JavaOne™

ORACLE®

Unleashing Lambdas in a Distributed System

Brian Oliver
Architect | Specification Lead
Oracle Corporation | Coherence Engineering
October 2015

Email: brian.oliver@oracle.com
Twitter: @pinocchiocode

Safe Harbor Statement

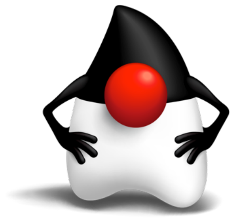
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development,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

Program Agenda

1. Lambdas
2. Remote Functional Interfaces & Lambdas
3. Demonstrations!
4. The Challenges and Limitations
5. Next Steps

Introduction to Lambdas

Introduction to Lambdas



A defining new feature of the Java 8 Platform

- Enable hybrid, object-oriented / functional programming in Java
- Allow you to pass code-as-data
  - **Both** as arguments and as return values
- Extensive enhancement of existing Java libraries to support them

```java
Map<String, String> candidates = new LinkedHashMap<>();

candidates.put("HC", "Hillary Clinton");
candidates.put(”DT", ”Donald Trump");
candidates.put(”CP", ”Cameron Purdy");

candidates.forEach((k, v) -> System.out.printf("key: %s, value: %s\n", k, v));
```

Introduction to Lambdas

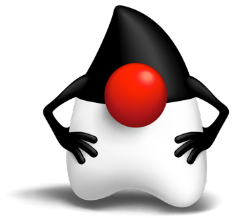
Automatically Capture Surrounding "effectively final" Context = Closures!

- Closures = function + environment

```java
Party party = Party.DEMOCRATIC;

Map<String, String> candidates = new LinkedHashMap<>();

candidates.put("HC", "Hillary Clinton");
candidates.put("MO", "Martin O'Malley");
candidates.put("BS", "Bernie Sanders");

candidates.forEach((k, v) -> System.out.printf(“%s (%s)\n", v, party));
```

Introduction to Lambdas

Replace anonymous "functional interface" inner-classes with lambdas



- Before:

```
executor.submit(new Runnable()
    {
    public void run()
        {
        System.out.println("Hello from anonymous class!");
        }
    });
```

- After:

```
executor.submit(() -> System.out.println("Hello from lambda!")
```

Introduction to Lambdas

An example functional interface

- Runnable Interface in Java 8:

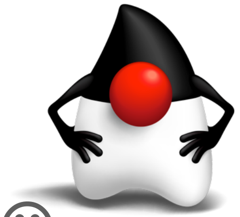
```java
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     */
    public void run();
}
```

But…

Can they be distributed and invoked across devices, machines, data-centers… cloud?

Introduction to Lambdas

Standard Functional Interfaces and thus Lambdas are not serializable by default ☹



- We can cast them though…

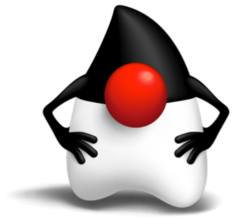
```
Runnable r = (Runnable & Serializable)
             () -> System.out.println("Hello from lambda!");
```

- Output*:

```
SerializedLambda[capturingClass=class JavaExamples,
             functionalInterfaceMethod=java/lang/Runnable.run:()V,
             implementation=invokeStatic JavaExamples.lambda$serialization$2feeadd5$1:()V,
             instantiatedMethodType=()V, numCaptured=0]
```

Introduction to Lambdas

None of the new functional interfaces are Serializable ☹



- There are many new *functional interfaces* in Java 8:
  - java.util.function.**Function**<T, R>
  - java.util.function.**Predicate**<T>
  - java.util.function.**Supplier**<T>
  - java.util.function.**Consumer**<T>
  - java.util.function.**BiConsumer**<T, U>
  - java.util.function.**UnaryOperator**<T>
  - java.util.function.**BinaryOperator**<T>
  - … and their primitive variants

Introduction to Lambdas

None of the existing functional interfaces are Serializable ☹

- And some of our old friends are also *functional interfaces*:
  - java.lang.**Runnable**
  - java.util.concurrent.**Callable**<V>
  - java.util.**Comparator**<T>

- **Perhaps all unusable in a distributed environment?**

Remote Functional Interfaces

Remote Functional Interfaces

Extensions of existing functional interfaces to support Serialization ☺

- Eg: The Coherence Remote Class defines serializable functional interfaces

```
public class Remote
    {
    @FunctionalInterface
    public interface Function<T, R>
            extends java.util.function.Function<T, R>, Serializable {}

    @FunctionalInterface
    public interface Runnable
            extends java.lang.Runnable, Serializable {}

    ...
    }
```

What is Coherence?

Remote Functional Interfaces

Most specific methods win!

- In addition to the standard Map method:

```
V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);
```

- Coherence NamedCache also defines:

```
V computeIfAbsent(K key, Remote.Function<? super K, ? extends V> mappingFunction);
```

- Java Compiler resolves to use the “most specific overloaded method”… so we’re ready to do some distributed lambdas!

An enormous number of demos!

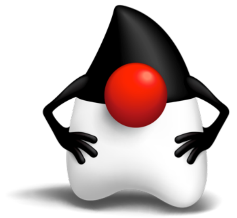
Challenges and Limitations

Oh!

“Serialization: The gift that keeps on giving”

Brian Goetz
Java Language Architect

Challenge: Lambda Serialization



Java provides the bare minimum… in a distributed environment we need a lot more

- Developers use multiple types of serialization
  - Java only provides one
- Lambdas need to be stable across application versions
  - Java does not provide any such guarantees, it's weak at best
- Developers want to introduce new lambdas without restarting
  - Java expects the same version of the capturing class to exist everywhere
- Coherence provides a custom and yet completely compatible remoting framework to solve these challenges

Limitation: Closure Serialization

Lambdas need to be self contained and serializable

- Should not reference fields or methods of the capturing class
- Should not reference anything that isn’t certain to exist both on the client and on the server
- Should only capture local serializable variables (use a static factory!)

```
public static Remote.BiFunction<String, Candidate, Candidate> changeToFunction(Party party)
    {
    return (key, candidate) ->
        {
        candidate.setParty(party);
        return candidate;
        };
    }
}
```

Limitation: Closure Serialization

Lambdas need to be self contained and serializable

- Should not nest other non-serializable lambdas
- Don't do this:

```
Map<String, String> names = candidates.invokeAll((entry) -> entry.extract(Candidate::getName));
```

- Instead do this:

```
ValueExtractor<Candidate, String> extractor = Candidate::getName;

Map<String, String> names = candidates.invokeAll((entry) -> entry.extract(extractor));
```

Summary

Summary

Distributed Lambdas Rock! Imagine the possibilities!

- Lambdas are a defining feature of Java 8
- Coherence 12.2.1 allows you to use lambdas
  - Like standard Java, but both locally & in a distributed manner
  - Allows in-place update without locking / synchronization
  - With existing Coherence features (like Entry Processors, Listeners…)
  - To perform stream-based operations
- Coherence adds support for serialization of standard functional interfaces
- Coherence handles distributed stream & lambdas in a dynamic way
  - Supports multiple versions of clients seamlessly running side-by-side without restart

Next Steps

Start Playing!

Coherence for Developers!

- https://www.oracle.com/goto/coherence
- https://coherence.java.net

https://twitter.com/OracleCoherence

https://www.linkedin.com/grp/home?gid=1782166

https://blogs.oracle.com/OracleCoherence

http://www.youtube.com/OracleCoherence

CREATE THE FUTURE

Note: The speaker notes for this slide include detailed instructions on how to reuse this Section Header slide in another presentation.

Tip! Remember to remove this text box.

Safe Harbor Statement

Integrated Cloud

Applications & Platform Services

JavaOne™

ORACLE®