Collections After Eight

Maurice Naftalin
Morningside Light Ltd.
@mauricenaftalin
Maurice Naftalin

Developer, designer, architect, teacher, learner, writer
Maurice Naftalin
Developer, designer, architect, teacher, learner, writer

Co-author
Maurice Naftalin

Developer, designer, architect, teacher, learner, writer

Co-author

Current Project

www.lambdafaq.org
First computer I programmed

Varian 620/i

Fast operation: 1.8-microsecond memory cycle.
Large instruction repertoire: 107 standard, 18 optional; with approximately 200 additional instruction configurations which can be microcoded.
Word length: 16- or 18-bit configurations.
Modular memory: 4096 word minimum, 32,768 words maximum.
Why is everyone so rude about us?
Why is everyone so rude about us?

- “90% of all Java programs are written by morons.”
Why is everyone so rude about us?

- “90% of all Java programs are written by morons.”
- “If Java had true garbage collection, most programs would delete themselves upon execution.”
Why is everyone so rude about us?

- “90% of all Java programs are written by morons.”
- “If Java had true garbage collection, most programs would delete themselves upon execution.”
- “Lambdas aren’t a new invention so why did it take Java so long to incorporate them? IIRC even COBOL had something like them.”
Why is everyone so rude about us?

• “90% of all Java programs are written by morons.”
• “If Java had true garbage collection, most programs would delete themselves upon execution.”
• “Lambdas aren’t a new invention so why did it take Java so long to incorporate them? IIRC even COBOL had something like them.”
• “Sufficiently advanced Java is indistinguishable from satire.”
Why is everyone so rude about us?

- “90% of all Java programs are written by morons.”
- “If Java had true garbage collection, most programs would delete themselves upon execution.”
- “Lambdas aren’t a new invention so why did it take Java so long to incorporate them? IIRC even COBOL had something like them.”
- “Sufficiently advanced Java is indistinguishable from satire.”
- “Java is the new Cobol”
Is it because we write code like this?

- Code plucked from a personal project

```java
// check each deadlined Plan. If it can’t be done in time for its deadline, return the Plan
// that caused the problem; otherwise, return null.

private Plan checkPlans(List<Plan> deadlinedPlans) {
    Plan problemPlan = null;
    Iterator<Plan> planItr = deadlinedPlans.iterator();
    while (planItr.hasNext() && problemPlan == null) {
        Plan plan = planItr.next();
        problemPlan = checkPlan(plan);
    }
    return problemPlan;
}
```
private Plan checkPlans(List<Plan> deadlinedPlans) {
    Plan problemPlan = null;
    Iterator<Plan> planItr = deadlinedPlans.iterator();
    while (planItr.hasNext() && problemPlan == null) {
        Plan plan = planItr.next();
        problemPlan = checkPlan(plan);
    }
    return problemPlan;
}
Could we improve our image?
Could we improve our image?

Wouldn’t it be cool if instead of writing this:

```java
class CheckPlans {
    private Plan checkPlans(List<Plan> deadlinePlans) {
        Plan problemPlan = null;
        Iterator<Plan> planItr = deadlinePlans.iterator();
        while (planItr.hasNext() && problemPlan == null) {
            Plan plan = planItr.next();
            problemPlan = checkPlan(plan);
        }
        return problemPlan;
    }
}
```
Could we improve our image?

Wouldn’t it be cool if instead of writing this:

```java
private Plan checkPlans(List<Plan> deadlinedPlans) {
    Plan problemPlan = null;
    Iterator<Plan> planItr = deadlinedPlans.iterator();
    while (planItr.hasNext() && problemPlan == null) {
        Plan plan = planItr.next();
        problemPlan = checkPlan(plan);
    }
    return problemPlan;
}
```

we could write this:

```java
private Optional<Plan> checkPlans(List<Plan> deadlinedPlans) {
    return deadlinedPlans.stream().
        map(p -> checkPlan(p)).
        filter(p -> p != null).
        findFirst();
}
```
Could we improve our image?

Wouldn’t it be cool if instead of writing this:

```java
private Plan checkPlans(List<Plan> deadlinedPlans) {
    Plan problemPlan = null;
    Iterator<Plan> planItr = deadlinedPlans.iterator();
    while (planItr.hasNext() && problemPlan == null) {
        Plan plan = planItr.next();
        problemPlan = checkPlan(plan);
    }
    return problemPlan;
}
```

...we could write this:

```java
private Optional<Plan> checkPlans(List<Plan> deadlinedPlans) {
    return deadlinedPlans.stream().
        map(p -> checkPlan(p)).
        filter(p -> p != null).
        findFirst();
}
```

And easily make it execute in parallel??!
Presentation Overview

- The Magic Ingredient
- Result: Better APIs
- Result: More Parallelism
Take values to a higher order
Take values to a higher order

instead of supplying \textbf{values} to \textbf{specific} library methods

```java
public interface Collection<E> {
    ...
    boolean remove(Object o);
    ...
}
```
Take values to a higher order

instead of supplying values to specific library methods

```java
public interface Collection<E> {
    ...
    boolean remove(Object o);
    ...
}
```

we’re going to supply behaviour to general library methods:

```java
public interface Collection<E> {
    ...
    boolean removeIf(Predicate<? super E> p);
    ...
}
```
Take values to a higher order

instead of supplying values to specific library methods

```java
public interface Collection<E> {
    ...
    boolean remove(Object o);
    ...
}
```

we’re going to supply behaviour to general library methods:

```java
public interface Collection<E> {
    ...
    boolean removeIf(Predicate<? super E> p);
    ...
}
```
Take values to a higher order

instead of supplying **values** to **specific** library methods

```java
public interface Collection<E> {
    ...
    boolean remove(Object o);
    ...
}
```

we’re going to supply **behaviour** to **general** library methods:

```java
public interface Collection<E> {
    ...
    boolean removeIf(Predicate<? super E> p);
    ...
}
```

Predicate is an interface with a single abstract boolean-valued method `test`

`removeIf` executes `test` for each element:

- if `test` returns `true`, `removeIf` removes that element
Take values to a higher order

instead of supplying **values** to **specific** library methods

```java
public interface Collection<E> {
    ...
    boolean remove(Object o);
    ...
}
```

we’re going to supply **behaviour** to **general** library methods:

```java
public interface Collection<E> {
    ...
    boolean removeIf(Predicate<? super E> p);
    ...
}
```

Predicate is an interface with a single abstract boolean-valued method:

removeIf executes `test` for each element:

- if `test` returns `true`, `removeIf` removes that element
How to make an interface instance?

Using an anonymous inner class, of course!

```java
planList.removeIf(new Predicate<Plan>() {
    public boolean test(Plan p) {
        return p.equals(problemPlan);
    }
});
```
How to make an interface instance?

Using an anonymous inner class, of course!

```java
planList.removeIf(new Predicate<Plan>() {
    public boolean test(Plan p) {
        return p.equals(problemPlan);
    }
});
```

Maybe *this* is why they’re so rude about us?
Let’s strip out the boilerplate!

Why do we have to say we’re supplying a Predicate?
Why do we have to say we’re implementing test, the only abstract method?

```java
planList.removeIf(new Predicate<Plan>() {
    public boolean test(Plan p) {
        return p.equals(problemPlan)
    }
});
```
Let’s strip out the boilerplate!

Why do we have to say we’re supplying a Predicate?
Why do we have to say we’re implementing test, the only abstract method?

```java
planList.removeIf(new Predicate<Plan>(){
    public boolean test(Plan p) {
        return p.equals(problemPlan)
    }
});
```
Let’s strip out the boilerplate!

Why do we have to say we’re supplying a Predicate?
Why do we have to say we’re implementing test, the only abstract method?

```java
planList.removeIf(new Predicate<Plan>(){
  public boolean test(Plan p) {
    return p.equals(problemPlan)
  }
});
```
Let’s strip out the boilerplate!

Why do we have to say we’re supplying a Predicate?
Why do we have to say we’re implementing test, the only abstract method?

```java
planList.removeIf(new Predicate<Plan>() {
    public boolean test(Plan p) {
        return p.equals(problemPlan)
    }
});
```

```java
planList.removeIf(p -> p.equals(problemPlan));
```
Why do we have to say we’re supplying a Predicate?
Why do we have to say we’re implementing test, the only abstract method?

```
planList.removeIf(new Predicate<Plan>(){
    public boolean test(Plan p) {
        return p.equals(problemPlan)
    }
});
```

```
planList.removeIf(p -> p.equals(problemPlan));
```

Basically, lambdas are just better syntax!
Presentation Overview

• The Magic Ingredient
• Result: Better APIs
  • Pipes and Filters
  • More Finely-Grained APIs
• Result: More Parallelism
Pipes and Filters

- Venerable Unix tool-building pattern:
  ```bash
  ps -ef | grep login | cut -c 50- | head
  ```

- and in Enterprise Integration Patterns
Pipes and Filters

Advantages of this pattern
Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```
Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```

- no intermediate variables
Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```

- no intermediate variables
- less (or no) intermediate storage
Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```

- no intermediate variables
- less (or no) intermediate storage
- lazy evaluation
Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```

- no intermediate variables
- less (or no) intermediate storage
- lazy evaluation
- flexible tool-building:
Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```

- no intermediate variables
- less (or no) intermediate storage
- lazy evaluation
- flexible tool-building:

*Write programs that do one thing well.*
*Write programs to work together.*
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deallinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
Pipes & Filters in collection operations

deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
deadlinedPlans.stream().map(p->checkPlan(p)).filter(p->p!=null).findFirst();
"Write programs that do one thing well"

java.util.functions.Mapper:

```java
public interface Mapper<T, R> {
    R map(T t);
}
```

java.util.functions.Predicate

```java
public interface Predicate<T> {
    boolean test(T t);
}
```
What is a Stream?

- A sequence of values
- May be partially evaluated
- Execution of lazy operations sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinedPlans.stream().map(p->preparePlan(p));
```
What is a Stream?

- A sequence of values
- May be *partially evaluated*
- Execution of *lazy operations* sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinePlans.stream().map(p->preparePlan(p));
```

Collection
What is a Stream?

- A sequence of values
- May be *partially evaluated*
- Execution of *lazy operations* sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinePlans.stream().map(p -> preparePlan(p));
```

```java
Collection
MapStreamIterator(StreamIterator<Plan> upstream, Mapper m) {
    ...  
    Plan next() {
        return m.map(upstream.next());
    }
    ...
}
```
What is a Stream?

- A sequence of values
- May be *partially evaluated*
- Execution of *lazy operations* sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinedPlans.stream().map(p->preparePlan(p));
```
What is a Stream?

- A sequence of values
- May be *partially evaluated*
- Execution of *lazy operations* sets the pipeline up:

```java
Stream<Plan> planStream = deadlinedPlans.stream().map(p->preparePlan(p)).filter(p->p!=null);
```
What is a Stream?

- A sequence of values
- May be partially evaluated
- Execution of lazy operations sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinedPlans.stream().map(p->preparePlan(p)).filter(p->p!=null);
```

```java
FilterStreamIterator(AsStreamIterator<Plan> upstream, Predicate pred) {
    public Plan next() {
        Plan p = upstream.next();
        return pred.test(p) ? p : next(); // assume upstream.hasNext()
    }
    ...
}
```
**What is a Stream?**

- A sequence of values
- May be *partially evaluated*
- Execution of *lazy operations* sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinedPlans.stream().map(p->preparePlan(p)).filter(p->p!=null());
```

---

Collection

MapStreamIterator(StreamIterator<Plan> upstream, Mapper m) {
    Plan next() { return m.map(upstream.next()); }
}

FilterStreamIterator(StreamIterator<Plan> upstream, Predicate pred) {
    public Plan next() { 
        Plan p = upstream.next();
        return pred.test(p) ? p : next(); // assume upstream.hasNext()
    }
}
What is a Stream?

- A sequence of values
- May be *partially evaluated*
- Execution of *lazy operations* sets the pipeline up:

```java
Stream<Plan> planStream =
    deadlinedPlans.stream().map(p->preparePlan(p)).filter(p->p!=null);
```

No evaluation yet!
Lazy and eager operations

- Execution of lazy operations (IntermediateOps) sets the pipeline up
- Execution of *eager operations* (TerminalOps) pulls data down the pipeline

```java
Collection<Plan> planStream =
    deadlinedPlans.stream().map(p->preparePlan(p)).filter(p->p!=null);
planStream.findFirst();    // stop after the first element
```
Lazy and eager operations

- Execution of lazy operations (IntermediateOps) sets the pipeline up
- Execution of eager operations (TerminalOps) pulls data down the pipeline

```java
Stream<Plan> planStream =
deadlinedPlans.stream().map(p->preparePlan(p)).filter(p->p!=null);
planStream.findFirst();  // stop after the first element
```
Some Stream methods

// get the plans longer than 15 minutes
planStream.filter(p -> p.getDuration().isLongerThan(minutes(15)))

// get the total duration of all the plans in a list
planStream.map(p -> p.getDuration()).reduce(Duration.ZERO, (d1,d2) -> d1.plus(d2))

// each plan belongs to a task; map each task to a collection of its plans
planStream.groupBy(p -> p.getTask())

// get the first five plans
planStream.limit(10)

// get all but the first five plans
planStream.skip(10)
## Some Stream methods

<table>
<thead>
<tr>
<th>name</th>
<th>returns</th>
<th>interface used</th>
<th>λ signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>Stream&lt;T&gt;</td>
<td>lazy</td>
<td>Predicate&lt;T&gt;</td>
</tr>
<tr>
<td>map</td>
<td>Stream&lt;T&gt;</td>
<td>lazy</td>
<td>Predicate&lt;T&gt;</td>
</tr>
<tr>
<td>reduce</td>
<td>T</td>
<td>eager</td>
<td>BinaryOperator&lt;T&gt;</td>
</tr>
<tr>
<td>groupBy</td>
<td>Map&lt;U,Collection&lt;T&gt;&gt;</td>
<td>eager</td>
<td>Mapper&lt;T,U&gt;</td>
</tr>
<tr>
<td>limit,</td>
<td>Stream&lt;T&gt;</td>
<td>lazy</td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findFirst</td>
<td>T</td>
<td>eager</td>
<td>Predicate&lt;T&gt;</td>
</tr>
<tr>
<td>forEach</td>
<td>void</td>
<td>eager</td>
<td>Block&lt;T&gt;</td>
</tr>
<tr>
<td>sorted</td>
<td>Stream&lt;T&gt;</td>
<td>lazy</td>
<td>Comparator&lt;T&gt;</td>
</tr>
</tbody>
</table>
Presentation Overview

- The Magic Ingredient
- Result: Better APIs
  - Pipes and Filters
  - More Finely-Grained APIs
- Result: More Parallelism
Making finer-grained APIs

Sorting a list using Comparator:

```java
Collections.sort(planList, new Comparator<Plan>() {
    public int compare(Plan p1, Plan p2) {
        return p1.getTask().compareTo(p2.getTask());
    }
});
```

The method `compare` must first extract the sort keys and then compare them.
Making methods more precise

Suppose instead we have a method that accepts—and returns—behaviours:

```java
Comparator<T> comparing(Mapper<T,U> m);
```

Much easier now to create custom Comparators:

```java
Comparator<Plan> byTask = Comparators.comparing(p -> p.getTask());
Comparator<Plan> byDuration = Comparators.comparing(p -> p.getDuration());
```
Composing fine-grained methods

Mappers and Comparators can now be composed
  • new Comparator method

```java
Collections.sort(planList, byTask.compose(byDuration));
```

and now also

```java
planList.sort(byTask.compose(byDuration));
```
Composing fine-grained methods

Mappers and Comparators can now be composed

- new Comparator method

```java
Collections.sort(planList, byTask.compose(byDuration));
```

and now also

```java
planList.sort(byTask.compose(byDuration));
```

Plan::getDuration
Composing fine-grained methods

Mappers and Comparators can now be composed

- new Comparator method

```java
Collections.sort(planList, byTask.compose(byDuration));
```

and now also

```java
planList.sort(byTask.compose(byDuration));
```

Plan::getTask

Plan::getDuration
Presentation Overview

- The Magic Ingredient
- Result: Better APIs
- Result: More Parallelism
**External Iteration**

**Foreach loop:**

```java
for (Plan plan : planList) {
    plan.setSection(null);
}
```

**or, equivalently,**

```java
for (Iterator<Plan> pItr = planList.iterator(); pItr.hasNext(); ) {
    pItr.next().setSection(null);
}
```
External Iteration

Foreach loop:

```java
for (Plan plan : planList) {
    plan.setSection(null);
}
```

or, equivalently,

```java
for (Iterator<Plan> pltr = planList.iterator(); pltr.hasNext(); ) {
    pltr.next().setSection(null);
}
```

- Client controls the iteration
External Iteration

Foreach loop:

```java
for (Plan plan : planList) {
    plan.setSection(null);
}
```

or, equivalently,

```java
for (Iterator<Plan> pltr = planList.iterator(); pltr.hasNext(); ) {
    pltr.next().setSection(null);
}
```

- Client controls the iteration
- Loop is inherently sequential
External Iteration

Foreach loop:

```java
for (Plan plan : planList) {
    plan.setSection(null);
}
```

or, equivalently,

```java
for (Iterator<Plan> pItr = planList.iterator(); pItr.hasNext(); ) {
    pItr.next().setSection(null);
}
```

- Client controls the iteration
- Loop is inherently sequential
  - client would have to manage parallelisation
Internal iteration

Strategy:

Supply *behaviour* to *general* library methods
Internal iteration

Strategy:

Supply *behaviour to general library methods*

```java
planList.forEach(p -> p.setSection(null))
```
Strategy:

Supply *behaviour* to *general* library methods

```java
planList.forEach(p -> p.setSection(null))
```

Much simpler interaction.

And now the library controls the iteration:

- Can use its own iteration tactics:
  - Parallelism, out-of-order execution, laziness
Example: ConcurrentHashMap

ConcurrentHashMap.parallel(ForkJoinPool)

- Returns a parallel view of the CHM
- Methods: forEach, forEachEntry, forEachKey, forEachValue,...
- Consider forEachValue(ConcurrentHashMap.Action<V> action)
  - Creates a new ForEachValueTask
    - subclass of ForkJoinTask
    - submits that to the ForkJoinPool
  - ForkJoinPool executes ForEachValueTask on one of its threads
static final class ForEachValueTask<K,V> extends BulkTask<K,V,Void> {
    final Action<V> action;
    ForEachValueTask()
        (ConcurrentHashMap<K,V> m,
         Action<V> action) {
        super(m);
        this.action = action;
    }
    ForEachValueTask
        (BulkTask<K,V,?> p, int b, boolean split,
         Action<V> action) {
        super(p, b, split);
        this.action = action;
    }
    @SuppressWarnings("unchecked") public final void compute() {
        final Action<V> action = this.action;
        if (action == null)
            throw new Error(NullFunctionMessage);
        int b = batch(), c;
        while (b > 1 && baseIndex != baseLimit) {
            do {} while (!casPending(c = pending, c+1));
            new ForEachValueTask<K,V>(this, b >>>= 1, true, action).fork();
        }
        Object v;
        while ((v = advance()) != null)
            action.apply((V)v);
        tryComplete();
    }
}
static final class ForEachValueTask<K,V>
    extends BulkTask<K,V,VVoid> {
    final Action<V> action;
    ForEachValueTask
        (ConcurrentHashMap<K,V> m,
         Action<V> action) {
        super(m);
        this.action = action;
    }
    ForEachValueTask
        (BulkTask<K,V,V> p, int b, boolean split,
         Action<V> action) {
        super(p, b, split);
        this.action = action;
    }
    @SuppressWarnings("unchecked") public final void compute() {
        final Action<V> action = this.action;
        if (action == null)
            throw new Error(NullFunctionMessage);
        int b = batch(), c;
        while (b > 1 && baseIndex != baseLimit) {
            do {} while (!casPending(c = pending, c+1));
            new ForEachValueTask<K,V>(this, b >>>= 1, true, action).fork();
        }
        Object v;
        while ((v = advance()) != null)
            action.apply((V)v);
        tryComplete();
    }
@SuppressWarnings("unchecked") public final void compute() {
    final Action<V> action = this.action;
    if (action == null)
        throw new Error(NullFunctionMessage);
    int b = batch(), c;
    while (b > 1 && baseIndex != baseLimit) {
        do {} while (!casPending(c = pending, c+1));
        new ForEachValueTask<K,V>(this, b >>>= 1, true, action).fork();
    }
    Object v;
    while ((v = advance()) != null)
        action.apply((V)v);
    tryComplete();
}
Stream parallelisation

- Collection\(\text{parallel}()\).filter(...).map(...).reduce(...);
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);
Stream parallelisation

- `Collection.parallel().filter(...).map(...).reduce(...);`
Stream parallelisation

- Collection\(\text{parallel()}.\text{filter(...).map(...).reduce(...)};\)
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);
Stream parallelisation

- `Collection.parallel().filter(...).map(...).reduce(...);`
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);
Stream parallelisation

- Collection.parallel().filter(...).map(...).reduce(...);

![Diagram of stream parallelisation process](image)
Stream parallelisation

- Collection\(\text{parallel()}\).filter(...).map(...).reduce(...);
Conclusion

Lambdas seem like a small syntactic change, but—

• a big difference in the style of Java code
• set the library writers free to innovate
• encourage a functional coding style
• less mutability = more parallel-friendly
Conclusion

Lambdas seem like a small syntactic change, but—

- a big difference in the style of Java code
- set the library writers free to innovate
- encourage a functional coding style
- less mutability = more parallel-friendly
Resources

Lambda resources

This is a selective list of online documents and resources relevant to Project Lambda.

Documentation

The central reference for Project Lambda is the OpenJDK page. Its primary links are to:

- JSR 335 Early Draft Review 3. This consists mainly of the changes that the Java Language Specification will require for Project Lambda;
- State of the Lambda v3. This is an informal and readable introduction to the language features of Project Lambda, written by the project lead, but is becoming out-of-date as the project evolves. The material in this FAQ (but it will be soon);
- State of the Lambda: Libraries Edition. As with State of the Lambda, this is becoming out-of-date. Its content is not yet covered by this FAQ (but it will be soon);
- A full and informative description of the design of default methods (PDF) and its rationale;
- A formal model for default method linkage (PDF);
- The strategy for translating lambda expressions.

The Java Community Process has formal progress pages for JSR 335 and JEP 187 (ODK Enhancement Proposal for providing the collections library with bulk operations).

JDK build and download resources

JavaDoc and binaries for Windows, Mac, Linux and Solaris can be downloaded from http://jdk8.java.net/lambdas/. These are sometimes, but not always, up-to-date, if you need a more up-to-date version:

- daily binary builds for Linux can be found at http://osbuildfactory.hgopenjdk.net/
- instructions for building Mac binaries from source at https://github.com/hgopenjdk/osbuildfactory/wiki
- instructions for the official "old build" (all platforms)
- instructions for the official "old build" (all platforms, being retired but still useful to know about)

Presentations:

A web search reveals many slides decks presenting the features of Project Lambda. Because the project is work in progress, most of these contain details which would now be misleading. For this reason, only very recent presentations are listed here, and only if they are accompanied by sound or video.

- JavaOne 2012
  - The Road to Lambda (Brin Goetz) provides a broad and comprehensive view of Project Lambda.
  - Using Starting Lambda Programming (Stuart Marks) is a contrast—a gentle and painless introduction.
  - Lambda: A Peek Under the Hood (Brin Goetz) gives a wealth of technical detail about the implementation.

- Others
  - JavaZone 2012: Lambda in Java 8 (Angelika Landg)
  - Strange Loop 2012: Project Lambda in Java 8 (Daniel Smith) (link downloads PDF slides with notes, video available Dec. 24th)

A different kind of presentation is an interview with Brian Goetz—the Java Language Architect at the helm of Project Lambda—in the Java Magazine for September/October 2012 (other register (free) as a subscriber to download the magazine as PDF, or get it via the Newsstand app on iPhone or iPad).

Tool support

JavaBinaries have released GMP (early access program) version of IntelliJ IDEA, already providing quite good support for lambda expressions and other parts of the Java 8 feature set. They are currently available as free downloads.

- NetBeans 8 nightly builds provide experimental lambda support.

Mailing lists

For a long time, the principal open mailing list for discussing the Java 8 lambda-associated features was lambda-dev. The expert group lists were closed. In September 2012 a long-decided goal was achieved with the introduction of new open lists for the expert group discussions. The function of the new and changed mailing lists was explained in this post. In brief, they are these:

The list lambda-dev is for discussion of implementation issues, including bug reports, code review comments, last cases, build or porting problems, migration experiences, and so on. It should no longer be used for language or feature design discussions, though the archives are a useful source for past discussions of this kind. Comments and discussion about the issues not belong on the new lists described next. It should be said, though, that in fact language and feature discussions on lambda-dev were continuing at least up until early Nov 2012.
Resources

Lambda resources

This is a selected list of all documents and resources relevant to Project Lambda.

Documentation

The central reference for Project Lambda is the OpenJDK page. So far, the Lambda

www.lambdafaq.org/resources
Resources

Lambda resources

This is a selected list of online documents and resources relevant to Project Lambda.

Documentation

This is the central reference for Project Lambda in the Q2 2014 release. It contains the changes that the Java Language Specification will make for Project Lambda.

- The Java Language Specification, 1st Edition

www.lambdafaq.org/resources

JDK build and download resources

JDK 8 code is available in source form from https://github.com/adoptium/jdk. The build instructions can be found in the https://github.com/adoptium/jdk/blob/master/README.md file.

- JDK 8 code is available in source form from https://github.com/adoptium/jdk. The build instructions can be found in the https://github.com/adoptium/jdk/blob/master/README.md file.

Presentations

- Web cast: (unavailable)
- Web cast: (unavailable)
- Web cast: (unavailable)
- Web cast: (unavailable)
- Web cast: (unavailable)

Other

- OpenJDK 11: Lambda in Java (unavailable)
- OpenJDK 10: Lambda in Java (unavailable)
- OpenJDK 9: Lambda in Java (unavailable)
- OpenJDK 8: Lambda in Java (unavailable)

Test support

- Test scripts have been released for the latest version of JRE, including support for lambda expressions and other parts of the Java 9 feature set. They are available as a free download.

Mailing lists

- Java mailing list: available
- Lambda mailing list: available
- JavaFX mailing list: available
- Java Runtime Environment (JRE) mailing list: available
- Java Development Kit (JDK) mailing list: available
- Java Platform Module (JPM) mailing list: available
- Java Platform Module (JPM) mailing list: available
- Java Platform Module (JPM) mailing list: available
- Java Platform Module (JPM) mailing list: available
- Java Platform Module (JPM) mailing list: available
- Java Platform Module (JPM) mailing list: available

www.lambdafaq.org
Q&A