Java™

Lambdas & Streams: Taking the Hard Work Out of Bulk Operations in Java SE 8

Simon Ritter
Head of Java Evangelism
Oracle Corporation

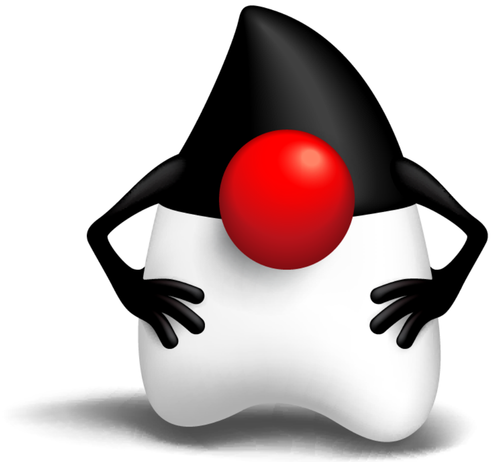
Twitter: @speakjava

MAKE THE FUTURE JAVA

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.

Lambdas In Java



The Problem: External Iteration

```
List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.gradYear == 2011) {
    if (s.score > highestScore) {
      highestScore = s.score;
    }
  }
}
```

- Client controls iteration
- *Inherently serial:* iterate from beginning to end
- Not thread-safe because business logic is stateful (mutable accumulator variable)

Internal Iteration With Inner Classes

More Functional, Fluent

```
List<Student> students = ...
double highestScore =
  students.filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011;
    }
  }).map(new Mapper<Student,Double>() {
    public Double extract(Student s) {
      return s.getScore();
    }
  }).max();
```

- Iteration, filtering and accumulation are handled by the library
- Not inherently serial – traversal *may* be done in parallel
- Traversal *may* be done lazily – so one pass, rather than three
- Thread safe – client logic is stateless
- High barrier to use
  – Syntactically ugly

 |

Java™ ORACLE®

Internal Iteration With Lambdas

```
SomeList<Student> students = ...
double highestScore =
  students.stream()
          .filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

- More readable
- More abstract
- Less error-prone
- No reliance on mutable state
- Easier to make parallel

Lambda Expressions

Some Details

- Lambda expressions represent anonymous functions
  - Like a method, has a typed argument list, a return type, a set of thrown exceptions, and a body
  - Not associated with a class
- We now have parameterised behaviour, not just values

```
double highestScore =
  students.stream()
          .filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

What

How

Lambda Expression Types

- Single-method interfaces are used extensively in Java
  - Functions and callbacks
  - Definition: a *functional interface* is an interface with one abstract method
  - *Functional interfaces* are identified structurally
  - The type of a lambda expression will be a *functional interface*

```
interface Comparator<T>     { boolean compare(T x, T y); }
interface FileFilter        { boolean accept(File x); }
interface Runnable          { void run(); }
interface ActionListener    { void actionPerformed(…); }
interface Callable<T>       { T call(); }
```


Target Typing

- A lambda expression is a way to create an instance of a functional interface
  - Which functional interface is inferred from the context
  - Works both in assignment and method invocation contexts

```
sort(myList, (String x, String y) -> x.length() – y.length());
```

```
Comparator<String> c = (String x, String y) -> x.length() - y.length();
```

  - Be careful, remember signature of functional interface

```
addActionListener((ae) -> System.out.println("Got it!"));
```

Local Variable Capture

- Lambda expressions can refer to *effectively final* local variables from the enclosing scope
  - Effectively final means that the variable meets the requirements for final variables (i.e., assigned once), even if not explicitly declared final
  - This is a form of type inference

```
void expire(File root, long before) {
    root.listFiles(File p -> p.lastModified() <= before);
}
```

 |

Lexical Scoping

- The meaning of names are the same inside the lambda as outside
  - A ‘this’ reference – refers to the enclosing object, not the lambda itself
  - Think of ‘this’ as a final predefined local
  - Remember the type of a Lambda is a *functional interface*

```
class SessionManager {
  long before = ...;

  void expire(File root) {
     // refers to 'this.before', just like outside the lambda
     root.listFiles(File p -> checkExpiry(p.lastModified(), this.before));
  }

  boolean checkExpiry(long time, long expiry) { ... }
}
```


Type Inferrence

- The compiler can often infer parameter types in a lambda expression
- Inferrence based on the target functional interface's method signature
- Fully statically typed (no dynamic typing sneaking in)
  - More typing with less typing

```
static T void sort(List<T> l, Comparator<? super T> c);
```

```
List<String> ls = getList();
Collections.sort(ls, (String x, String y) -> x.length() - y.length());
```

↓

```
Collections.sort(ls, (x, y) -> x.length() - y.length());
```


Method References

- Method references let us reuse a method as a lambda expression

```
FileFilter x = f -> f.canRead();
```

```
FileFilter x = File::canRead;
```


Constructor References

```
interface Factory<T> {
  T make();
}
```

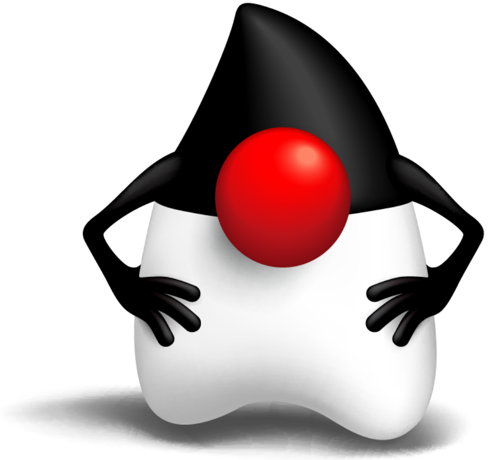
```
Factory<List<String>> f = ArrayList<String>::new;
```

Equivalent to

```
Factory<List<String>> f = () -> return new ArrayList<String>();
```

- When `f.make()` is invoked it will return a `new ArrayList<String>`

Library Evolution



Library Evolution

The Real Challenge

- Adding lambda expressions is a big language change
  - If Java had them from day one, the APIs would definitely look different
- Most important APIs (Collections) are based on interfaces
  - How to extend an interface without breaking backwards compatability?
- Adding lambda expressions to Java, but not upgrading the APIs to use them, would be silly
- Therefore we also need better mechanisms for *library evolution*

Library Evolution Goal

- Requirement: aggregate operations on collections
  - New methods required on Collections to facilitate this

```
int heaviestBlueBlock =
    blocks.stream()
          .filter(b -> b.getColor() == BLUE)
          .map(Block::getWeight)
          .reduce(0, Integer::max);
```

- This is problematic
  - Can’t add new methods to interfaces without modifying all implementations
  - Can’t necessarily find or control all implementations

Solution: Extension Methods

AKA Defender Methods

- Specified in the interface
- From the caller's perspective, just an ordinary interface method
- Provides a default implementation
  - Default is only used when implementation classes do not provide a body for the extension method
  - Implementation classes can provide a better version, or not

```
interface Collection<E> {
  default Stream<E> stream() {
    return StreamSupport.stream(spliterator());
  }
}
```

Virtual Extension Methods

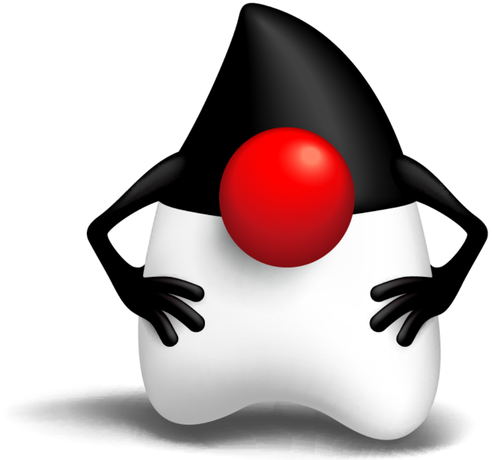
Stop right there!

- Err, isn't this implementing multiple inheritance for Java?
  - Yes, but Java already has multiple inheritance of *types*
  - This adds multiple inheritance of *behavior* too
  - But not *state*, which is where most of the trouble is
  - Can still be a source of complexity due to separate compilation and dynamic linking
    - Class implements two interfaces, both of which have default methods
    - Same signature
    - How does the compiler differentiate?

Functional Interface Definition

- Single Abstract Method (SAM) type
- A functional interface is an interface that has one abstract method
  - Represents a single function contract
  - Doesn't mean it only has one method
- Abstract classes may be considered later
- `@FunctionalInterface` annotation
  - Helps ensure the functional interface contract is honoured
  - Compiler error if not a SAM

Lambdas In Full Flow: Streams



Aggregate Operations

- Most business logic is about aggregate operations
  - Most profitable product by region
  - Group transactions by currency
- As we have seen, up to now, Java uses external iteration
  - Inherently serial
  - Frustratingly imperative
- Java SE 8's answer: **`Streams`**
  - With help from Lambdas

Stream Overview

At The High Level

- Abstraction for specifying aggregate computations
  - Not a data structure
  - Can be infinite
- Simplifies the description of aggregate computations
  - Exposes opportunitires for optimisation
  - Fusing, laziness and parrallelism

Stream Overview

Pipeline

- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect

```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals(“London”)).
  mapToInt(Transaction::getPrice).
  sum();
```

Source

Intermediate operation

Terminal operation

Stream Overview

Execution

- The **`filter`** and **`map`** methods don't really do any work
  - Set up a pipeline of operations and return a new **`Stream`**
- All work happens when we get to the **`sum()`** operation
  - **`filter()/map()/sum()`** fused into one pass on the data
    - For both sequential and parallel pipelines

```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals("London")).  // Lazy
  mapToInt(Transaction::getPrice).                        // Lazy
  sum();                                   // Execute the pipeline
```


Java™ ORACLE

Stream Sources

Many Ways To Create

- From collections and arrays
  - `Collection.stream()`
  - `Collection.parallelStream()`
  - `Arrays.stream(T array)` or `Stream.of()`
- Static factories
  - `IntStream.range()`
  - `Files.walk()`
- Roll your own
  - `java.util.Spliterator()`

Stream Sources Provide

- Access to stream elements
- Decomposition (for parallel operations)
  - Fork-join framework
- Stream characteristics
  - `ORDERED`
  - `DISTINCT`
  - `SORTED`
  - `SIZED`
  - `SUBSIZED`
  - `NONNULL`
  - `IMMUTABLE`
  - `CONCURRENT`

Stream Intermediate Operations

- Can affect pipeline characteristics
  - `map()` preserves `SIZED` but not necessarily `DISTINCT` or `SORTED`
- Some operations fuse/convert to parallel better than others
  - Stateless operations (`map`, `filter`) fuse/convert perfectly
  - Stateful operations (`sorted`, `distint`, `limit`) fuse/convert to varying degrees

Stream Terminal Operations

- Invoking a terminal operation executes the pipeline
  - All operations can execute sequentially or in parallel
- Terminal operations can take advantage of pipeline characteristics
  - `toArray()` can avoid copying for `SIZED` pipelines by allocating in advance

java.util.function Package

- **`Predicate<T>`**
  - Determine if the input of type T matches some criteria
- **`Consumer<T>`**
  - Accept a single input argumentof type T, and return no result
- **`Function<T, R>`**
  - Apply a function to the input type T, generating a result of type R
- Plus several more

The iterable Interface

Used by most collections

- One method, **`forEach()`**
  - Parameter is a **`Consumer`**
- Replace with **`reduce`** or **`collect`** where possible
  - **`forEach`** is not thread safe, and cannot be made parallel

```
wordList.forEach(System.out::println);  // OK

List<T> l = ...
s.map(λ).forEach(e -> l.add(e));
```

Replace with

```
List<T> l = s.map(λ).collect(Collectors.toList());
```

 |

Maps and FlatMaps

Map Values in a Stream

- One-to-one mapping
  - `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
  - `mapToDouble`, `mapToInt`, `mapToLong`
- One-to-many mapping
  - `<R> Stream<R> flatMap(`
    `Function<? super T, ? extends Stream<? extends R> mapper)`
  - `flatMapToDouble`, `flatMapToInt`, `flatMapToLong`

Example 1

Convert words in list to upper case

```
List<String> output = wordList.
  stream().
  map(String::toUpperCase).
  collect(Collectors.toList());
```


Example 2

Find words in list with even length

```
List<String> output = wordList.
  stream().
  filter(w -> (w.length() & 1 == 0).
  collect(Collectors.toList());
```


Example 3

Count lines in a file

- BufferedReader has new method
  - `Stream<String> lines()`

```
long count = bufferedReader.
  lines().
  count();
```

Example 4

Join lines 3-4 into a single string

```
String output = bufferedReader.
  lines().
  skip(2).
  limit(2).
  collect(Collectors.joining());
```


Example 5

Find the length of the longest line in a file

```
int longest = reader.
  lines().
  mapToInt(String::length).
  max().
  getAsInt();
```

Example 6

Collect all words in a file into a list

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(Collectors.toList());
```


Example 7

List of words lowercased, in aphabetical order

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  map(String::toLowerCase).
  sorted().
  collect(Collectors.toList());
```


Conclusions

- Java needs lambda statements
  - Significant improvements in existing libraries are required
- Require a mechanism for interface evolution
  - Solution: virtual extension methods
- Bulk operations on Collections
  - Much simpler with Lambdas
- Java SE 8 evolves the language, libraries, and VM together

 |

Java™