Collections After Eight

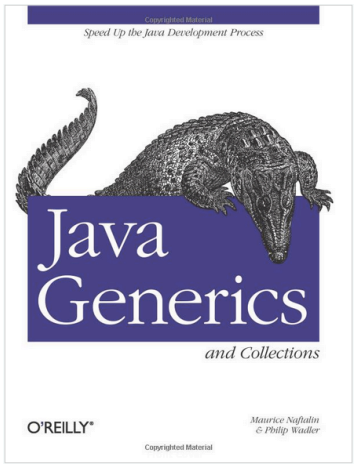
Maurice Naftalin

Morningside Light Ltd.

@mauricenaftalin

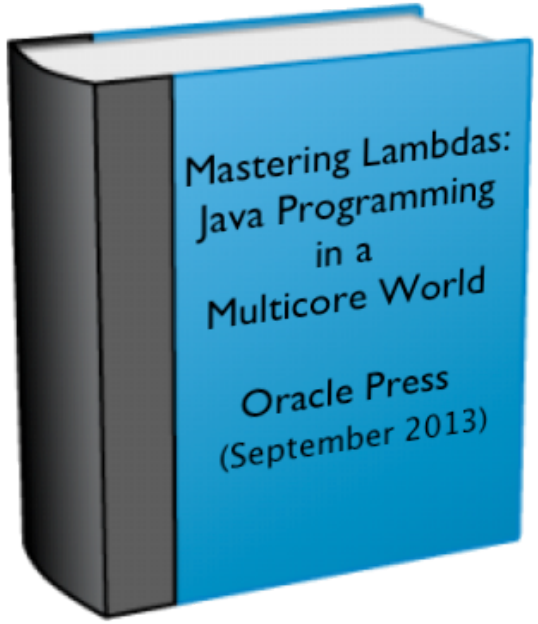
Maurice Naftalin

Developer, designer, architect, teacher, learner, writer



Co-author

www.lambdafaq.org



Current Projects

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

```
// 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;
 }
```

Could we improve our image?

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

```
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:

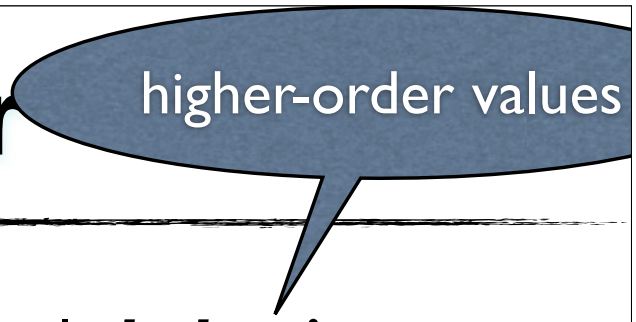
```
private Optional<Plan> checkPlans(
                        List<Plan> deadlinedPlans) {

    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



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

```
public interface Collection<E> {
    ...
    boolean removeAll(Collection<?> c);
    ...
}
```

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

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

`Predicate` is an interface with a single abstract boolean-valued met[illegible]

`removeAll` executes `test` for each element:

- if `test` returns `true`, `removeAll` removes that element

How to make an interface instance?

Using an anonymous inner class, of course!

```
planList.removeAll(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!

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

```
planList.removeAll(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?

Why do we have to say the type parameter is `Plan`?

So, lambdas are *at least* better syntax!

Presentation Overview

- The Magic Ingredient
- Result: Better APIs
  - Internal Iteration
  - Pipes and Filters
  - Composing Behaviours
- Result: More Parallelism

External Iteration

Standard Java loop forms:

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

==

```
Iterator<Plan> planItr = planList.iterator();
while (planItr.hasNext()) {
    planItr.next().setSection(null);
}
```

Problems:

- Client has to manage iteration—more complex than it looks
- Loop is inherently sequential
  - client would have to manage parallelisation

Internal iteration

Strategy:

> Supply *behaviour* to *general* library methods

```
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

Presentation Overview

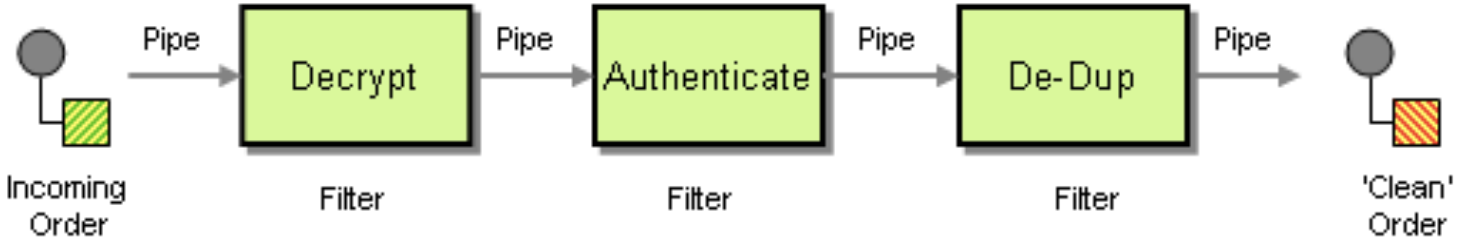
- The Magic Ingredient
- Result: Better APIs
  - Internal Iteration
  - Pipes and Filters
  - Composing Behaviours
- Result: More Parallelism

Pipes and Filters

- Venerable Unix tool-building pattern:

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

- and in Enterprise Integration Patterns



Pipes and Filters

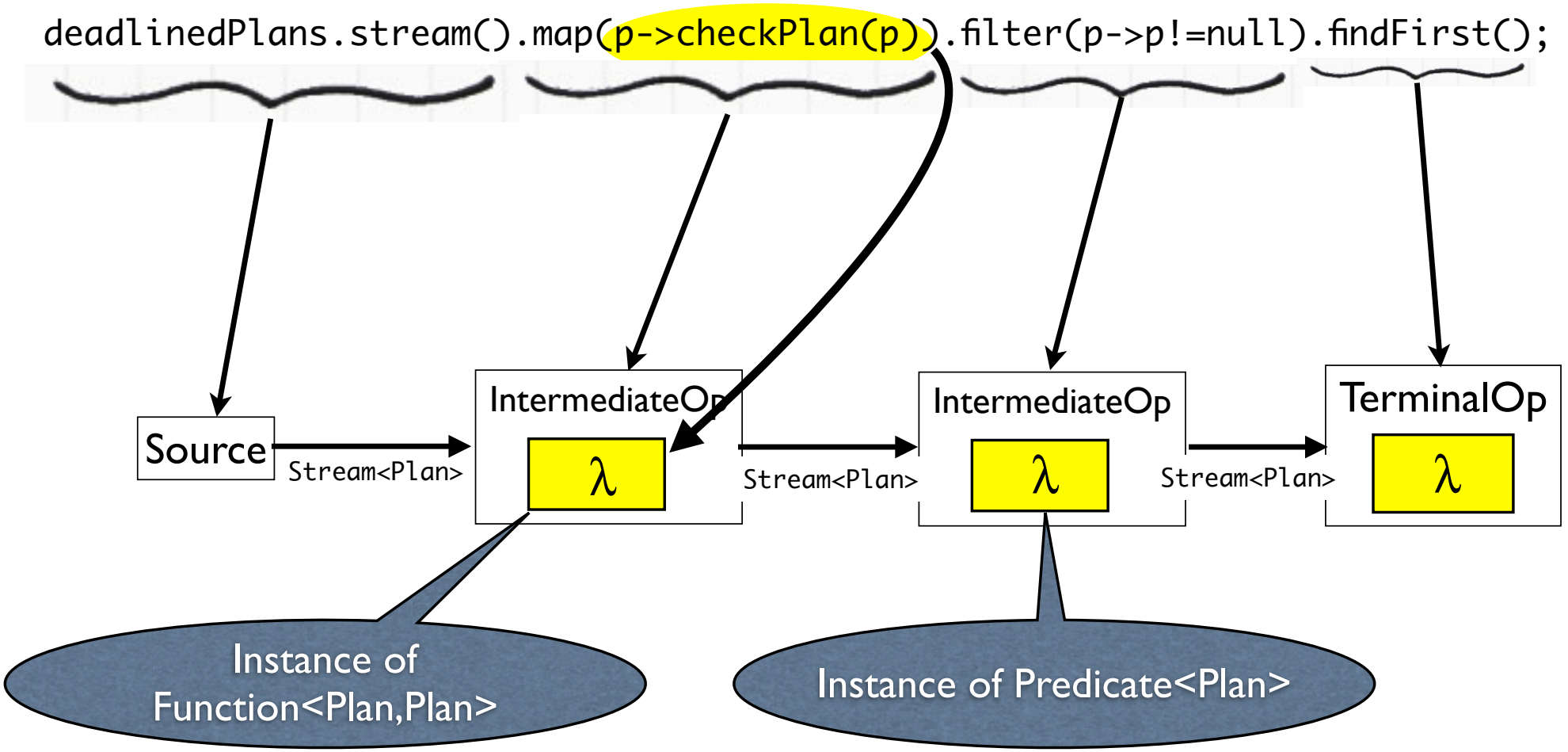
Advantages of this pattern

```
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



“Write programs that do one thing well”

`java.util.functions.Function:`

```
public interface Function<T, R> {
    R apply(T t);
}
```

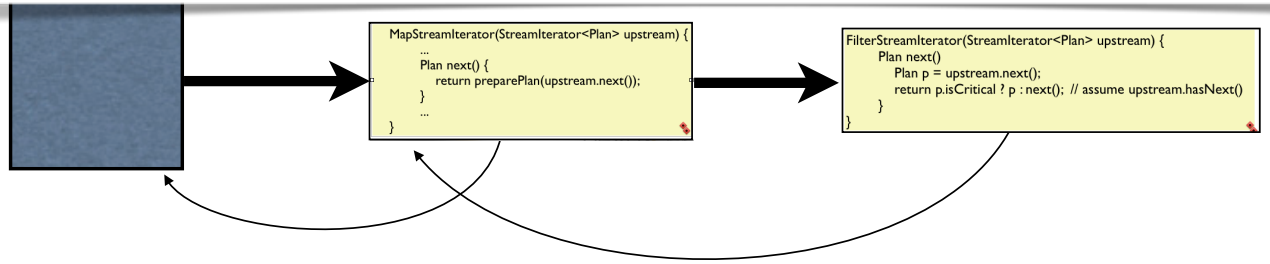
`java.util.functions.Predicate`

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

What is a `Stream`?

-
-
-

```
MapStreamIterator(StreamIterator<Plan> upstream, Function<Plan,Plan> m) {
        ...
        Plan next() {
            return m.apply(upstream.next());
        }
        ...
}
```

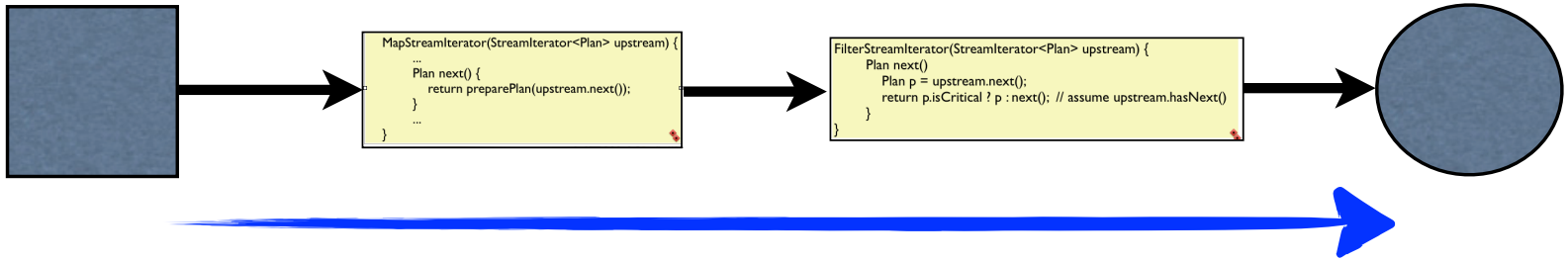


No evaluation yet!

Lazy and eager operations

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

```
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))

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

// get all but the first five plans
planStream.skip(10)
```

Some `Stream` methods

name	returns		interface used	λ signature
filter	Stream<T>	lazy	Predicate<T>	T → boolean
map	Stream<R>	lazy	Function<T,R>	T → R
sorted	Stream<T>	lazy	Comparator<T>	(T, T) → boolean
limit, substream	Stream<T>	lazy		
reduce	T	eager	BinaryOperator<T>	(T, T) → T
findFirst	T	eager	Predicate<T>	T → boolean
forEach	void	eager	Consumer<T>	T → void

Presentation Overview

- The Magic Ingredient
- Result: Better APIs
  - Internal Iteration
  - Pipes and Filters
  - Composing Behaviours
- Result: More Parallelism

Composing Behaviours

Sorting a list using `Comparator`:

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

`compare` is *monolithic*

- combines key *extraction* with key *comparison*

Could we parameterise `compare` on the key extractor?

Composing Behaviours

Starting with a Comparator

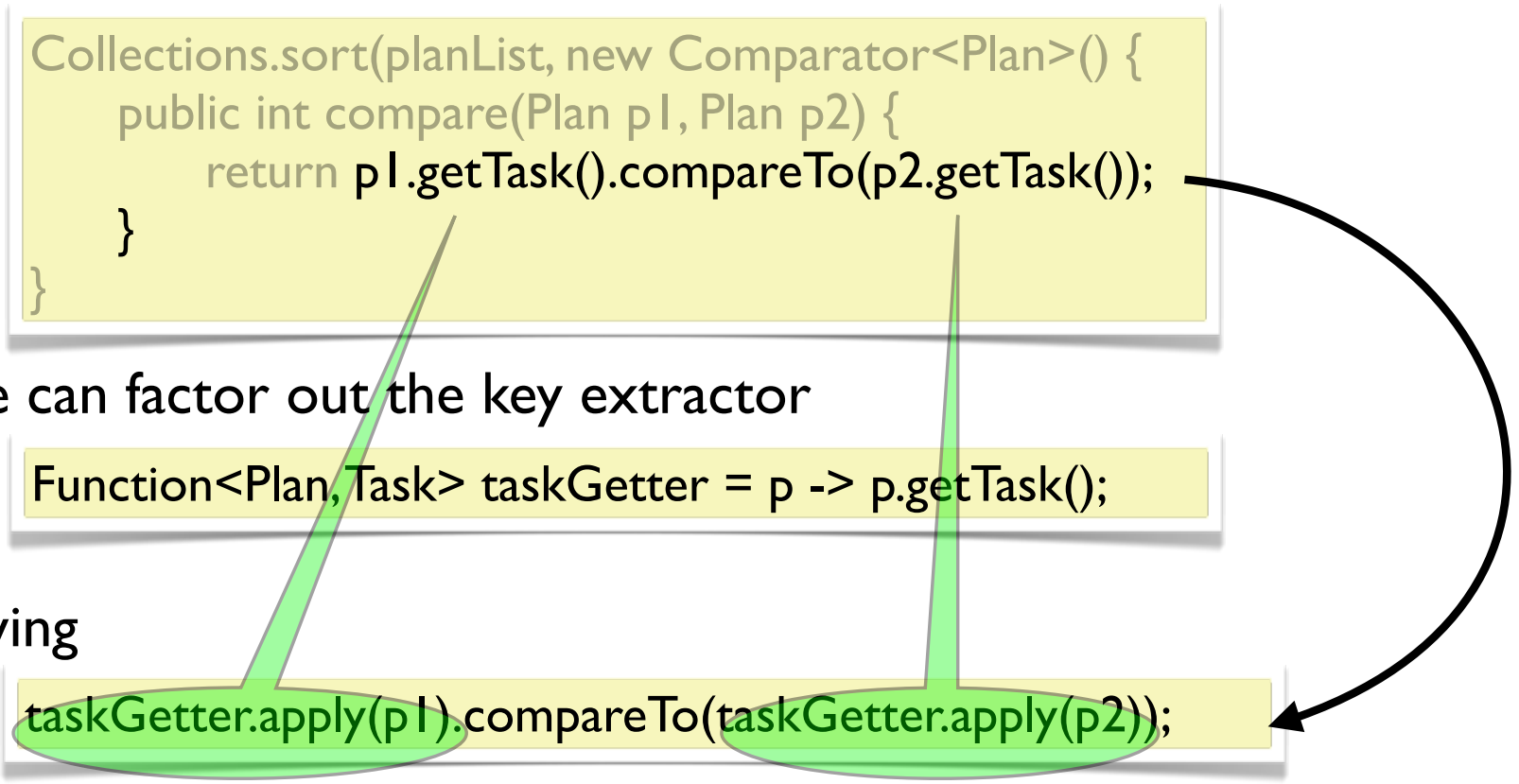
```
Collections.sort(planList, new Comparator<Plan>() {
      public int compare(Plan p1, Plan p2) {
            return p1.getTask().compareTo(p2.getTask());
      }
}
```

We can factor out the key extractor

```
Function<Plan, Task> taskGetter = p -> p.getTask();
```

Giving

```
taskGetter.apply(p1).compareTo(taskGetter.apply(p2));
```



Making methods more precise

Parameterising on the key extractor is already done for us:

In class `java.util.Comparators`:

```
public static <T,U> Comparator<T> comparing(Function<T,U> keyExtractor) {
        return (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

This method accepts—and returns—behaviours!

And

```
taskGetter.apply(p1).compareTo(taskGetter.apply(p2));
```

becomes

```
Comparators.comparing( taskGetter );
```

Composing fine-grained methods

Can now fine-tune behaviours:

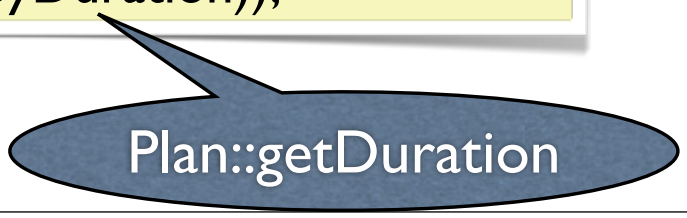
```
Comparator<Plan> byTask = Comparators.comparing(p -> p.getTask());
Comparator<Plan> byDuration = Comparators.comparing(p -> p.getDuration());
```

and combine them:

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

and even better:

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



Presentation Overview

- The Magic Ingredient
- Result: Better APIs
- Result: More Parallelism

Example: ConcurrentHashMap

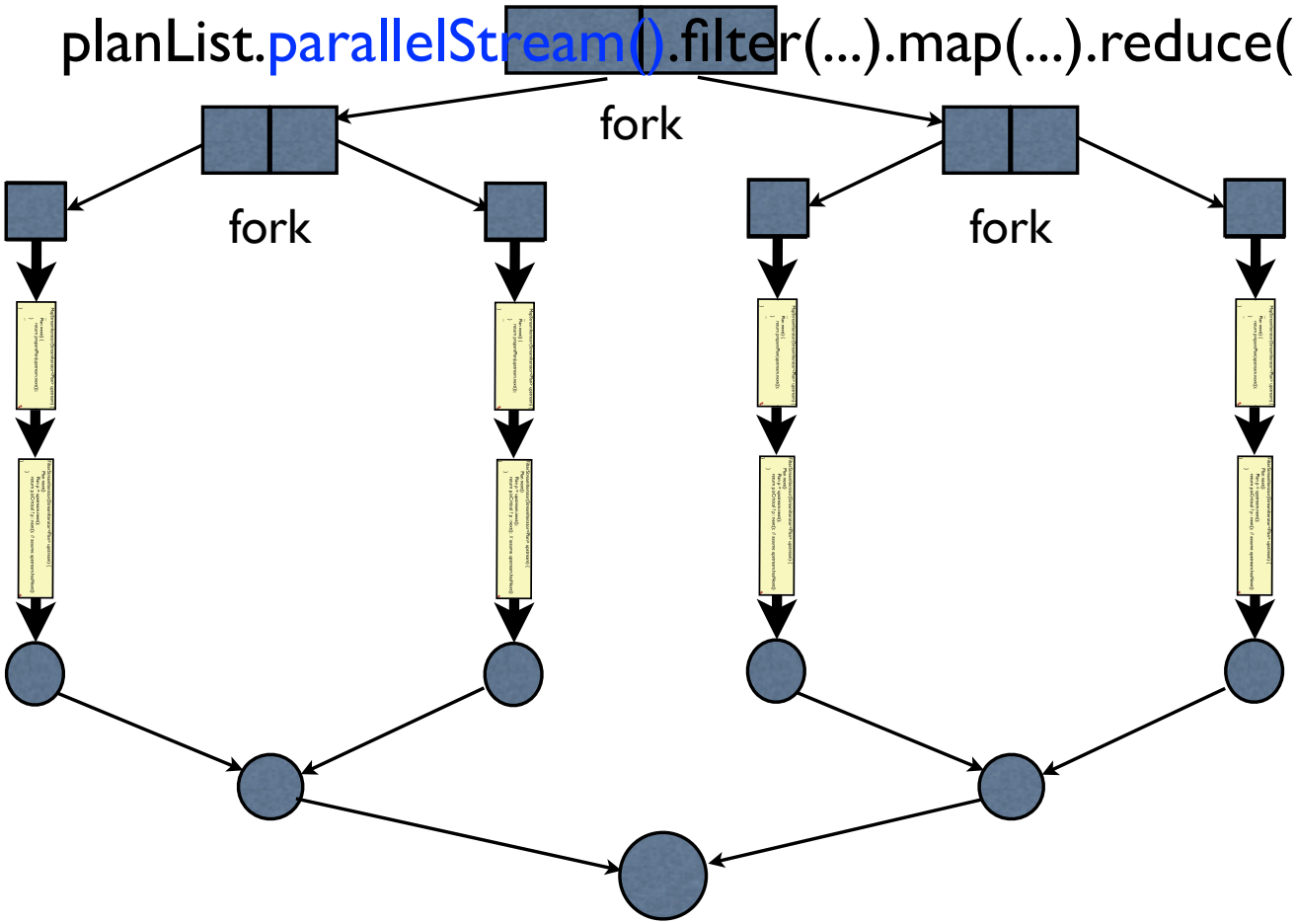
- Methods: `forEachInParallel`, `forEachEntryInParallel`, `forEachKeyInParallel`, `forEachValueInParallel`, ...
- Consider `forEachValueInParallel(Consumer<? super V> action)`
  - Creates a new `ForEachValueTask`
    - subclass of `ForkJoinTask`
    - submits that to the `ForkJoinPool`
  - `ForkJoinPool` executes `ForEachValueTask` on one of its threads

ConcurrentHashMap.ForEachValueTask

```
public final void compute() {
    final Consumer<? super V> action;
    if ((action = this.action) != null) {
        for (int b; (b = preSplit()) > 0;)
            new ForEachValueTask<K,V>(map, this, b, action).fork();
        forEachValue(action);
        propagateCompletion();
    }
}
```

Stream parallelisation

planList.parallelStream().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

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 2**. This consists mainly of the changes that the **Java Language Specification** will require for Project Lambda;
- **State of the Lambda v4**. 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 it is covered in this FAQ;
- **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 107** (JDK 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/lambda/**. These are sometimes, but not always, up-to-date; if you need a more up-to-date version:

www.lambdafaq.org/resources

- daily binary builds for Linux can be found at **http://obuildfactory.hgomez.net/**
- instructions for building Mac binaries from source at **https://github.com/hgomez/obuildfactory/wiki**
- **instructions for the official "new 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 slide 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** (Brian Goetz) provides a deep and comprehensive view of Project Lambda.
  - **Jump Starting Lambda Programming** (Stuart Marks) is a contrast—a gentle and painstaking introduction.
  - **Lambda: A Peek Under the Hood** (Brian Goetz) gives a wealth of technical detail about the implementation.
- **Others**
  - JavaZone 2012: **Lambdas in Java 8** (Angelika Langer)
  - Strange Loop 2012: **Project Lambda in Java SE 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 (either register (free) as a subscriber to download the magazine as PDF, or get it via the Newsstand app on iPhone or iPad).

Tool support

www.lambdafaq.org

- JetBrains have released EAP (early access program) versions of Intellij IDEA, already providing quite good support for lambda expressions and other parts of the Java 8 feature set. **Currently available** as EAP downloads.
- The **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-desired 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, test 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 specs now belong on the new lists, described next. (It should be said, though, that in fact language and feature discussion on lambda-dev was continuing at least up until early Nov 2012.)