Lambda: A peek under the hood

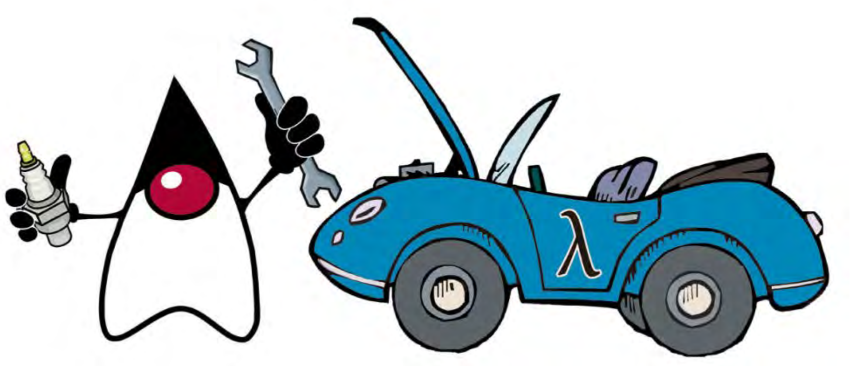
Brian Goetz
Java Language Architect, Oracle

ORACLE

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.

About this talk

- This talk outlines the technical details of how lambda expressions are implemented in Java SE 8 at the bytecode level
  - This will be a highly technical talk!
  - We assume some familiarity with JVM and bytecode
    - Including JSR-292 facilities



ORACLE

Lambda expressions for Java

- A lambda expression is like an “anonymous method”
  - Has an argument list, a return type, and a body

```
people.forEach( (Person e) -> names.add(e.getName()) );
```

  - Many of these can be inferred by the compiler

```
people.forEach(e -> names.add(e.getName()) );
```

- Lambda expressions can capture values from the enclosing context

```
people.removeIf(e -> e.getAge() < minAge);
```

Why lambdas for Java?

- Provide libraries a path to multicore
  - Parallel-friendly APIs need internal iteration
  - Internal iteration needs a concise code-as-data mechanism
- Empower library developers
  - Enable a higher degree of cooperation between libraries and client code
- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point to not have closures
  - Adding closures to Java is no longer a radical idea

Why lambdas for Java?

- Java already has inner classes, isn't that enough?
  - Inner classes give us some of these benefits, but are too clunky
  - Clunkiness is not entirely syntactic!
    - Complex naming lookup rules
- How to represent lambda expressions at runtime is not a trivial question!
  - Need a clean story at both the language and VM levels
  - That's what this talk is about

Big question #1: typing

- What is the *type* of a lambda expression?
  - Most languages with lambdas have some notion of a *function type*
    - “function from int to int”
  - Early proposals included adding *function types* to the type system
  - Seems an obvious idea, so why not?
    - Proposals rarely went deeper than syntax
    - Need a good story for how to translate to the bytecode level
    - Without impedance mismatches

Why not “just” add function types?

- Inconvenient questions
  - How would we represent functions in VM method/field type signatures?
    - Need a way to write `void method(String -> int f)` in bytecode
    - Method signatures are (currently) entirely nominal
  - How would we represent invocation in bytecode?
  - How would we create instances of function-typed variables?
  - How would we deal with variance?

ORACLE

Why not "just" add function types?

- JVM has no native representation of function type in VM type signatures
  - Closest tool we have is generics
  - To represent `List<String> -> int` with generics…
    - Would have to box int to Integer
    - Would have to erase List<String> to List
  - Boxed and erased function types would be no fun
  - Gaps between language and VM representation are always pain points
- Teaching the VM about "real" function types would be a huge effort
  - New type signatures for functions, new bytecodes for invocation, new verification rules

Alternative: functional interfaces

Use what you know

- Historically we modeled functions using single-method interfaces
  - Runnable, Comparator
- Rather than complicate the type system, let’s just formalize that
  - Give them a name: “functional interfaces”
  - Always convert lambda expressions to instance of a functional interface

```
interface Predicate<T>  { boolean test(T x); }

people.removeIf(p -> p.getAge() >= 18);
```

  - Compiler identifies Predicate as a functional interface (structurally)
  - Compiler infers the lambda is a Predicate<Person>

ORACLE

Functional interfaces

- “Just add function types” was obvious … and wrong
  - Would have introduced complexity and corner cases
  - Would have bifurcated libraries into “old” and “new” styles
    - Would have spent years migrating the “old” libraries to the “new” style
    - For very little value
  - Would have created interoperability challenges
    - Interoperating old libraries with new

ORACLE

Functional interfaces

- People already understand one-method interfaces
  - So they already understand the typing for lambdas
- Bonus: existing libraries are now *forward-compatible* to lambdas
  - Libraries that never imagined lambdas still work with them!
  - Maintains significant investment in existing libraries
  - Fewer new concepts
- Lesson learned: a stodgy old approach is sometimes better than a shiny new one!

ORACLE

Big question #2: representation

- How does the lambda instance get created?
  - Need to convert a function into an instance of a functional interface
  - Need to handle captured variables

```
Predicate<Person> pred = (Person p) -> p.age < minAge ;
```

- The obvious choice is … inner classes

ORACLE

Why not "just" use inner classes?

- We could say that a lambda is "just" an inner class instance

```
    class Foo$1 implements Predicate<Person> {
        private final int $minAge;
        Foo$1(int v0) { this.$minAge = v0; }
        public boolean test(Person p) {
            return p.age < $minAge;
        }
    }
```

- Then, lambda capture becomes constructor invocation

```
list.removeIf(p -> p.age < minAge);
```

→

```
list.removeIf(new Foo$1(minAge));
```


ORACLE

Why not “just” use inner classes?

- Inner class constructor invocation translates as

```
list.removeIf(p -> p.age < minAge);
```

↓

```
aload_1                 // list
new #2                  // class Foo$1
dup
iload_2                 // minAge
invokespecial #3        // Method Foo$1."<init>":(I)V
invokeinterface #4      // java/util/List.removeIf:(Ljava/util/functions/Predicate;)Z
```


Why not "just" use inner classes?

- Translating to inner classes means we inherit most of their problems
  - Performance issues
    - One class per lambda expression
    - Type profile pollution
    - Always allocates a new instance
  - Complicated and error-prone "comb" lookup for names
- Whatever we do becomes a *binary representation* for lambdas in Java
  - Would be stuck with it forever
  - Would rather not conflate implementation with binary representation
- Another "obvious but wrong" choice

New bytecode tool: MethodHandle

- Java SE 7 adds VM-level *method handles*
  - Can store references to methods in the constant pool, load with LDC
  - Can obtain a method handle for any method (or field access)
  - VM will happily inline through MH calls
  - API-based combinators for manipulating method handles
    - Add, remove, reorder arguments
    - Adapt (box, unbox, cast) arguments and return types
    - Compose methods
  - Compiler writers swiss-army knife!

ORACLE

Why not “just” use MethodHandle?

- At first, translating lambdas to MethodHandle seems obvious
  - Lambda is language-level method object
  - MethodHandle is VM-level method object
- We could
  - Desugar lambdas expressions to methods
  - Represent lambdas using MethodHandle in bytecode signatures

Why not “just” use MethodHandle?

- If we represented lambdas as MethodHandle, we’d translate:

```
list.removeIf(p -> p.age < minAge);
```

↓

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

```
MethodHandle mh = LDC[lambda$1];
mh = MethodHandles.insertArguments(mh, 0, minAge);
list.removeIf(mh);
```

 | |

ORACLE

Why not “just” use MethodHandle?

- If we did this, the signature of List.removeIf would be:
  `void removeIf(MethodHandle predicate)`
- This is erasure on steroids!
  - Confusing: can’t overload methods that take differently “shaped” lambdas
    - `void foo(String -> int)`
    - `void foo(int -> int)`
  - Plus, still would need to encode the erased type information somewhere
  - (Also: is invocation performance competitive with bytecode invocation?)
- Again: conflates binary representation with implementation
  - Obvious but wrong … again

Stepping back…

- We would like a binary interface not tied to a specific implementation
  - Inner classes have too much baggage
  - MethodHandle is too low-level, is erased
  - Can’t force users to recompile, ever, so have to pick now
- What we need is … another level of indirection
  - Let the static compiler emit a declarative *recipe*, rather than *imperative code*, for creating a lambda
  - Let the runtime execute that recipe however it deems best
  - And make it darned fast
  - This sounds like a job for invokedynamic!

ORACLE

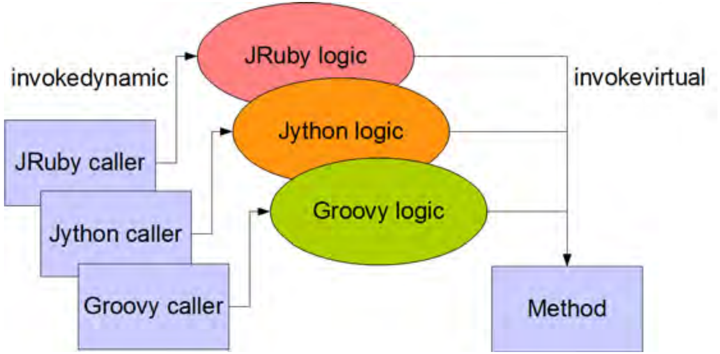
Bytecode invocation modes

- Prior to Java SE 7, the JVM had four bytecodes for method invocation
  - invokestatic: for static methods
  - invokevirtual: for class methods
  - invokeinterface: for interface methods
  - invokespecial: for constructors, private methods, and super-calls
- Each specifies a class name, method name, and method signature

```
aload_1                // list
new #2                 // class Foo$1
dup
iload_2                // minAge
invokespecial #3       // Method Foo$1."<init>":(I)V
invokeinterface #4     // java/util/List.removeIf:(Ljava/util/functions/Predicate;)Z
```


New bytecode tool: invokedynamic

- Java SE 7 adds a fifth invocation mode: *invokedynamic* (indy)
- Behavior of invoke{virtual,static,interface} are fixed and Java-like
  - Other languages need custom method linkage
- Basic idea: let some "language logic" determine call target
  - But then *get out of the way*
  - Language and VM become partners in flexible and efficient method dispatch



New bytecode tool: invokedynamic

- Invokedynamic started out as a tool for dynamic languages
  - A typical application would be invoking a method like
    `def add(a, b) { a+b }`
- Here, the types of a and b are not known at compile time
  - And can change from call to call … *but probably don't*
    - Good chance that, if called with two ints, next call will be with two ints
  - We win by not having to re-link the call site for every invocation

New bytecode tool: invokedynamic

- The first time the JVM executes an invokedynamic
  - Consults the *bootstrap method* for the call site (the “language logic”)
  - Bootstrap returns a linked call site
  - Call site can embed conditions under which it needs relinking (if any)
    - Such as the argument types changing
    - Otherwise, JVM does not have to consult bootstrap again
- After linkage, JVM can treat the call site as fully linked
  - Can inline through linked indy callsites like any other

New bytecode tool: invokedynamic

- An indy callsite has three groups of operands
  - A *bootstrap method* (the “language logic”)
    - Called by the VM for linking the callsite on first invocation
    - Not called again after that
  - A static argument list, embedded in the constant pool
    - Available to the bootstrap method
  - A dynamic argument list, like any other method invocation
    - Not available to the bootstrap, but their *static types and arity* are
    - Passed to whatever target the callsite is linked to

ORACLE

Its not just for dynamic languages anymore

- So, if indy is for dynamic languages, why is the Java compiler using it?
  - All the types involved are static
  - What is dynamic here is the *code generation strategy*
    - Generate inner classes?
    - Use method handles?
    - Use dynamic proxies?
    - Use VM-private APIs for constructing objects?
- Indy lets us turn this choice into a pure implementation detail
  - Separate from the binary representation

ORACLE

Its not just for dynamic languages anymore

- We use indy to embed a *recipe* for constructing a lambda, including
  - The desugared implementation method (static)
  - The functional interface we are converting to (static)
  - Additional metadata, such as serialization information (static)
  - Values captured from the lexical scope (dynamic)
- The capture site is called the *lambda factory*
  - Invoked with indy, returns an instance of the desired functional interface
  - Subsequent captures bypass the (slow) linkage path

Desugaring lambdas to methods

- First, we desugar the lambda to a method, as before
  - Signature matches functional interface method
  - Plus captured arguments prepended
  - Simplest lambdas desugar to static methods
    - But some need access to receiver, and so are instance methods

```
Predicate<Person> pred = p -> p.age < minAge;
```

↓

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


Factories and metafactories

- We generate an indy call site which, when called, returns the lambda
  - This is the *lambda factory*
  - Bootstrap for the lambda factory selects the translation strategy
    - Bootstrap is called the *lambda metafactory*
    - Part of Java runtime
  - Captured args passed to lambda factory

```
list.removeIf(p -> p.age < minAge);
```

```
Predicate $p = indy[bootstrap=LambdaMetafactory,
                    staticargs=[Predicate, lambda$1],
                    dynargs=[minAge])
list.removeIf($p);
```

```
private static boolean lambda$1(int minAge, Person p) {
    return p.getAge() >= minAge;
}
```


ORACLE

Translation strategies

- The metafactory could spin inner classes dynamically
  - Generate the same class the compiler would, just at runtime
  - Link factory call site to constructor of generated class
    - Conveniently, dynamic args and ctor args will line up
  - Our initial strategy until we can prove that there’s a better one
- Alternately could spin one wrapper class per interface
  - Constructor would take a method handle
  - Methods would invoke that method handle
- Could also use dynamic proxies or MethodHandleProxy
- Or VM-private APIs to build object from scratch, or…

Indy: the ultimate procrastination aid

- By deferring the code generation choice to runtime, it becomes a pure implementation detail
  - Can be changed dynamically
  - We can settle on a binary protocol now (metafactory API) while delaying the choice of code generation strategy
    - Moving more work from static compiler to runtime
  - Can change code generation strategy across VM versions, or even days of the week

ORACLE

Indy: the ultimate lazy initialization

- For stateless (non-capturing) lambdas, we can create one single instance of the lambda object and always return that
  - Very common case – many lambdas capture nothing
  - Sometimes we do this by hand in source code – e.g., pulling a Comparator into a static final variable
- Indy functions as a lazily initialized cache
  - Defers initialization cost to first use
  - No heap overhead if lambda is never used
  - No extra field or static initializer
  - All stateless lambdas get lazy initialization and caching for free

Indy: the ultimate indirection aid

- Just because we defer code generation strategy to runtime, we don’t have to pay the price on every call
  - Metafactory only invoked once per call site
  - For non-capturing case, subsequent captures are FREE
    - VM optimizes to constant load
  - For capturing case, subsequent capture cost on order of a constructor call / method handle manipulation
    - MF links to constructor for generated class

ORACLE

Performance costs

- Any translation scheme imposes costs at several levels:
  - Linkage cost – one-time cost of setting up lambda capture
  - Capture cost – cost of creating a lambda instance
  - Invocation cost – cost of invoking the lambda method
- For inner class instances, these correspond to:
  - Linkage: loading the class
  - Capture: invoking the constructor
  - Invocation: invokeinterface

  ORACLE

Performance example – linkage cost

- Oracle Performance Team measured linkage costs
  - Time in ms for 32K distinct inner classes / lambdas
  - Fastest possible disk (to negate IO component of anon class loading)

	Anonymous	Lambda	Difference
-Capturing -Tiered	7473ms	6891ms	8.4%
-Capturing +Tiered	7038	5743	14.4%
+Capturing -Tiered	7885	6550	20.3%
+Capturing +Tiered	7638	5727	24%

ORACLE

Performance example – capture cost

- Oracle Performance Team measured capture costs (ops / uSec)
  - 4 socket x 10 core x 2 thread Nehalem EX server
- Worst-case lambda numbers equal to inner classes
  - Best-case numbers much better
  - And this is just our V1 strategy…

	Single-threaded	Saturated	Scalability
Inner class	150	750	5x
Non-capturing lambda	230	15500	67x
Capturing lambda	150	740	5x

Not just for the Java Language!

- The lambda conversion metafactories will be part of java.lang.invoke
  - Semantics tailored to Java language needs
  - But, other languages may find it useful too!
- Java APIs will be full of functional interfaces
  - Collection.forEach(Consumer)
- Other languages probably will want to call these APIs
  - Maybe using their own closures
  - Will want a similar conversion
- Since metafactories are likely to receive future VM optimization attention, using platform runtime is likely to be faster than spinning your own inner classes

ORACLE

Future VM support (?)

- With VM help, we can optimize even further
- VM could intrinsify lambda capture sites
  - Capture semantics are straightforward properties of method handles
  - Capture operation is pure, therefore freely reorderable
  - Can use code motion to delay/eliminate captures
- Lambda capture is essentially a “boxing” operation
  - Boxing a method handle into a lambda object
  - Invocation is the corresponding “unbox”
  - Can use box elimination techniques to eliminate capture overhead
    - Intrinsification of capture + inline + escape analysis

Serialization

- No language feature is complete without some interaction with serialization ☹
  – Users will expect this code to work

```
interface Foo extends Serializable {
    public boolean eval();
}
Foo f = () -> false;
// now serialize f
```

- We can’t just serialize the lambda object
  – Implementing class won’t exist at deserialization time
  – Deserializing VM may use a different translation strategy
  – Need a dynamic serialization strategy too!
    - Without exposing security holes…

Serialization

- Just as our classfile representation for a lambda is a recipe, our serialized representation needs to be too
  - We can use readResolve / writeReplace
  - Instead of serializing lambda directly, serialize the recipe (to a java.lang.invoke.SerializedLambda)
  - This means that for serializable lambdas, MF must provide a way of getting at the recipe
  - We provide an alternate MF bootstrap for that
- On deserialization, reconstitute from recipe
  - Using then-current translation strategy, which might be different from the one that originally created the lambda
  - Without opening new security holes

Summary

- The evolutionary path is often full of obvious-but-wrong ideas
- We use invokedynamic to capture lambda expressions
  – Gives us flexibility *and* performance
  – Free to change translation strategy at runtime
- Even using the “dumb” translation strategy…
  – No worse than inner classes in the worst case
  – 5-20x better than inner classes in a common case

ORACLE