Implementing lambda expressions in Java

Brian Goetz
Java Language Architect
Adding lambda expressions to Java

• In adding lambda expressions to Java, the obvious question is: what is the type of a lambda expression?
  • Most languages with lambda expressions have some notion of a function type in their type system
    • Java has no concept of function type
    • JVM has no native (unerased) representation of function type in type signatures
• Adding function types would create many questions
  • How do we represent functions in VM type signatures?
  • How do we create instances of function-typed variables?
  • How do we deal with variance?
• Want to avoid significant VM changes
We could “just” use MethodHandle

• At first, this seems “obvious”
• Desugar lambdas expressions to methods, and represent as MethodHandles in signatures
  • But, like erasure on steroids
    • Can’t overload two methods that take differently “shaped” lambdas
    • Still would need to encode the erased type information somewhere
  • Is MH invocation performance competitive with bytecode invocation yet?
• Conflates binary interface with implementation
Functional interfaces

- Java has historically represented functions using single-method interfaces like Runnable
- So, let’s make things simple and just formalize that
  - Give them a name: “functional interfaces”
  - Always convert lambda expressions to instance of a functional interface

```java
interface Predicate<T> {
    boolean test(T x);
}
adults = people.filter(p -> p.getAge() >= 18);
```

- Compiler figures out the types – lambda is converted to Predicate<Person>
- How does the lambda instance get created?
- How do other languages participate in the lambda fun?
We could “just” use inner classes

- We could define that a lambda is “just” an inner class instance (where the compiler spins the inner class)
  - \( p \rightarrow p.age < k \) translates to
    ```java
    class Foo$1 implements Predicate<Person> {
        private final int $v0;
        Foo$1(int v0) { this.$v0 = v0; }
        public boolean test(Person p) {
            return p.age < $v0;
        }
    }
    ```
  - Capture == invoke constructor (new Foo$1(k))
  - One class per lambda expression – yuck
  - Would like to improve over inner classes
    - If we define things this way, we’re stuck with inner class behavior forever
    - Back to that “conflates binary representation with implementation” problem
Stepping back...

- We would like to use a binary interface that doesn’t commit us 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 recipe, rather than imperative code, for creating a lambda
  - Let the runtime execute that recipe however it deems best
  - And make it darned fast
  - Sounds like a job for invokedynamic!
Its not just for dynamic languages anymore

- Where's the dynamism here?
  - All the types involved are static
  - What is dynamic here is the code generation strategy
- We use indy to embed a *recipe* for constructing a lambda at the capture site
  - The capture site is call the *lambda factory*
  - Invoked with indy, returns a lambda object
    - The bootstrap method is called the *lambda metafactory*
    - Static arguments describe the behavior and target type
    - Dynamic arguments are captured variables (if any)
- At first capture, a translation strategy is chosen
- Subsequent captures bypass the (slow) linkage path
Desugaring lambdas to methods

First, we desugar the lambda to a method

- Signature matches functional interface method, plus captured arguments prepended
  - Captured arguments must be *effectively final*
- Simplest lambdas desugar to static methods, but some need access to receiver, and so are instance methods

```java
Predicate<Person> isAdult = p -> p.getAge() >= k;

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

- Lambda capture is implemented by an indy invocation
  - Static arguments describe target type, behavior
  - Dynamic arguments describe captured locals
  - Result is a lambda object

```java
Predicate<Person> isAdult = p -> p.getAge() >= k;

isAdult = indy[bootstrap=LambdaMetafactory,
    type=MH[Predicate.test],
    impl=MH[lambda$1]](k);
```
The metafactory API

• Lambda metafactory looks like:

```java
metaFactory(Lookup caller,           // provided by VM
            String invokedName,   // provided by VM
            MethodType invokedType, // provided by VM
            MethodHandle target,   // target type
            MethodHandle body)     // lambda body
```

• Use method handles to describe both target name/type descriptor and implementation behavior
  • Metafactory semantics deliberately kept simple to enable VM intrinsification
  • “Link methods of target type to body.insertArgs(dynArgs).asType(target.type())”
Candidate 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
    - Since dynamic args and ctor arg 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
  - Use ClassValue to cache wrapper for interface
- Could also use dynamic proxies or MethodHandleProxy
- Or VM-private APIs to build object from scratch, or…
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
  - People sometimes 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 overhead if lambda is never used
  - No extra field or static initializer
  - All stateless lambdas get lazy init and caching for free
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
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
    - MF links to `new CCS(MethodHandles.constant(...))`
  - For capturing case, subsequent capture cost on order of a constructor call / method handle manipulation
    - MF links to constructor for generated class
Performance costs

- Any translation scheme imposes costs at several levels:
  - 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 correspond to:
  - Linkage: loading the class
  - Capture: invoking the constructor
  - Invocation: invokeinterface
Performance example – capture cost

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

<table>
<thead>
<tr>
<th></th>
<th>Single-threaded</th>
<th>Saturated</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner class</td>
<td>160</td>
<td>1407</td>
<td>8.8x</td>
</tr>
<tr>
<td>Non-capturing</td>
<td>636</td>
<td>23201</td>
<td>36.4x</td>
</tr>
<tr>
<td>lambda</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capturing lambda</td>
<td>160</td>
<td>1400</td>
<td>8.8x</td>
</tr>
</tbody>
</table>
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.filter(Predicate)`
- 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
Possible VM support

• VM can 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 like a “boxing” operation
  • Essentially boxing a method handle into lambda object
  • Invocation is the corresponding “unbox”
  • Can use box elimination techniques to eliminate capture overhead
    • Intrinsicsification of capture + inline + escape analysis
Serialization

• No language feature is complete without some interaction with serialization 😞
  • Users will expect this code to work

```java
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 (say, to some well defined interface 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
  • See paper for details
Serialization

• We record which class captured a lambda
  • And hand the recipe back to that class for reconstitution
  • Eliminating need for privileged magic in metafactory

```java
private static $deserialize$(SerializableLambda lambda) {
    switch (lambda.getImplName()) {
        case "lambda$1":
            if (lambda.getSamClass().equals("com/foo/SerializableComparator")
                && lambda.getSamMethodName().equals("compare")
                && lambda.getSamMethodDesc().equals("...")
                && lambda.getImplReferenceKind() == REF_invokeStatic
                && lambda.getImplClass().equals("com/foo/Foo")
                && lambda.getImplDesc().equals(...)
                && lambda.getInvocationDesc().equals(...))
                return indy(MH(serializableMetafactory),
                              MH(invokeVirtual
                              SerializableComparator.compare),
                              MH(invokeStatic lambda$1))
            (lambda.getCapturedArgs());
            break;
    }
    ...
```
My VM wish-list

- Intrinsification of functional interface conversion
- Better support for functional data structures
  - When we translate a typical filter-map-reduce chain, we create an expression tree whose leaves are lambdas
  - Use of Indy allows us to turn the leaves into constants
  - But we’d like to be able to turn the intermediate nodes into constants too!
    - Often practical, because these are value classes
    - Very common pattern in functional languages
    - I’ll take the leaves, but I’d rather have the whole tree
- Control over whether CallSite state is shared or cleared on cloning / inlining
  - Sometimes I want yes, sometimes I want no
  - One-size-fits-all not good enough