducing Java 7

MOVING JAVA FORWARD



ORACLE®

Method Handles and Beyond... Some basis vectors

John R. Rose, Da Vinci Machine Project, JSR 292 Lead

Victory in JDK 7!



Victory in JDK 7!

JSR 292 is finished.



Victory in JDK 7!

JSR 292 is finished.



(Or is it?)



Victory in JDK 7!

JSR 292 is finished.



(Or is it?)



Back to the future!

...a perspective from the original JVM Specification

Back to the future!

...a perspective from the original JVM Specification

The Java Virtual Machine knows nothing about the Java programming language, only of a particular binary format, the class file format.

A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.

Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine.

Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages.

In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.



Overview...

- What we did in JDK 7
- How it works in Hotspot
- Advice to users
- Next steps in JSR 292
- Building the future



Overview...

- What we did in JDK 7
- How it works in Hotspot
- Advice to users
- Next steps in JSR 292
- Building the future



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

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

Sun Microsystems, JAOO 2005

<http://blogs.sun.com/roller/resources/gbracha/JAOO2005.pdf>

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

Sun Microsystems, JAOO 2005

<http://blogs.sun.com/roller/resources/gbracha/JAOO2005.pdf>

JSR 292 Timeline

JSR 292 Timeline

- 2005: Initial design sketch

JSR 292 Timeline

- 2005: Initial design sketch
- 2006: JSR 292 Expert Group formed

JSR 292 Timeline

- 2005: Initial design sketch
- 2006: JSR 292 Expert Group formed
- 2007: EG reboot

JSR 292 Timeline

- 2005: Initial design sketch
- 2006: JSR 292 Expert Group formed
- 2007: EG reboot
- 2008: API with Method Handles (Early Draft Review)



JSR 292 Timeline

- 2005: Initial design sketch
- 2006: JSR 292 Expert Group formed
- 2007: EG reboot
- 2008: API with Method Handles (Early Draft Review)
- 2009: API refinement (e.g., CONSTANT_MethodHandle)



JSR 292 Timeline

- 2005: Initial design sketch
- 2006: JSR 292 Expert Group formed
- 2007: EG reboot
- 2008: API with Method Handles (Early Draft Review)
- 2009: API refinement (e.g., CONSTANT_MethodHandle)
- 2010: API refinement (e.g., BootstrapMethods)



JSR 292 Timeline

- 2005: Initial design sketch
- 2006: JSR 292 Expert Group formed
- 2007: EG reboot
- 2008: API with Method Handles (Early Draft Review)
- 2009: API refinement (e.g., CONSTANT_MethodHandle)
- 2010: API refinement (e.g., BootstrapMethods)
- 2011: Final balloting.



Key Features

Key Features

- New bytecode instruction: *invokedynamic*.



Key Features

- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.

Key Features

- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.
 - User-visible object: `java.lang.invoke.CallSite`

Key Features

- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.
 - User-visible object: `java.lang.invoke.CallSite`
 - Dynamic call sites can be linked and relinked, dynamically.

Key Features

- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.
 - User-visible object: `java.lang.invoke.CallSite`
 - Dynamic call sites can be linked and relinked, dynamically.
- New unit of behavior: *method handle*



Key Features

- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.
 - User-visible object: `java.lang.invoke.CallSite`
 - Dynamic call sites can be linked and relinked, dynamically.
- New unit of behavior: *method handle*
 - The content of a dynamic call site is a method handle.



Key Features

- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.
 - User-visible object: `java.lang.invoke.CallSite`
 - Dynamic call sites can be linked and relinked, dynamically.
- New unit of behavior: *method handle*
 - The content of a dynamic call site is a method handle.
 - Method handles are function pointers for the JVM.



Key Features

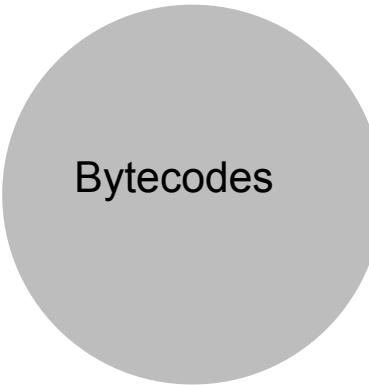
- New bytecode instruction: *invokedynamic*.
 - Linked reflectively, under user control.
 - User-visible object: `java.lang.invoke.CallSite`
 - Dynamic call sites can be linked and relinked, dynamically.
- New unit of behavior: *method handle*
 - The content of a dynamic call site is a method handle.
 - Method handles are function pointers for the JVM.
 - (Or if you like, each MH implements a single-method interface.)



Dynamic program composition

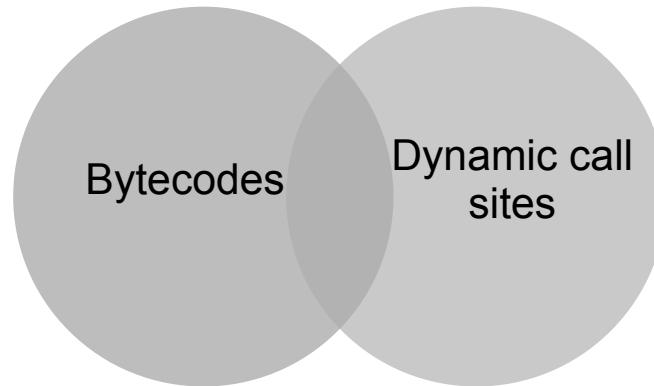
Dynamic program composition

- Bytecodes are created by Java compilers or dynamic runtimes.



Dynamic program composition

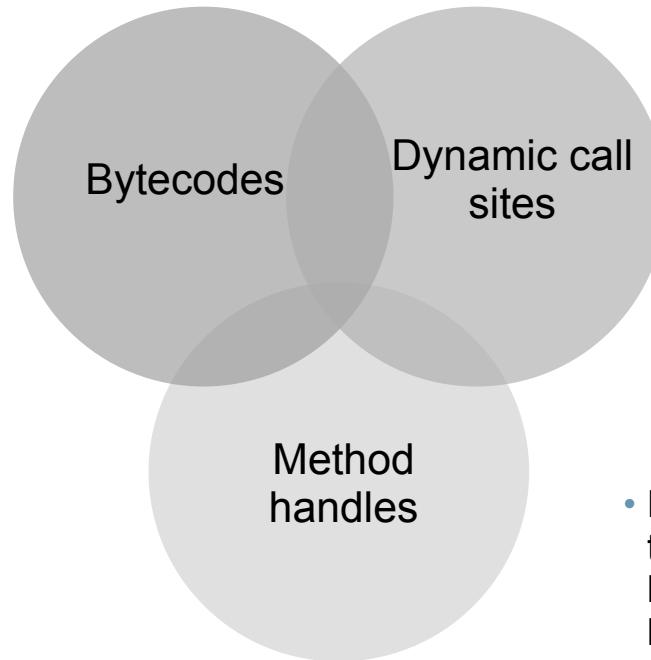
- Bytecodes are created by Java compilers or dynamic runtimes.



- A dynamic call site is created for each invokedynamic bytecode.

Dynamic program composition

- Bytecodes are created by Java compilers or dynamic runtimes.



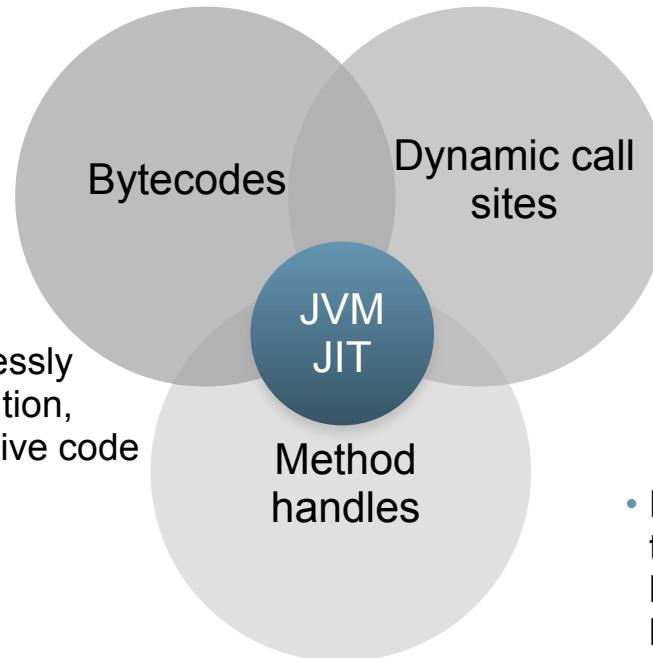
- A dynamic call site is created for each invokedynamic bytecode.

- Each call site is bound to one or more method handles, which point back to bytecoded methods.



Dynamic program composition

- Bytecodes are created by Java compilers or dynamic runtimes.
- The JVM seamlessly integrates execution, optimizing to native code as necessary.



- A dynamic call site is created for each invokedynamic bytecode.
- Each call site is bound to one or more method handles, which point back to bytecoded methods.



What's in a method call? (*before invokedynamic*)

	Source code	Bytecode	Linking	Executing
Naming	Identifiers	Utf8 constants	JVM “dictionary”	
Selecting	Scopes	Class names	Loaded classes	V-table lookup
Adapting	Argument conversion		C2I / I2C adapters	Receiver narrowing
Calling				<i>Jump with arguments</i>



What's in a method call? (*using invokedynamic*)

	Source code	Bytecode	Linking	Executing
Naming	∞	∞	∞	∞
Selecting	∞	Bootstrap methods	Bootstrap method call	∞
Adapting	∞		Method handles	∞
Calling				<i>Jump with arguments</i>



Coding directly with method handles

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;  
import static java.lang.invoke.MethodType.*;
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;  
import static java.lang.invoke.MethodType.*;  
...
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();
MethodHandle HASHCODE = LOOKUP
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();
MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();
MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
    "identityHashCode", methodType(int.class, Object.class));
```

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
```



Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));}
```



Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));

MethodHandle CONCAT = LOOKUP
```



Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));

MethodHandle CONCAT = LOOKUP
    .findVirtual(String.class,
```



Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));

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

Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));

MethodHandle CONCAT = LOOKUP
    .findVirtual(String.class,
        "concat", methodType(String.class, String.class));
assertEquals("xy", (String) CONCAT.invokeExact("x", "y"));}
```



Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));

MethodHandle CONCAT = LOOKUP
    .findVirtual(String.class,
        "concat", methodType(String.class, String.class));
assertEquals("xy", (String) CONCAT.invokeExact("x", "y"));

MethodHandle CONCAT_FU = CONCAT.bindTo("fu");
```



Coding directly with method handles

```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
assertEquals(System.identityHashCode("xy"),
    (int) HASHCODE.invoke("xy"));

MethodHandle CONCAT = LOOKUP
    .findVirtual(String.class,
        "concat", methodType(String.class, String.class));
assertEquals("xy", (String) CONCAT.invokeExact("x", "y"));

MethodHandle CONCAT_FU = CONCAT.bindTo("fu");
{assertEquals("futbol", CONCAT_FU.invoke("tbol"))};
```



Pseudo-coding with invokedynamic

[http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/
tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java](http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java)

Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
```

Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}
```

Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}

INDY_baz.CALL_SITE.invokeExact("baz arg", 2, 3.14);
```



Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}

INDY_baz.CALL_SITE.invokeExact("baz arg", 2, 3.14);
static private class INDY_baz {
    static final MethodHandle CALL_SITE; static {
```



Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}
```

```
INDY_baz.CALL_SITE.invokeExact("baz arg", 2, 3.14);
static private class INDY_baz {
    static final MethodHandle CALL_SITE; static {
        MethodType mt = methodType(void.class,
            String.class, int.class, double.class);
```

Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}

INDY_baz.CALL_SITE.invokeExact("baz arg", 2, 3.14);
static private class INDY_baz {
    static final MethodHandle CALL_SITE; static {
        MethodType mt = methodType(void.class,
            String.class, int.class, double.class);
        CallSite cs = (CallSite) MH_bsm2().invoke(
            lookup(), "baz", mt // standard BSM arguments
```



Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}

INDY_baz.CALL_SITE.invokeExact("baz arg", 2, 3.14);
static private class INDY_baz {
    static final MethodHandle CALL_SITE; static {
        MethodType mt = methodType(void.class,
            String.class, int.class, double.class);
        CallSite cs = (CallSite) MH_bsm2().invoke(
            lookup(), "baz", mt // standard BSM arguments
            , 1234.5);           // optional static argument
    }
}
```



Pseudo-coding with invokedynamic

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/test/java/lang/invoke/InvokeDynamicPrintArgs.java>

invokedynamic baz(Ljava/lang/String;ID)V [#bsm2, 1234.5]

```
static CallSite bsm2(
    Lookup caller, String name, MethodType type, Object... arg) {
    // ignore caller and name, but match the type:
    return new PrintingCallSite(caller, name, type, arg);
}

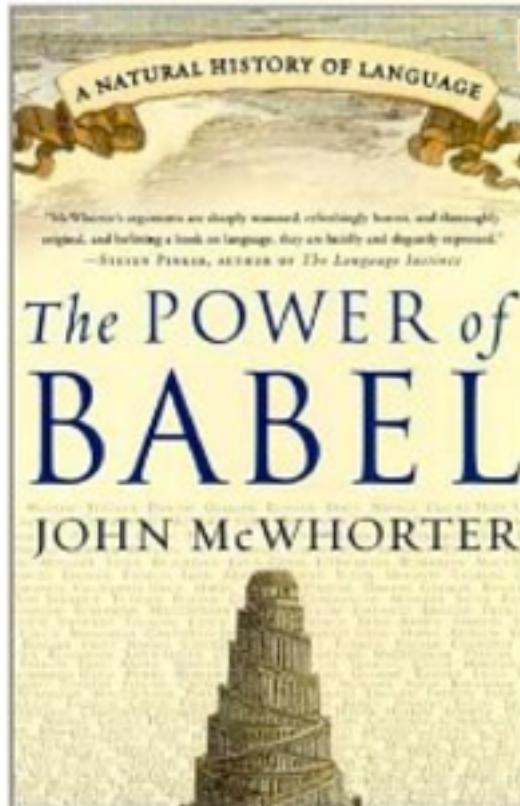
INDY_baz.CALL_SITE.invokeExact("baz arg", 2, 3.14);
static private class INDY_baz {
    static final MethodHandle CALL_SITE; static {
        MethodType mt = methodType(void.class,
            String.class, int.class, double.class);
        CallSite cs = (CallSite) MH_bsm2().invoke(
            lookup(), "baz", mt // standard BSM arguments
            , 1234.5);           // optional static argument
        CALL_SITE = cs.dynamicInvoker();
    }
}
```



| © 2011 Oracle Corporation

Bonus book plug!

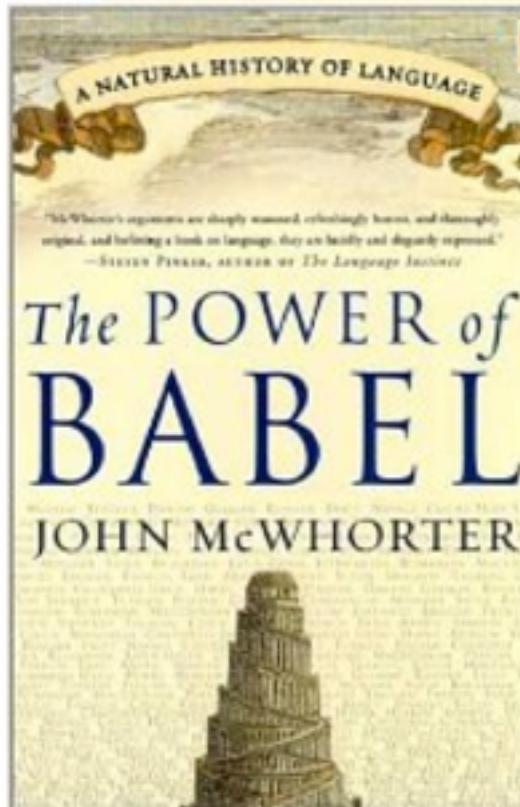
[Click to LOOK INSIDE!](#)



Bonus book plug!

[Click to LOOK INSIDE!](#)

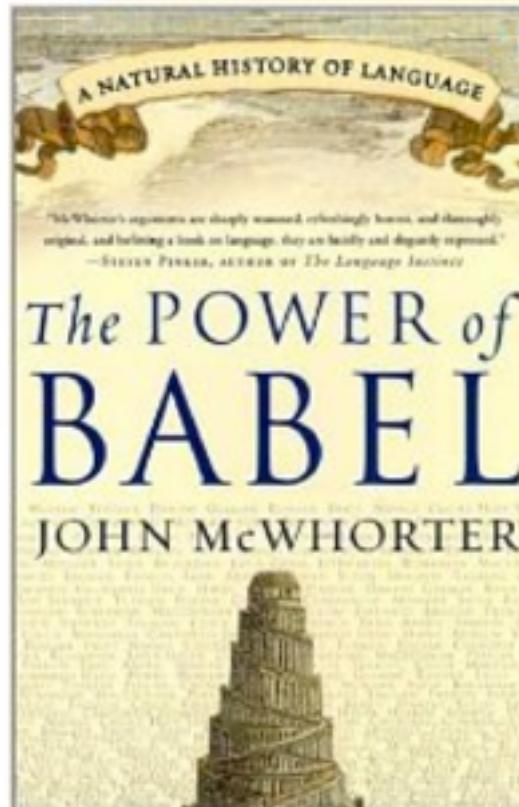
Interesting
account of *human*
language change.



Bonus book plug!

[Click to LOOK INSIDE!](#)

Interesting account of *human* language change.

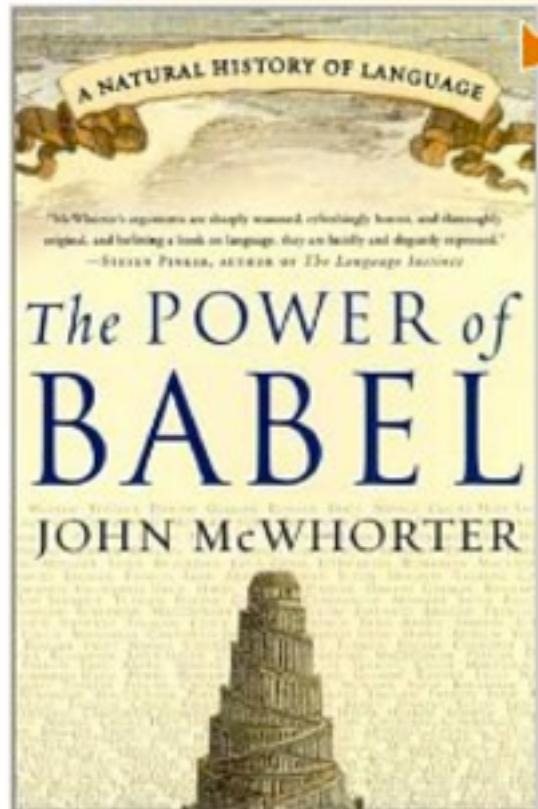


Bonus book plug!

Interesting account of *human* language change.

Makes computer language change look easy.

[Click to LOOK INSIDE!](#)

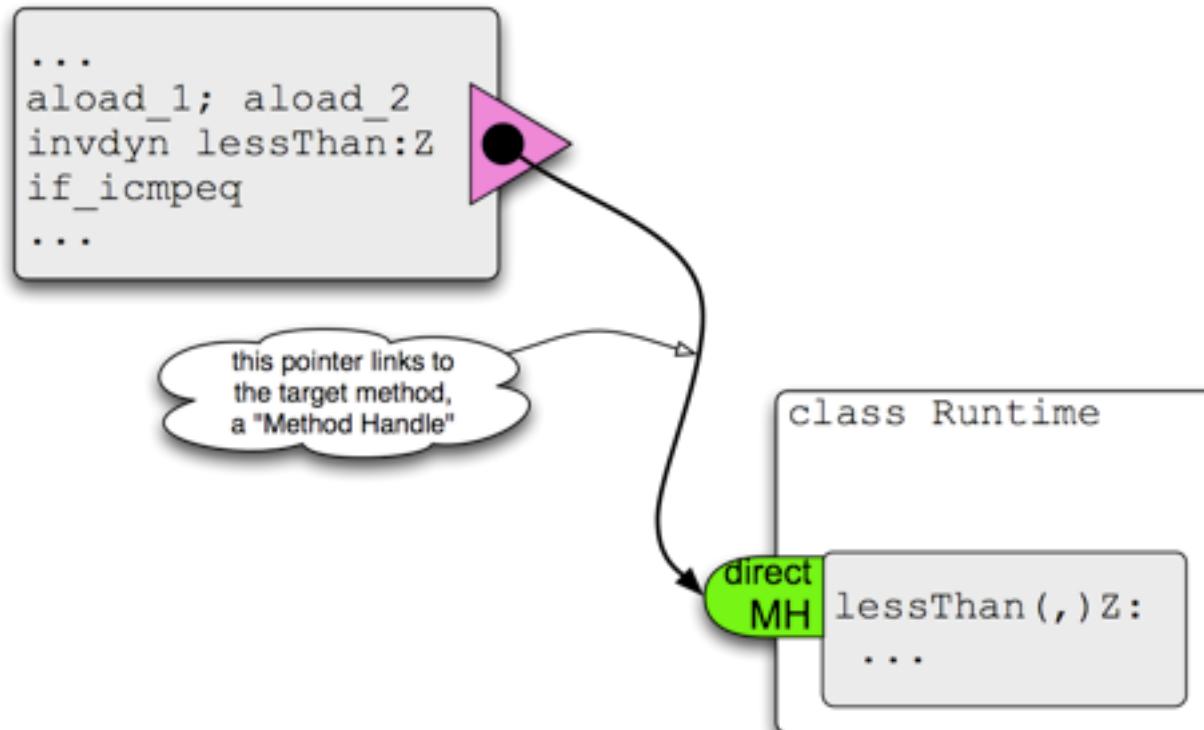


Overview...

- What we did in JDK 7
- How it works in Hotspot
- Advice to users
- Next steps in JSR 292
- Building the future



Invokedynamic “plumbing”, take 1



Invokedynamic pseudo-code

Invokedynamic pseudo-code

```
Object a1 = ..., a2 = ...;
```

Invokedynamic pseudo-code

```
Object a1 = ..., a2 = ...;  
CallSite cs = INDY_lessThan.CALL_SITE; // linked as a constant
```

Invokedynamic pseudo-code

```
Object a1 = ..., a2 = ...;  
CallSite cs = INDY_lessThan.CALL_SITE; // linked as a constant  
MethodHandle mh = cs.getTarget(); // can be variable
```

Invokedynamic pseudo-code

```
Object a1 = ..., a2 = ...;  
  
CallSite cs = INDY_lessThan.CALL_SITE; // linked as a constant  
  
MethodHandle mh = cs.getTarget(); // can be variable  
  
boolean z = (boolean) mh.invokeExact(a1, a2);
```

Invokedynamic pseudo-code

```
Object a1 = ..., a2 = ...;  
  
CallSite cs = INDY_lessThan.CALL_SITE; // linked as a constant  
  
MethodHandle mh = cs.getTarget(); // can be variable  
  
boolean z = (boolean) mh.invokeExact(a1, a2);  
  
if (z) { ... }
```

Method handle invocation pseudo-code

Method handle invocation pseudo-code

```
MethodHandle mh = ...;  
// (boolean) mh.invokeExact(a1, a2);
```

Method handle invocation pseudo-code

```
MethodHandle mh = ...;
// (boolean) mh.invokeExact(a1, a2);

MethodType mt = methodType(boolean.class,
    Object.class, Object.class);      // internal constant
if (mh.type() != mt) throw new WrongMethodTypeException();
```



Method handle invocation pseudo-code

```
MethodHandle mh = ...;
// (boolean) mh.invokeExact(a1, a2);

MethodType mt = methodType(boolean.class,
    Object.class, Object.class);      // internal constant
if (mh.type() != mt) throw new WrongMethodTypeException();

goto (*mh.vmentry)    // jump indirect to invokestatic_mh
```

Method handle invocation pseudo-code

```
MethodHandle mh = ...;
// (boolean) mh.invokeExact(a1, a2);

MethodType mt = methodType(boolean.class,
    Object.class, Object.class);      // internal constant
if (mh.type() != mt) throw new WrongMethodTypeException();

goto (*mh.vmentry)    // jump indirect to invokestatic_mh

invokestatic_mh:      // vmentry for direct MH to a static method
```



Method handle invocation pseudo-code

```
MethodHandle mh = ...;
// (boolean) mh.invokeExact(a1, a2);

MethodType mt = methodType(boolean.class,
    Object.class, Object.class);      // internal constant
if (mh.type() != mt) throw new WrongMethodTypeException();

goto (*mh.vmentry)    // jump indirect to invokestatic_mh

invokestatic_mh:        // vmentry for direct MH to a static method
addr = mh.vmtarget      // Runtime#lessThan
```



Method handle invocation pseudo-code

```
MethodHandle mh = ...;
// (boolean) mh.invokeExact(a1, a2);

MethodType mt = methodType(boolean.class,
    Object.class, Object.class);      // internal constant
if (mh.type() != mt) throw new WrongMethodTypeException();

goto (*mh.vmentry)    // jump indirect to invokestatic_mh

invokestatic_mh:        // vmentry for direct MH to a static method
addr = mh.vmtarget      // Runtime#lessThan

goto (*addr.from_interpreted_entry) // entry point of #lessThan
```



Method handle invocation pseudo-code

```
MethodHandle mh = ...;
// (boolean) mh.invokeExact(a1, a2);

MethodType mt = methodType(boolean.class,
    Object.class, Object.class);      // internal constant
if (mh.type() != mt) throw new WrongMethodTypeException();

goto (*mh.vmentry)    // jump indirect to invokestatic_mh

invokestatic_mh:        // vmentry for direct MH to a static method
addr = mh.vmtarget    // Runtime#lessThan

goto (*addr.from_interpreted_entry) // entry point of #lessThan

static boolean lessThan(Object x, Object y) { ... }
```



More details about method handles

More details about method handles

- A *direct method handle* points to a Java method.
 - A DMH can emulate any of the pre-existing invoke instructions.



More details about method handles

- A *direct method handle* points to a Java method.
 - A DMH can emulate any of the pre-existing invoke instructions.
- A *bound method handle* includes an saved argument.
 - The bound argument is specified on creation, and is used on call.
 - The bound argument is inserted into the argument list.
 - Any MH can be be bound, and the binding is invisible to callers.

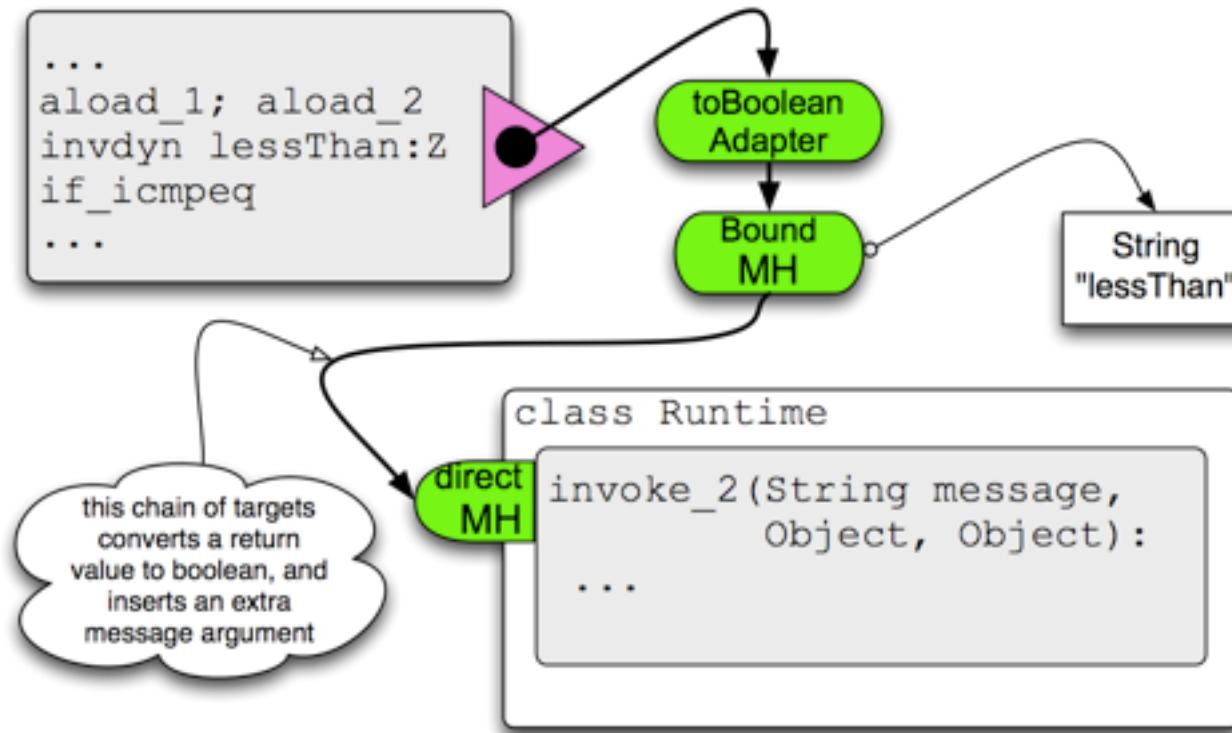


More details about method handles

- A *direct method handle* points to a Java method.
 - A DMH can emulate any of the pre-existing invoke instructions.
- A *bound method handle* includes an saved argument.
 - The bound argument is specified on creation, and is used on call.
 - The bound argument is inserted into the argument list.
 - Any MH can be be bound, and the binding is invisible to callers.
- An *adapter method handle* adjusts values on the fly.
 - Both argument and return values can be adjusted.
 - Adaptations include cast, box/unbox, collect/spread, filter, etc.
 - Any MH can be adapted. Adaptation is invisible to callers.



Invokedynamic “plumbing”, take 2



Method handle invocation pseudo-code (take 2)

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
```

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle  
goto (*mh1.vmentry) // jump indirect to adapter_collect_args  
adapter_collect_args:
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle  
goto (*mh1.vmentry) // jump indirect to adapter_collect_args  
adapter_collect_args:  
push_frame           // for eventual call to #booleanValue
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle  
goto (*mh1.vmentry) // jump indirect to adapter_collect_args  
adapter_collect_args:  
push_frame           // for eventual call to #booleanValue  
mh2 = mh1.vmtarget   // a bound method handle
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry)    // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame             // for eventual call to #booleanValue
mh2 = mh1.vmtarget     // a bound method handle
goto (*mh2.vmentry)    // jump indirect to bound_ref_mh
bound_ref_mh:
```

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame // for eventual call to #booleanValue
mh2 = mh1.vmtarget // a bound method handle
goto (*mh2.vmentry) // jump indirect to bound_ref_mh
bound_ref_mh:
push mh2.vmargument // insert extra argument &"lessThan"
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
mh2 = mh1.vmtarget   // a bound method handle
goto (*mh2.vmentry) // jump indirect to bound_ref_mh
bound_ref_mh:
push mh2.vmargument // insert extra argument &"lessThan"
mh3 = mh2.vmtarget   // a direct method handle
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry)    // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
mh2 = mh1.vmtarget   // a bound method handle
goto (*mh2.vmentry)   // jump indirect to bound_ref_mh
bound_ref_mh:
push mh2.vmargument // insert extra argument &"lessThan"
mh3 = mh2.vmtarget   // a direct method handle
goto (*mh3.vmentry)   // jump indirect to invokestatic_mh
invokestatic_mh:      // vmentry for direct MH to static method
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
mh2 = mh1.vmtarget   // a bound method handle
goto (*mh2.vmentry) // jump indirect to bound_ref_mh
bound_ref_mh:
push mh2.vmargument // insert extra argument &"lessThan"
mh3 = mh2.vmtarget   // a direct method handle
goto (*mh3.vmentry) // jump indirect to invokestatic_mh
invokestatic_mh:      // vmentry for direct MH to static method
addr = mh3.vmtarget  // Runtime#invoke_2
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
mh2 = mh1.vmtarget   // a bound method handle
goto (*mh2.vmentry) // jump indirect to bound_ref_mh
bound_ref_mh:
push mh2.vmargument // insert extra argument &"lessThan"
mh3 = mh2.vmtarget   // a direct method handle
goto (*mh3.vmentry) // jump indirect to invokestatic_mh
invokestatic_mh:      // vmentry for direct MH to static method
addr = mh3.vmtarget  // Runtime#invoke_2
goto (*addr.from_interpreted_entry) // entry point of #invoke_2
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
mh2 = mh1.vmtarget   // a bound method handle
goto (*mh2.vmentry) // jump indirect to bound_ref_mh
bound_ref_mh:
push mh2.vmargument // insert extra argument &"lessThan"
mh3 = mh2.vmtarget   // a direct method handle
goto (*mh3.vmentry) // jump indirect to invokestatic_mh
invokestatic_mh:      // vmentry for direct MH to static method
addr = mh3.vmtarget  // Runtime#invoke_2
goto (*addr.from_interpreted_entry) // entry point of #invoke_2

static Object invoke_2(String m, Object x, Object y) { ... }
```



Method handle invocation pseudo-code (take 2)

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
```

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry)    // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
```

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
pop_frame           // return to call to mh1
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
pop_frame           // return to call to mh1
mh4 = mh1.argument // adapter method handle auxiliary
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame           // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
pop_frame           // return to call to mh1
mh4 = mh1.argument // adapter method handle auxiliary
push rax            // push object return value as argument
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
pop_frame // return to call to mh1
mh4 = mh1.argument // adapter method handle auxiliary
push rax // push object return value as argument
goto (*mh4.vmentry) // jump indirect to conversion method
```



Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
pop_frame // return to call to mh1
mh4 = mh1.argument // adapter method handle auxiliary
push rax // push object return value as argument
goto (*mh4.vmentry) // jump indirect to conversion method
...
```

Method handle invocation pseudo-code (take 2)

```
MethodHandle mh1 = cs.getTarget(); // an adapter method handle
goto (*mh1.vmentry) // jump indirect to adapter_collect_args
adapter_collect_args:
push_frame // for eventual call to #booleanValue
...
static Object invoke_2(String m, Object x, Object y) {
    return (Object) true;
}
pop_frame // return to call to mh1
mh4 = mh1.argument // adapter method handle auxiliary
push rax // push object return value as argument
goto (*mh4.vmentry) // jump indirect to conversion method
...
static boolean convL2Z(Object x) { return (boolean)x; }
```



Overview...

- What we did in JDK 7
- How it works in Hotspot
- **Advice to users**
- Next steps in JSR 292
- Building the future



Method handle life cycle

Method handle life cycle

- Creation (*direct MHs*)
 - reflective factory API: MethodHandles.Lookup
 - ldc of CONSTANT_MethodHandle
 - special factories: identity, invoker

Method handle life cycle

- Creation (*direct MHs*)
 - reflective factory API: MethodHandles.Lookup
 - ldc of CONSTANT_MethodHandle
 - special factories: identity, invoker
- Transformation (*bound or adapter MHs*)
 - bindTo, insertArguments, guardWithTest, etc.
 - asType, filterArguments, etc.

Method handle life cycle

- Creation (*direct MHs*)
 - reflective factory API: MethodHandles.Lookup
 - ldc of CONSTANT_MethodHandle
 - special factories: identity, invoker
- Transformation (*bound or adapter MHs*)
 - bindTo, insertArguments, guardWithTest, etc.
 - asType, filterArguments, etc.
- Invocation (*exact or inexact*)

Method handle life cycle

- Creation (*direct MHs*)
 - reflective factory API: MethodHandles.Lookup
 - ldc of CONSTANT_MethodHandle
 - special factories: identity, invoker
- Transformation (*bound or adapter MHs*)
 - bindTo, insertArguments, guardWithTest, etc.
 - asType, filterArguments, etc.
- Invocation (*exact or inexact*)
- Linking (*invokedynamic call site or other constant*)



Method handle life cycle: Creation

Method handle life cycle: Creation

- Method handle creation performs method linkage

Method handle life cycle: Creation

- Method handle creation performs method linkage
- Can be slow; do it just once if possible

Method handle life cycle: Creation

- Method handle creation performs method linkage
- Can be slow; do it just once if possible
- If generating bytecode, use CONSTANT_MethodHandle

Method handle life cycle: Transformation

Method handle life cycle: Transformation

- Bound or adapter method handles are additional nodes.
 - This adds internal pointers seen by GC and MH.invoke

Method handle life cycle: Transformation

- Bound or adapter method handles are additional nodes.
 - This adds internal pointers seen by GC and MH.invoke
- In OpenJDK7, creation requires a *JNI call*
 - This means even a simple transform can be surprisingly slow.



Method handle life cycle: Transformation

- Bound or adapter method handles are additional nodes.
 - This adds internal pointers seen by GC and MH.invoke
- In OpenJDK7, creation requires a *JNI call*
 - This means even a simple transform can be surprisingly slow.
- We hope to fix this soon.



Method handle life cycle: Invocation

Method handle life cycle: Invocation

- Exact invocation (MH.invokeExact) is simple.
 - In OpenJDK7, a pointer check and jump.
 - Only slightly more expensive than a regular method call.



Method handle life cycle: Invocation

- Exact invocation (MH.invokeExact) is simple.
 - In OpenJDK7, a pointer check and jump.
 - Only slightly more expensive than a regular method call.
- Inexact invocation (MH.invoke) can be complex.
 - If the types match exactly, same as MH.invokeExact.
 - If the types do not match, can perform asType transform.



Method handle life cycle: Invocation

- Exact invocation (MH.invokeExact) is simple.
 - In OpenJDK7, a pointer check and jump.
 - Only slightly more expensive than a regular method call.
- Inexact invocation (MH.invoke) can be complex.
 - If the types match exactly, same as MH.invokeExact.
 - If the types do not match, can perform asType transform.
- Invocation via an invokedynamic instruction is fast
 - Always exact (since call sites are strongly typed)
 - No runtime check of the type (VM enforces the invariant)
 - The MH code is typically inlined.



Method handle life cycle: Linking

Method handle life cycle: Linking

- An invokedynamic instruction is a constant CallSite...



Method handle life cycle: Linking

- An invokedynamic instruction is a constant CallSite...
- ...with an optimistically predicted target method handle.
 - Predicted handle is treated as a constant.

Method handle life cycle: Linking

- An invokedynamic instruction is a constant CallSite...
- ...with an optimistically predicted target method handle.
 - Predicted handle is treated as a constant.
- “static final” MH variables are also constant.

Method handle life cycle: Linking

- An invokedynamic instruction is a constant CallSite...
- ...with an optimistically predicted target method handle.
 - Predicted handle is treated as a constant.
- “static final” MH variables are also constant.
- Constant method handles (of any source) can be inlined
 - Inlining can boil away adapters and bound arguments.
 - Inlining can continue all the way through a direct method handle.



Bottom lines (*specific to OpenJDK7!*)

Bottom lines (*specific to OpenJDK7!*)

- Don't make new method handles in inner loops (yet).



Bottom lines (*specific to OpenJDK7!*)

- Don't make new method handles in inner loops (yet).
- Bind method handles to invokedynamic or constants.

Bottom lines (*specific to OpenJDK7!*)

- Don't make new method handles in inner loops (yet).
- Bind method handles to invokedynamic or constants.
- Prefer invokeExact.

Bottom lines (*specific to OpenJDK7!*)

- Don't make new method handles in inner loops (yet).
- Bind method handles to invokedynamic or constants.
- Prefer invokeExact.
- Work with us to prioritize benchmark/optimization work.

Overview...

- What we did in JDK 7
- How it works in Hotspot
- Advice to users
- Next steps in JSR 292
- Building the future



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.

Loose ends in the Java 7 API

Loose ends in the Java 7 API

- Method handle introspection (reflection)



Loose ends in the Java 7 API

- Method handle introspection (reflection)
- Generalized proxies (more than single-method intfs)



Loose ends in the Java 7 API

- Method handle introspection (reflection)
- Generalized proxies (more than single-method intfs)
- Class hierarchy analysis (override notification)



Loose ends in the Java 7 API

- Method handle introspection (reflection)
- Generalized proxies (more than single-method intfs)
- Class hierarchy analysis (override notification)
- Smaller issues:
 - usability (MethodHandle.toString, polymorphic bindTo)
 - sharp corners (MethodHandle.invokeWithArguments)
 - repertoire (tryFinally, more fold/spread/collect options)



Loose ends in the Java 7 API

- Method handle introspection (reflection)
- Generalized proxies (more than single-method intfs)
- Class hierarchy analysis (override notification)
- Smaller issues:
 - usability (MethodHandle.toString, polymorphic bindTo)
 - sharp corners (MethodHandle.invokeWithArguments)
 - repertoire (tryFinally, more fold/spread/collect options)
- Integration with other APIs (java.lang.reflect)



Support for Lambda in OpenJDK8

| © 2011 Oracle Corporation



Support for Lambda in OpenJDK8

- More transforms for SAM types (as needed).

Support for Lambda in OpenJDK8

- More transforms for SAM types (as needed).
- Faster bindTo operation to create bound MHs
 - No JNI calls.
 - Maybe multiple-value bindTo.



Support for Lambda in OpenJDK8

- More transforms for SAM types (as needed).
- Faster bindTo operation to create bound MHs
 - No JNI calls.
 - Maybe multiple-value bindTo.
- Faster inexact invoke (as needed).



Issue tracking

Issue tracking

- <http://java.net/jira/browse/MLVM>

Issue tracking

- <http://java.net/jira/browse/MLVM>
- This is an experiment. Can we make it work?



Issue tracking

- <http://java.net/jira/browse/MLVM>
- This is an experiment. Can we make it work?
- Public readable, writable after java.net login.



Issue tracking

Issue tracking

- <http://java.net/jira/browse/MLVM>

Issue tracking

- <http://java.net/jira/browse/MLVM>
- This is an experiment. Can we make it work?



Issue tracking

- <http://java.net/jira/browse/MLVM>
- This is an experiment. Can we make it work?
- (Is there a JIRA expert in the house?)

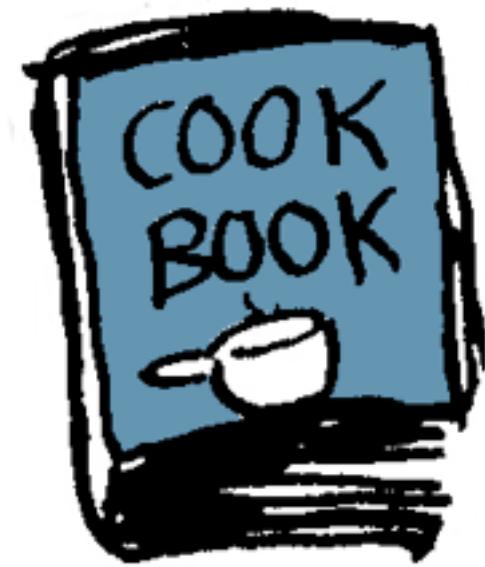


Overview...

- What we did in JDK 7
- How it works in Hotspot
- Advice to users
- Next steps in JSR 292
- Building the future

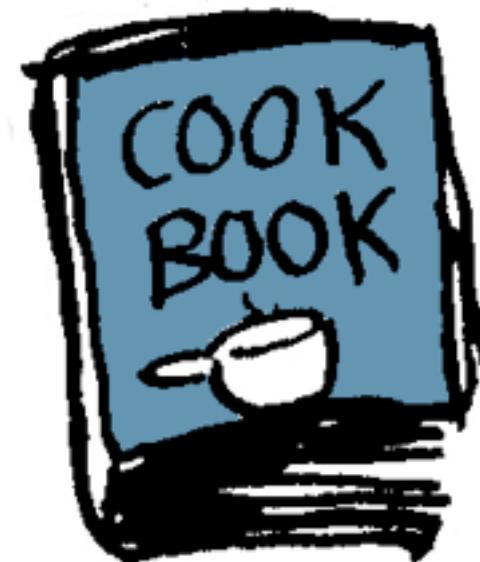


Let's build up libraries and client languages



Let's build up libraries and client languages

- Asm



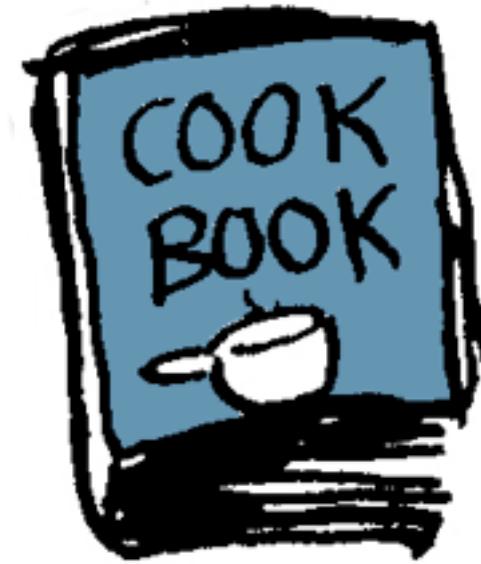
Let's build up libraries and client languages

- Asm
- Mirah



Let's build up libraries and client languages

- Asm
- Mirah
- Dynalang



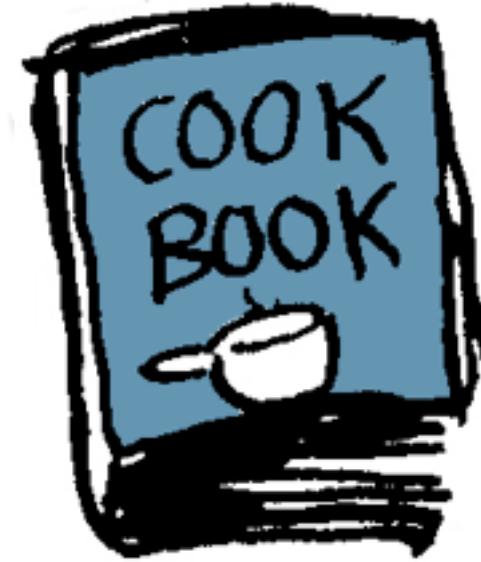
Let's build up libraries and client languages

- Asm
- Mirah
- Dynalang
- Indify



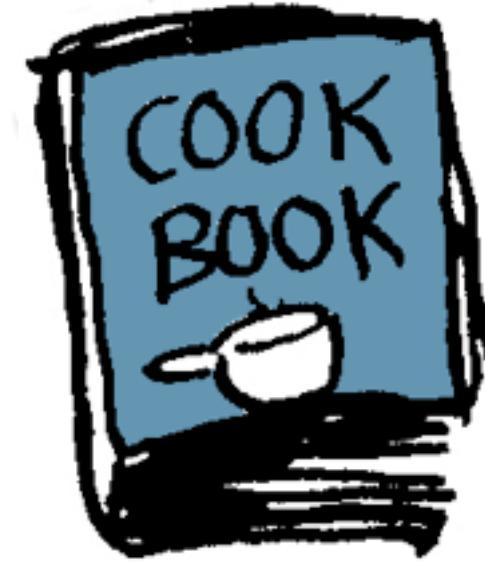
Let's build up libraries and client languages

- Asm
- Mirah
- Dynalang
- Indify
- JSR 292 backport



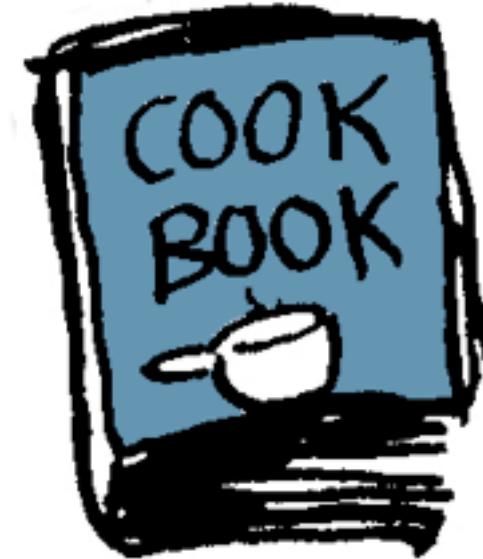
Let's build up libraries and client languages

- Asm
- Mirah
- Dynalang
- Indify
- JSR 292 backport
- MethodHandle transforms/adapters



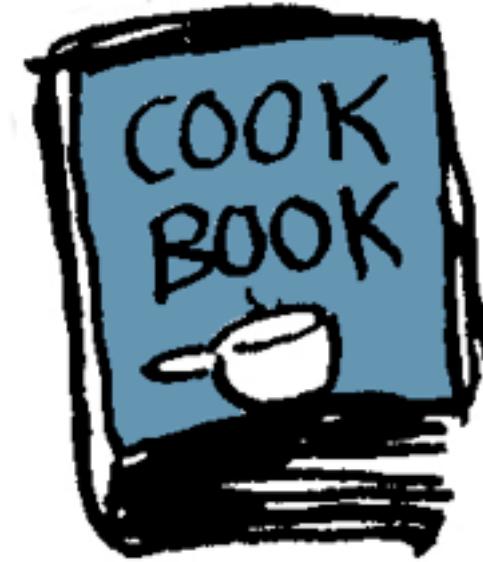
Let's build up libraries and client languages

- Asm
- Mirah
- Dynalang
- Indify
- JSR 292 backport
- MethodHandle transforms/adapters
- Ruby/Smalltalk/Groovy/Scala/...



Let's build up libraries and client languages

- Asm
- Mirah
- Dynalang
- Indify
- JSR 292 backport
- MethodHandle transforms/adapters
- Ruby/Smalltalk/Groovy/Scala/...
- *(your favorite language here)*



Let's nurture Project Lambda



Let's nurture Project Lambda

- More than aid to Lisp refugees...



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...
- Part of our attack on multi-cores



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...
- Part of our attack on multi-cores
- Optimization problems:



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...
- Part of our attack on multi-cores

- Optimization problems:
 - Support for closures via invokedynamic.



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...
- Part of our attack on multi-cores
- Optimization problems:
 - Support for closures via invokedynamic.
 - SAM / MH conversion (boxing/unboxing).



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...
- Part of our attack on multi-cores
- Optimization problems:
 - Support for closures via invokedynamic.
 - SAM / MH conversion (boxing/unboxing).
 - Loop versioning in the presence of closures.



Let's nurture Project Lambda

- More than aid to Lisp refugees...
- More than a renaissance of languages...
- Part of our attack on multi-cores
- Optimization problems:
 - Support for closures via invokedynamic.
 - SAM / MH conversion (boxing/unboxing).
 - Loop versioning in the presence of closures.
 - Online optimistic (re-)compilation.



Let's continue building our “future VM”

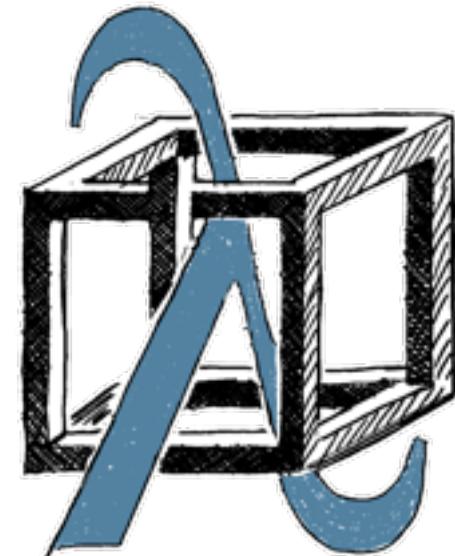
<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures
- Contains code fragments (patches).



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures
- Contains code fragments (patches).
- Movement to OpenJDK requires:



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures
- Contains code fragments (patches).
- Movement to OpenJDK requires:
 - a standard (e.g., JSR 292)



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures
- Contains code fragments (patches).
- Movement to OpenJDK requires:
 - a standard (e.g., JSR 292)
 - a feature release plan (7 vs. 8 vs. ...)



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures
- Contains code fragments (patches).
- Movement to OpenJDK requires:
 - a standard (e.g., JSR 292)
 - a feature release plan (7 vs. 8 vs. ...)
- bsd-port for developer friendliness.



Let's continue building our “future VM”

<http://hg.openjdk.java.net/mlvm/mlvm/hotspot/>

- Da Vinci Machine Project: an open source incubator for JVM futures
- Contains code fragments (patches).
- Movement to OpenJDK requires:
 - a standard (e.g., JSR 292)
 - a feature release plan (7 vs. 8 vs. ...)
- bsd-port for developer friendliness.
- mlvm-dev@openjdk.java.net



Current Da Vinci Machine patches

MLVM patches	
meth	method handles implementation
indy	invokedynamic
coro	lightweight coroutines
inti	interface injection (Tobias Ivarsson)
tailc	hard tail call optimization (Arnold Schwaighofer)
tuple	integrating tuple types (new from Michael Barker!)
hotswap	online general class schema updates (Thomas Wuerthinger)
anonk	anonymous classes; light weight bytecode loading

Caveat: Change is hard and slow

(especially the “last 20%”)



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.



Caveat: Change is hard and slow

(especially the “last 20%”)

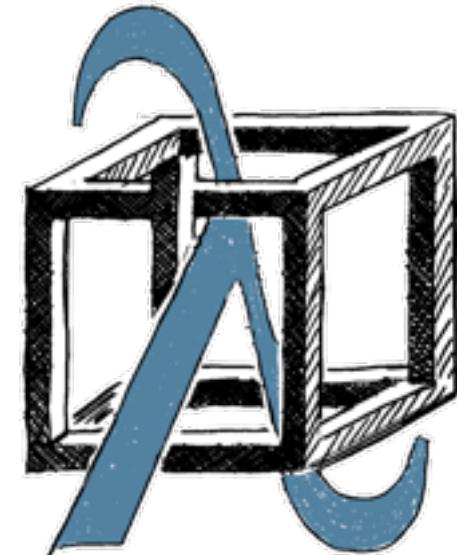
- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).



Caveat: Change is hard and slow

(especially the “last 20%”)

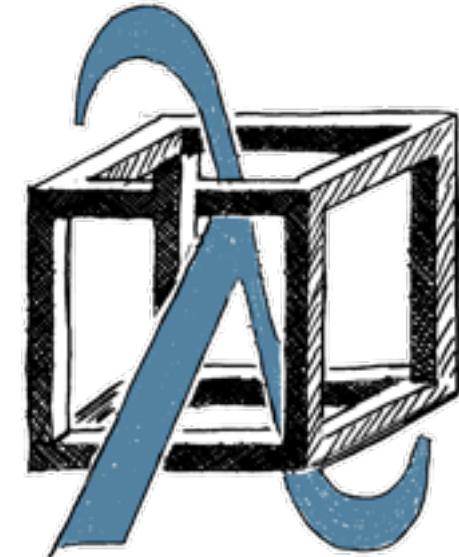
- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.
 - interpreter, multiple compilers



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.
 - interpreter, multiple compilers
 - managed heap (multiple GC algos.)



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.
 - interpreter, multiple compilers
 - managed heap (multiple GC algos.)
 - debugging, monitoring, profiling machinery



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.
 - interpreter, multiple compilers
 - managed heap (multiple GC algos.)
 - debugging, monitoring, profiling machinery
 - security interactions



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.
 - interpreter, multiple compilers
 - managed heap (multiple GC algos.)
 - debugging, monitoring, profiling machinery
 - security interactions
- ***Specifying*** is hard (the last 20%...).



Caveat: Change is hard and slow

(especially the “last 20%”)

- Hacking code is relatively simple.
- Removing bugs is harder.
- Verifying is difficult (millions of users).
- Integrating to a giant system very hard.
 - interpreter, multiple compilers
 - managed heap (multiple GC algos.)
 - debugging, monitoring, profiling machinery
 - security interactions
- **Specifying** is hard (the last 20%...).
- Running process is time-consuming.



Let's continue *designing* our “future VM”

<http://java.net/jira/browse/MLVM>



Let's continue *designing* our “future VM”

<http://java.net/jira/browse/MLVM>

- JIRA issue tracker for API conversations



Let's continue *designing* our “future VM”

<http://java.net/jira/browse/MLVM>

- JIRA issue tracker for API conversations
- Contains specification fragments.



Let's continue *designing* our “future VM”

<http://java.net/jira/browse/MLVM>

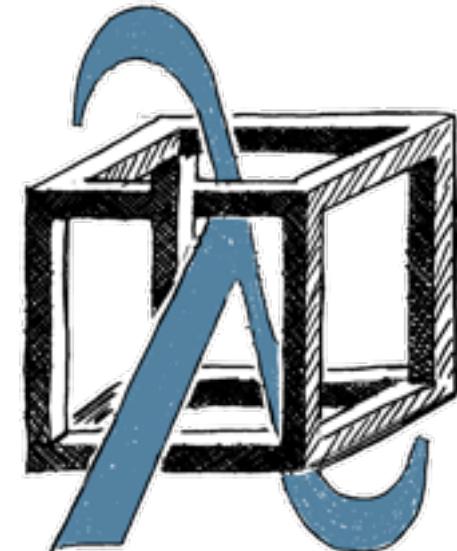
- JIRA issue tracker for API conversations
- Contains specification fragments.
- Requires login to java.net!



Let's continue *designing* our “future VM”

<http://java.net/jira/browse/MLVM>

- JIRA issue tracker for API conversations
- Contains specification fragments.
- Requires login to java.net!
- No code; for all JVMs (not just OpenJDK).



JVM design issue tracking by category

Category	Description
bytecode loading	Mechanisms for bytecode loading, such as anonymous classes, single-method loads, and whole-module loads.
coroutines	Coroutines, fibers, continuations, and other extensions to threading.
immutability	Immutable variables and objects. (I.e., beyond blank finals initialized in constructors.)
interface injection	JVM features related to online updates to interface APIs.
invokedynamic	The invokedynamic instruction, with related APIs such as <code>java.lang.invoke.CallSite</code> .
memory locality	Mechanisms for giving applications control (at the bytecode level) over memory locality, such as native layout, heterogeneous arrays, object fusing, object federation, and object replication.
method handles	<code>java.lang.invoke.MethodHandle</code> and related APIs. For bootstrap methods and <code>CallSite</code> , use the "invokedynamic" component.
reification	Mechanisms for reifying erased information, such as generic type instance parameters.
tailcalls	Bytecode support for hard (user-specified) tail call elimination.
tuples	Signature-polymorphic structural record types. Proposals may affect arguments, return values, fields, or array elements, and interoperability (via box/unbox) with reference types.
typestate	Mechanisms for updating the runtime type description of an object.



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>

- E.g., see my talk from last year



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>

- E.g., see my talk from last year
 - starting slide 25, “JVM features to investigate”



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>

- E.g., see my talk from last year
 - starting slide 25, “JVM features to investigate”
- Value versus Identity



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>

- E.g., see my talk from last year
 - starting slide 25, “JVM features to investigate”
- Value versus Identity
- Tail calls



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>

- E.g., see my talk from last year
 - starting slide 25, “JVM features to investigate”
- Value versus Identity
- Tail calls
- Better data structures



Discussions at previous Summits

<http://cr.openjdk.java.net/~jrose/pres/201007-JVMFutureTalk.pdf>

- E.g., see my talk from last year
 - starting slide 25, “JVM features to investigate”
- Value versus Identity
- Tail calls
- Better data structures
- Object species (typestate)
http://blogs.oracle.com/jrose/entry/larval_objects_in_the_vm



An integrating goal: Get multi.



| © 2011 Oracle Corporation

An integrating goal: Get multi.

- Thread/task/loop decomposition.



An integrating goal: Get multi.

- Thread/task/loop decomposition.
- Immutability & replication.



An integrating goal: Get multi.

- Thread/task/loop decomposition.
- Immutability & replication.
- Transactional APIs.



An integrating goal: Get multi.

- Thread/task/loop decomposition.
- Immutability & replication.
- Transactional APIs.
- Memory locality.



An integrating goal: Get multi.

- Thread/task/loop decomposition.
- Immutability & replication.
- Transactional APIs.
- Memory locality.
- More fluid code.



An integrating goal: Get multi.

- Thread/task/loop decomposition.
- Immutability & replication.
- Transactional APIs.
- Memory locality.
- More fluid code.
- More fluid data (traits, species).



Q&A

(continuing in today's workshop)

