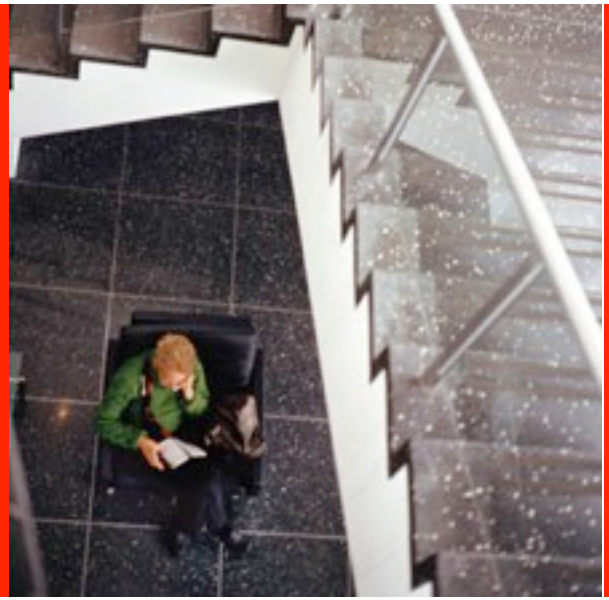

ORACLE®



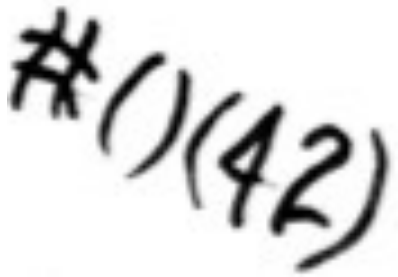
ORACLE

Gathering the threads

John R. Rose
Consulting Member of Technical Staff

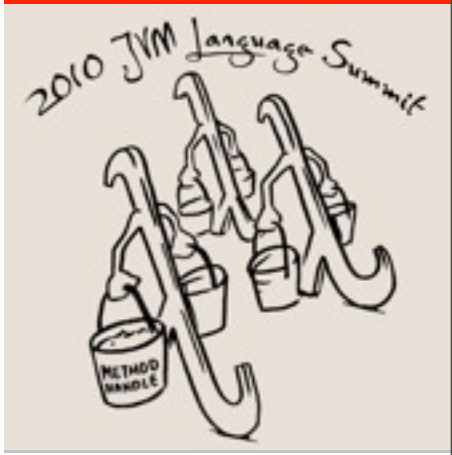
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Outline



- What's new with invokedynamic
- Why languages are important
  - (and threads are bad)
- JVM features to investigate

What's new with invokedynamic



JSR 292 News

(in the constant pool)

- New pool types for use with `ldc`
  - lazy symbol linkage, like `ldc` of CONSTANT_Class
- CONSTANT_MethodType (u2 Utf8_signature_ref)
- CONSTANT_MethodHandle (u1 kind, u2 ref)
  - kind is one of getField, putStatic, invokeVirtual, etc.
  - ref is a Fieldref, Methodref, or InterfaceMethodref
- No notational support (maybe Project Lambda?)
- Informally, we can write "`String#length()`"

JSR 292 News

(bootstrap methods in more places)

- New pool type for use with `invokedynamic`
  - scope + (name + type signature), like CONSTANT_Methodref
- Bootstrap method reference takes place of class.
  - BSM ref + (name + type signature)
  - BSM ref is a CONSTANT_MethodHandle
- Bootstrap method is a statically defined property
- Can be different for each `invokedynamic` instr.

Java Notation

(polymorphic signatures for MH & indy)

- Existing notation (since 2/2009) looks like:

```
int x = mh.<int>invokeExact("foo", false);
```

- Simplified notation uses target typing:

```
int x = mh.invokeExact("foo", false);
```

- Parameter and return types are equally implicit.

- Advising with Project Coin and Project Lambda

Java Notation

(local bootstrap methods)

- Existing notation looks like:

```
static { registerBootstrapMethod("foo"); }
```

- Declarative notation (*draft*):

```
@@@BootstrapMethod(name="foo")
```

- Advising with Project Coin and Project Lambda

Invokedynamic and Lambda

(what depends on what?)

- JSR 292 does not and will not require closures
  - It is an “assembly kit” for multiple language features
  - Closures are merely a feature in a language...
- But, of course, we will interoperate and synergize
- The `MethodHandle` is a useful building block
  - Non-bound closures could be direct method handles.
  - Bound closures could be bound direct method handles.
  - SAM conversion is a natural method handle operation

Case study

(generic arithmetic)

- Scheme: (set! acc (+ acc 1))
- Compiled: AddOp.$PI(acc, Integer.valueOf(1))
- Modified: InvokeDynamic.#"kawa:+"(acc, (int)1)

- Result: ~10% slower if already optimized
- ~10% faster if type profile is polluted (~20% penalty)
- ~80% of cost is Integer.valueOf (need *fixnums*)

  - See: http://blogs.sun.com/jrose/entry/an_experiment_with_generic_arithmetic
  - Thalinger & Rose, PPPJ 2010

Case study

(JavaScript objects)

- Rhino retrofit
- Hidden classes
- Monomorphic inline caches (using invokedynamic)

- One benchmark! ~4x performance gain
- What's left?
  - better indy calls
  - better split classes (*species*)
  - etc.

- See: kenai.com/projects/davincimonkey

ORACLE

Why languages are important

What's in a loop?

```
for (i = 0; i < a.length; i++)
  foo(a[i]);
```

- Essentially sequential
- Meaningless without side effects

What's in a loop?

```
100 let i = 0
110 goto 150
120 call foo(a[i])
130 let i = i+1
150 if i < a.length goto 120
```

fun with Lisp: a better “for”?

```
a.forEach(#foo);
```

- Still essentially sequential
- Still meaningless without side effects

Lisp with functions

```
a.mapReduce(...#foo...);
```

- Task decomposition unknown to caller
- Should be meaningless *with side effects*

“for, while, break, continue, if, else”

- Sequential control flow: Considered harmful
- Changing paradigms: Known to be painful

“Our language inhabits us”

- language channelizes our basic presumptions
- (about software and computers, at least)

Meanwhile, in the computer factory

- HW Designer: do you like my capability machine?
- SW Designer: ummm, you are on your own
- SW Designer: do you like my event processing paradigm?
- HW Designer: now you are on your own
- SW Designer: (I'll keep my exceptions to myself.)

Meanwhile, in the computer factory

- SW Designer: what I want is a new machine for my old software
- HW Designer: what I can make for you is an old machine, *tessellated*
- JVM Designer: Hi! What ‘cha talking about?

how the JVM is part of the problem

- fundamentally, the JVM is a graph-mutation machine
- the graphs are type-safe, GC-ed, class-rich, modular
- It is all based on side effects, hence sequential
- Threads help, but they also confuse us

why I hate threads

- they only help at grain size which is VERY LARGE
  - at most scales, the paradigms are still sequential
- a thread is a huge promise about many tasks
  - they will be run in sequence
  - at any given moment, there is a huge ugly backtrace
  - backed by a huge immobile stack
- a thread *claims* to represent a virtual processor
  - as processors get numerous, the claim gets thinner
  - even more, as processors get *cheaper*
- **thread : task :: memory page : typed object**

how the JVM can help

- the JVM is also an *excellent* function-calling machine
- now we are cross-breeding functions & object graphs

- let's push for light-weight events and tasks
- let's design APIs that can allow task decomposition

JVM features to investigate

Value versus Identity

- Every Java object has a identity:
  - operator==, System.identityHashCode, clone
  - serialization: sync/wait/notify
  - these are necessary to manage the burden of **mutable fields**
  - a sequential, continuous story
    - fields change but the object remains the same
  - “His right rear paw is in a cast, but he’s still good old Rex.”
- Pure values are frozen in timelessness, have no story
  - comparisons have only to do with the permanent value parts
  - values don’t change; values can help us produce new similar

Tail calls

- One task can delegate to another w/o stack overflow
- Open loops
  - flexible, dynamic task decomposition
  - delegating (“threaded”!) control flow
- Sample syntax (prototype only!)

```
public static int tailcaller(int x) {
  if (x==0) return x+1;
  return goto tailcaller(x-1);
}
```

  - See: http://blogs.sun.com/jrose/entry/tail_calls_in_the_vm
  - http://hg.openjdk.java.net/mlvm/mlvm/hotspot/file/tip/tailc.txt

Better data structures

- Mixed arrays: [length: 2]{String, int, String, int}
- Array headers: {Object,String}[length: 2]{...}

- Why better?
  - object graphs ≠ memory structs
  - indirections are difficult to collapse
  - move back toward explicit memory layout

- We care, today, because of cache effects

- Better, really?
  - Research problem: treat this all as a hidden optimization

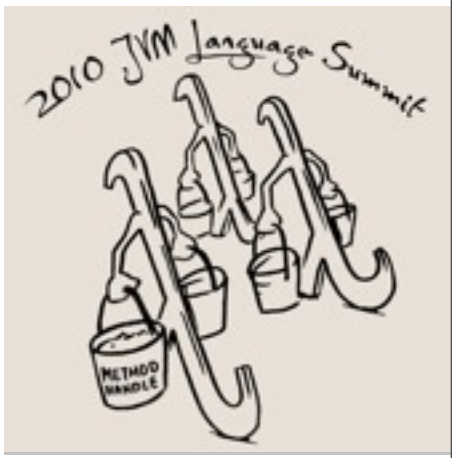
Origin of Species?

- List<String> strs = ...;
  List<Integer> ints = ...;
  String s = strs.get(0); int n = ints.get(0);
  assert strs.getClass() == ints.getClass();
- Species<List<String>> slspec = strs.getSpecies();
  Species<List<Integer>> ilspec = ints.getSpecies();
  assert slspec != ilspec;
- slspec.getString(); // List<String>
  slspec.getEnclosingClass() == List.class;
- GenericTypeSpecies g = (GenericTypeSpecies)slspec
  assert g.getTypeArg(0) == String.class;

Conclusion

- Paradigm shift hurts
- Getting left behind hurts even more
- Let's help our programmers adapt to tessellated HW

Q & A ... Workshop




ORACLE®