ORACLE®

Java 8: Selected Updates

John Rose —Da Vinci Machine Project Lead & Java Nerd
April 2, 2012
http://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2012

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.

Warning: Dense Slides Ahead!

It's all on http://cr.openjdk.java.net/~jrose/pres:

http://cr.openjdk.java.net/~jrose/pres/201204-LangNext.pdf

What's Brewing!

http://openjdk.java.net/projects/jdk8/

- **Modules: Project Jigsaw**
- "Nashorn" JavaScript engine
  - uses invokedynamic; strong Java integration
- JVM convergence (JRockit + HotSpot)
  - "permgen" removal, manageability hooks, optimizations
- **Project Lambda**
  - Better inner classes; defender methods (= no-state traits)
- Technical debt: going into collections
  - Lambda queries, fork/join integration, immutability, etc.
- More: APIs, annotations, OS X, Java FX, etc., etc.

Jigsaw: A few big pieces

http://mreinhold.org/blog/jigsaw-focus

- Retire the classpath.
- Explicit module versions and dependencies.
- Consistent behaviors for compile, build, install, run.
  - This means language, toolchain, and VM integration!
- Encapsulation: Real privacy, module composition.
  - A social network for code.
- Optionality, providers, platforms.
  - Pluggable impls., including platform-specific native code.
- Stop torturing ClassLoaders.
- Works with JARs, maven, OSGi, Debian, etc., etc. (!)

Nashorn: A rhino with an attitude

http://wiki.jvmlangsummit.com/images/c/ce/Nashorn.pdf

- Clean rewrite on JVM of ECMAScript-262-5.
- State-of-the-art map based data structures.
  - Map normalization and reuse.
- Builds inline caches with invokedynamic.
  - I.e., mutable call sites with variable profiles and targets.
  - Invokedynamic surfaces the recompilation magic.
- Full use of Hotspot JVM GC and JIT.
- Strong interoperability with Java.

A convergence of JVMs

https://blogs.oracle.com/henrik/

- JRockit and Hotspot today
- And SE/ME/CDC tomorrow, using module system
- Oracle JRockit and Hotspot teams have merged

- Monitoring/manageability hooks ported to HS
- Removing "permgen" using JR design (Java 7 and 8)
  - Helps scale very broad or very dynamic systems.
- Working on combined optimization algorithms
  - Example: Escape analysis, flow insensitive + flow sensitive
  - Inlining heuristics, including manually directed ones.

Big λ Goal: parallel queries!

http://blogs.oracle.com/briangoetz/resource/devoxx-lang-lib-vm-co-evol.pdf

- Make parallel (collection) computations simple.
- And similar in look & feel to serial computation.
- (A familiar goal… Cf. LINQ.)

Key parts require lambdas:

- Internal iterators, not classic Java external iterators
- Chained queries, not side effects or accumulators.

```
people.filter(p -> p.age() >= 21)
  .sort(comparing(Person::getLastName));
```

What's in a (Java-style) lambda?

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html

- (Type) Params '->' Body
  - Examples: `()->42, x->x+1, (int x)->{foo(x);}`
- Type = target type from assignment, invocation, etc.
  - Must be a functional interface type (e.g., `Runnable`)
  - Typically inferred from context, could be an explicit cast.
- Params = '(' type var … ')'
  - Elided types can be inferred from lambda target type
  - Can elide parens if arity = 1
- Body = expression | block
  - In a block, '`return`' keyword presents the result value

Lambdas in Java and C#

- Minor syntax differences. (Arrow shaft, dot arity.)
- Functional **interfaces** vs. **sealed** delegate types.
  - A functional ("SAM type") interface is a *pattern*, an *old* one.
- Type inference, type matching differ. (Naturally.)
- Captured outer variables **must be constants** in Java
- Java 8 has no reification, no expression trees. Alas.

Similarities (besides basic lambda-ness)

- Contextual typing; lambdas have no intrinsic type.
- No branching to outer scopes.

Outer variable capture (more details)

- Captured outer variables must be constants in Java.
  - Same rule as for inner/nested classes.
- This restriction was not (and is not) a mistake:
  - `for (i=0;i<4;i++) launch(()->doTask(`~~`i`~~`));`
- Can elide "`final`" under new "effectively final" rules.

- Even for "safe" uses, mutable accumulators are bad.
  - Accumulators are inherently serial. The future is functional!
  - ~~`int a=0; es.forEach((e)->{a+=e.salary;});`~~

Outer variable capture (an alternative)

MSDN Blogs > Fabulous Adventures In Coding > Closing over the loop variable considered harmful

Closing over the loop variable considered harmful

Eric Lippert 12 Nov 2009 6:50 AM | 133

RATE THIS

(This is part one of a two-part series on the loop-variable-closure problem. Part two is here.)

UPDATE: We **are** taking the breaking change. In C# 5, the loop variable of a foreach will be logically inside the loop, and therefore **closures will close over a fresh copy of the variable each time.** The "for" loop will not be changed. We return you now to our original article.

I don't know why I haven't blogged about this one before; this is the single most common incorrect bug report we get. That is, someone thinks they have found a bug in the compiler, but in fact the compiler is correct and their code is wrong. That's a terrible situation for everyone; we very much wish to design a language which does not have "gotcha" features like this.

Method references

- Unbound: `String::length`
- Bound: `"pre-"::concat`
- Constructor: `StringBuffer::new`

Comparable lambdas:

- Unbound: `(String s) -> s.length`
- Bound: `(String t) -> "pre-".concat(t)`
- Constructor: `() -> new StringBuffer()`

C#, with delegate typing magic, has ¼ the dots!

Lambda example: Query on collection

- Functional type (aka "Single Abstract Method"):
  - `interface Predicate<T> {boolean apply(T t);}`
- Queryable type, with higher-order methods:
  - `Collection<T> filter(Predicate<T> p) { … }`
- The end user writes this:
  - `kids = people.filter(p -> p.age() < agelim);`
- The compiler infers λ-type `Predicate<Integer>`

Fattening up the collection types

- Higher-order methods are not found in `List`, etc.
- New in Java 8: extension ("defender") methods.
  - ```
    interface List<T> … { …
      List<T> filter(Predicate<T> p)
        default { … }
      … }
    ```
- Default method supplied to all implementations.
  - As with abstract classes, subtypes can override.
  - This shares algorithmic responsibility. (Not just sugar!)
- Details are TBD. Stay tuned http://blogs.oracle.com/briangoetz
    ```

Translation options for lambdas

- Could just translate to inner classes
  - `p -> p.age() < agelim` translates to

    ```
    class Foo$1 implements Predicate<Person> {
        private final int v0;
        Foo$1(int $v0) { this.$v0 = v0 }
        public boolean apply(Person p) {
            return (p.age() < $v0);
        }
    }
    ```
- Capture == invoke constructor (`new Foo$1(agelim)`)
- One class per lambda expression – yuck, JAR explosion
- Would burden lambdas with identity
  - Would like to improve performance over inner classes
- Why copy yesterday's mistakes?

Translation options

- Could translate directly to method handles
  - Desugar lambda body to a static method
  - Capture == take method reference + curry captured args
  - Invocation == MethodHandle.invoke
- Whatever translation we choose becomes not only implementation, but a binary specification
  - Want to choose something that will be good forever
  - Is the MH API ready to be a permanent binary specification?
  - Are raw MHs yet performance-competitive with inner classes?

Translation options

- What about “inner classes now and method handles later”?
  - But old class files would still have the inner class translation
  - Java has never had “recompile to get better performance” before
- Whatever we do now should be where we want to stay
  - But the “old” technology is bad
  - And the “new” technology isn’t proven yet
  - What to do?

Invokedynamic to the rescue!

- We can use invokedynamic to delay the translation strategy until runtime
  - Invokedynamic was originally intended for dynamic languages, not statically typed languages like Java
  - But why should the dynamic languages keep all the dynamic fun for themselves?
- We can use invokedynamic to embed a *recipe* for constructing a lambda at the capture site
  - At first capture, a translation strategy is chosen and the call site linked (the strategy is chosen by a **metafactory**)
  - Subsequent captures bypass the slow path
  - As a bonus, stateless lambdas translated to constant loads

Layers of cost for lambdas

- Any translation scheme imposes phase costs:
  - Linkage cost – one-time cost of setting up capture
  - Capture cost – cost of creating a lambda
  - Invocation cost – cost of invoking the lambda method
- For inner class instances, these costs are
  - Linkage: loading the class (`Foo$1.class`)
  - Capture: invoking the constructor (`new Foo$1(agelim)`)
  - Invocation: invokeinterface (`Predicate.apply`)
- The key phase to optimize is *invocation*
  - Capture is important too, and must be inlinable.

Layers of cost for lambdas (take two)

- For invokedynamic, the phase costs are flexible:
- Linkage: **metafactory** selects a local lambda factory
- Capture: Invokes the local lambda factory.
- Invocation: invokeinterface (as before)

- The metafactory decides, once, how to spin each λ
  - It can spin inner classes, and/or tightly couple to the JVM.
  - The metafactory is named symbolically in the class file.
  - Its behavior is totally decoupled from the bytecode shape.

Code generation strategy

- All lambda bodies are **desugared** to static methods
  - For "stateless" (non-capturing) lambdas, lambda signature matches SAM signature exactly

```
(Person p) -> p.age() < 18
```

  - Becomes (when translated to `Predicate<String>`)

```
private static boolean lambda$1(Person p) {
  return p.age() < 18;
}
```

- In this case, the lambda instance λ_0 can be created eagerly by the metafactory.
  - The meta factory uses a K combinator, so that the linked semantics of the invokedynamic instruction becomes $K(\lambda_0)$.

Code generation strategy

- For lambdas that capture variables from the enclosing context, these are prepended to the argument list.
  - So we can freely copy variables at point of capture

```
(Person p) -> p.age() < agelim
```

  - Becomes (when translated to `Predicate<String>`)

```
private static boolean lambda$2(int agelim,
                                Person p) {
  return p.age() < agelim;
}
```

- Desugared (lifted) `lambda$2` is a curried function.

Code generation strategy

- At point of lambda capture, compiler emits an invokedynamic call to the local lambda factory
  - Bootstrap is metafactory (standard language runtime API)
  - Static arguments identify properties of the lambda and SAM
  - Call arguments are the captured values (if any)

```
list.filter(p -> p.age() < agelim);
```

becomes

```
list.filter(indy[BSM=Lambda::metafactory,
   body=Foo::lambda$2,
   type=Predicate.class]( agelim ));
```

- Static args encode properties of lambda and SAM
  - Is lambda cacheable? Is SAM serializable?

Benefits of invokedynamic

- Invokedynamic is the ultimate lazy evaluation idiom
  - For stateless lambdas that can be cached, they are initialized at first use and cached at the capture site
  - Programmers frequently cache inner class instances (like Comparators) in static fields, but indy does this better
- No overhead if lambda is never used
  - No field, no static initializer
  - Just some extra constant pool entries
- SAM conversion strategy becomes a pure implementation detail
  - Can be changed dynamically by changing metafactory

What’s dynamic about invokedyamic?

- Invokedynamic has user-defined **linkage semantics**.
  - Defined by a per-instruction “bootstrap method” or BSM.
- In the case of lambda, the BSM is the metafactory.
- Invokedynamic linkage info is open-ended.
  - BSM has up to 252 optional arguments from constant pool.
- For lambda, BSM takes a couple extra BSM args.
  - Method handle reference to desugared body.
  - Class reference to target type (functional interface).
  - Added in Java 8: Method handle constant cracking.
- (Caveat: The BSM is hairier for serializables.)

(That's not very dynamic, is it?)

- (Invokedynamic also provides **mutable call sites**.)
- (But this feature is not used by Lambda.)

- Used for JRuby (1.7), Nashorn, Smalltalk, etc.

- ∴ Indy = linker macros + mutable call sites.

- *Linker macros can help with any language implementation plagued by small class file infestations.*

Invokedynamic odds & ends (Java 7)

For the record: Late developments from Java 7.

- Bootstrap method takes any constant arguments.
- Each invokedynamic instruction (potentially) has its own bootstrap method arguments.
- Constant pool holds method handles, method types.
- Method handles are fully competent with Java APIs.
  - Including autoboxing & varargs conversions, when approp.
  - Big exception: The types are erased.
  - Small exception: “invokespecial <init>” not available.

After 8 comes ∞

- More stuff incubating in the Da Vinci Machine Project
- Some possibilities:
  - Tailcall, coroutines, continuations
  - Extended arrays
  - Primitive / reference unions
    http://hg.openjdk.java.net/mlvm/mlvm/hotspot/file/tip/tagu.txt
  - Tuples, value types
    https://blogs.oracle.com/jrose/entry/value_types_in_the_vm
  - Species, larvae, typestate, reification
    https://blogs.oracle.com/jrose/entry/larval_objects_in_the_vm

Other channels to tune in on…

- Maxine project: Java as a system language.
  - https://wikis.oracle.com/display/MaxineVM/Home
- Graal project (OpenJDK): Self-hosting JIT.
  - http://openjdk.java.net/projects/graal/

- JVM Language Summit **2012**
  - **July 30 – August 1**; Oracle Santa Clara (same as last year)
  - CFP coming in a few days

http://openjdk.java.net/

∞…

P.S. The Java/JVM team is hiring!

https://blogs.oracle.com/jrose/entry/the_openjdk_group_at_oracle