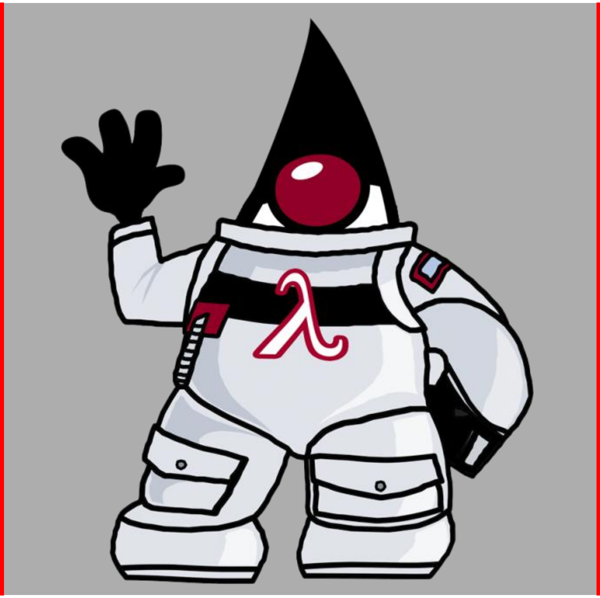


Project Lambda: To Multicore and Beyond

Brian Goetz Alex Buckley Maurizio Cimadamore
Java Platform Group

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.

Introduction to Project Lambda

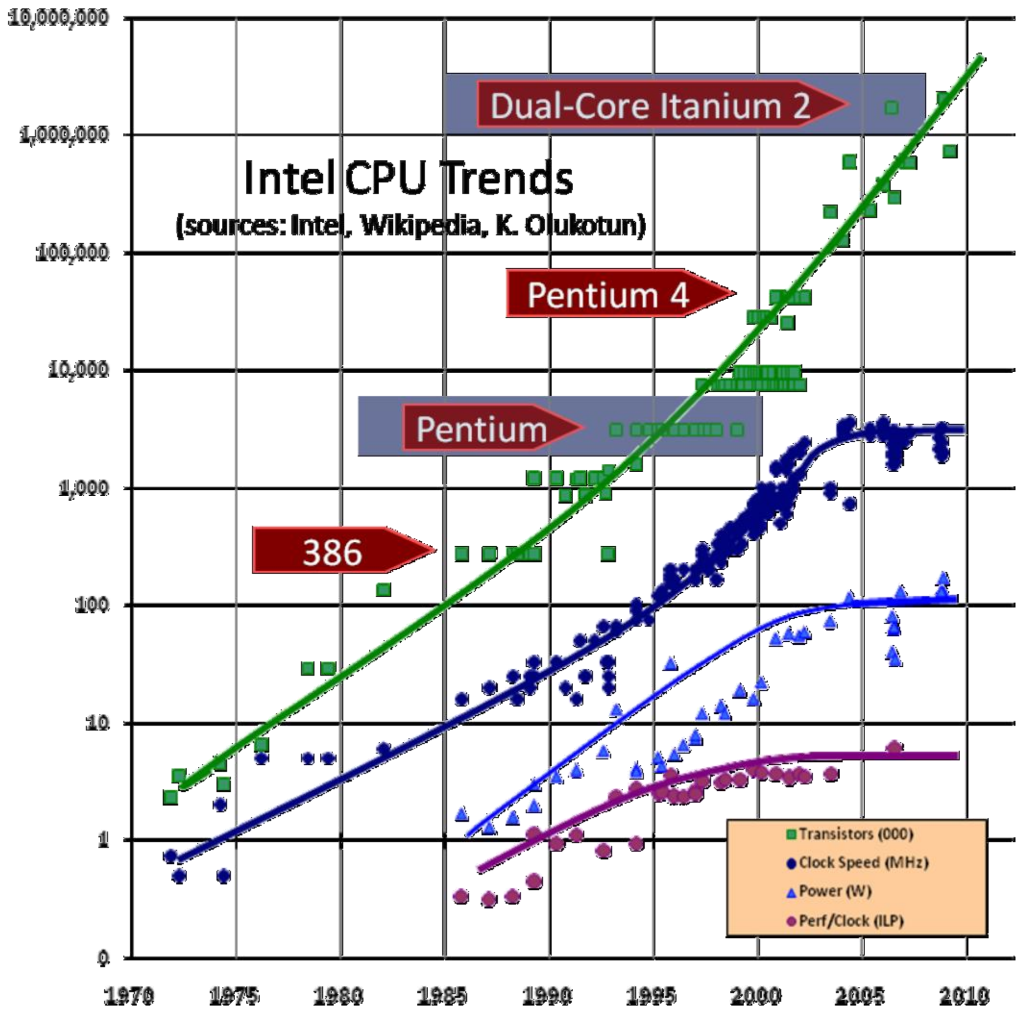
- OpenJDK Project Lambda started Dec 2009
- Targeted for Java SE 8 as JSR 335
- Aims to support programming in a multicore environment by adding closures and related features to the Java SE platform

MOTIVATION

Hardware trends – the future is parallel

(Graphic courtesy Herb Sutter)

- Chip designers have nowhere to go but parallel
  - Moore's Law gives more cores, not faster cores
  - Have hit the wall in power dissipation, instruction-level parallelism, clock rate, and chip scale
- We must learn to write software that parallelizes gracefully



Developers need simple parallel libraries

- One of Java's strengths has always been its libraries
  - Better libraries are key to making parallelization easier
  - Ideally, let the libraries worry about algorithmic decomposition, scheduling, computation topology
- Obvious place to start: parallel operations in collections
  - filter, sort, map/reduce
  - select, collect, detect, reject
- High-level operations tend to improve the readability of code, as well as its performance
- Why don't we see more parallel libraries today?

Without more language support for parallel idioms, people will instinctively reach for serial idioms

The biggest serial idiom of all: the for loop

```
double highestScore = 0.0;
for (Student s : students) {
    if (s.gradYear == 2010) {
        if (s.score > highestScore) {
            highestScore = s.score;
        }
    }
}
```

- This code is *inherently serial*
  - Traversal logic is fixed (iterate serially from beginning to end)
  - Business logic is stateful (use of > and accumulator variable)

ORACLE

The biggest serial idiom of all: the for loop

```
double highestScore = 0.0;
for (Student s : students) {
    if (s.gradYear == 2010) {
        if (s.score > highestScore) {
            highestScore = s.score;
        }
    }
}
```

- Existing collections impose *external iteration*
  - Client of collection determines mechanism of iteration
  - Implementation of accumulation is over-specified
  - Computation is achieved via side-effects

Let’s try a more parallel idiom: internal iteration

```
double highestScore =
    students.filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.gradYear == 2010;
        }
    }).map(new Extractor<Student,Double>() {
        public Double extract(Student s) {
            return s.score;
        }
    }).max();
```

- Not inherently serial!
  - Traversal logic is not fixed by the language
  - Business logic is stateless (no stateful accumulator)

ORACLE

Let’s try a more parallel idiom: internal iteration

```
double highestScore =
    students.filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.gradYear == 2010;
        }
    }).map(new Extractor<Student,Double>() {
        public Double extract(Student s) {
            return s.score;
        }
    }).max();
```

- Iteration and accumulation are embodied in the library
  - e.g. filtering may be done in parallel
  - Client is more flexible, more abstract, less error-prone

ORACLE

But … Yuck!

```
double highestScore =
    students.filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.gradYear == 2010;
        }
    }).map(new Extractor<Student,Double>() {
        public Double extract(Student s) {
            return s.score;
        }
    }).max();
```

- Can’t see the beef for the bun!

A wise customer once said:

“The pain of anonymous inner classes makes us roll our eyes in the back of our heads every day.”

LAMBDA EXPRESSIONS

A better way to represent "code as data"

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

- Lambda expression is introduced with #
- Zero or more formal parameters
  - Like a method
- Body may be an expression or statements
  - Unlike a method
  - If body is an expression, no need for 'return' or ';'

A better way to represent "code as data"

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

- Code reads like the problem statement:
  "Find the highest score of the students who graduated in 2010"

Lambda expressions support internal iteration

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

- Shorter than nested for loops, and *potentially faster* because implementation determines how to iterate
  - Virtual method lookup chooses the best filter() method
  - filter() method body can exploit representation knowledge
  - Opportunities for lazy evaluation in filter() and map()
  - Opportunities for parallelism

The science of lambda expressions

- The name comes from the lambda calculus created by Church (1936) and explored by Steele and Sussman (1975-1980)
- A lambda expression is like a lexically scoped anonymous method
  - Lexical scoping: can read variables from the lexical environment, including ‘this’, unlike with inner classes
  - No shadowing of lexical scope, unlike with inner classes
  - Not a member of any class, unlike with inner classes

Syntax wars

#()(7)

{ => 7 }

() -> 7;

{ -> 7 }

[ ] { return 7; }

lambda() (7);

#(->int) { return 7; }

#(: int i) { i = 7; }

new #<int() > (7)

TYPING

What is the type of a lambda expression?

```
#{ Student s -> s.gradYear == 2010 }
```

- Morally, a function type from Student to boolean
- But Java does not have function types, so:
  - How would we write a function type?
  - How would it interact with autoboxing?
  - How would it interact with generics?
  - How would it describe checked exceptions?

“Use what you know”

- Java already has an idiom for describing “functional things”: single-method interfaces (or abstract classes)

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

- Let’s reuse these, rather than introduce function types

  `Comparator<T>` ~ a function type from (T,T) to boolean

  `Predicate<T>` ~ a function type from T to boolean

ORACLE

Introducing: SAM types

- A SAM type is an interface or abstract class with a Single Abstract Method

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

- No special syntax to declare a SAM type
  - Recognition is automatic for suitable interfaces and abstract classes
  - Not just for java.* types!

DirectoryStream.Filter (Java Platform SE 7 b108) - Mozilla Firefox 4.0 Beta 6

File Edit View History Bookmarks Tools Help

http://download.java.net/jdk7/docs/api/java/nio/file/DirectoryStream.Filter.html

Please note that the specifications and other information contained herein are not final and are subject to change. The information is being made available to you solely for purpose of evaluation.

Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Java™ Platform Standard Ed. 7 DRAFT ea-b108

java.nio.file

Interface DirectoryStream.Filter<T>

Type Parameters:

`T` - the type of the directory entry

Enclosing interface:

DirectoryStream<T>

```
public static interface DirectoryStream.Filter<T>
```

An interface that is implemented by objects that decide if a directory entry should be accepted or filtered. A `Filter` is passed as the parameter to the `newDirectoryStream` method when opening a directory to iterate over the entries in the directory.

Since:

1.7

Method Summary

Modifier and Type	Method and Description
`boolean`	`accept(T entry)` Decides if the given directory entry should be accepted or filtered.

Done

The type of a lambda expression is a SAM type

- “SAM conversion” infers a SAM type for a lambda expression

```
Predicate<Student> p = #{ Student s -> s.gradYear == 2010 };
```

- Invoking the SAM type’s method invokes the lambda’s body

```
boolean ok = p.isTrue(aStudent);
```

- Instant compatibility with existing libraries!

```
executor.submit(#{ -> println(“Boo”); });

btn.addActionListener(#{ ActionEvent e -> println(“Boo”) });
```

The science of SAM conversion

- Lambda expression must have:
  - Same parameter types and arity as SAM type's method
  - Return type compatible with SAM type's method
  - Checked exceptions compatible with SAM type's method

- SAM type's method name is not relevant:

```
interface Predicate<T> { boolean op(T t); }
Predicate<Student> p = #{ Student s -> s.gradYear == 2010 };
interface StudentQualifier { Boolean check(Student s); }
StudentQualifier c   = #{ Student s -> s.gradYear == 2010 };
```

- Lambda expressions may only appear in contexts where they can undergo SAM conversion (assignment, method call/return, cast)

But wait, there's more

- Lambdas solve the "vertical problem" of inner classes
- Parameter types can still be a "horizontal problem"

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

ORACLE

But wait, there’s more

- Lambdas solve the “vertical problem” of inner classes
- Parameter types can still be a “horizontal problem”

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

- SAM conversion can usually infer them!

```
double highestScore =
   students.filter(#{ s -> s.gradYear == 2010 })
           .map(   #{ s -> s.score })
           .max();
```

- Lambda expressions are always statically typed

SAM conversion includes *target typing*

- Target typing identifies parameter types for the lambda expression based on the candidate SAM type’s method

```
interface Collection<T> {
    Collection<T> filter(Predicate<T> t);
}
Collection<Student> students = …
… students.filter(#{ s -> s.gradYear == 2010 }) …
```

  - students.filter() takes a Predicate<Student>
  - Predicate<Student> is a SAM type whose method takes Student
  - Therefore, **s** must be a Student
  - Programmer can give parameter types in case of ambiguity

ORACLE

Recap: SAM types

- Self-documenting
- Build on existing concepts
  - Wildcards have made us wary of aggressive new type systems
- Ensure lambda expressions work easily with existing libraries
  - Java SE will likely define a “starter kit” of SAM types such as Predicate, Filter, Extractor, Mapper, Reducer…
- Type inference gets your eyes to the “beef” quickly
  - Style guide: One-line lambdas may omit parameter types, but multi-line lambdas should include parameter types
- You could think of our lambda expressions as “SAM literals”

METHOD REFERENCES

Motivation

- Consider sorting a list of Person objects by last name:

```
class Person { String getLastName() {…} }

List<Person> people = …
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person a, Person b) {
        return a.getLastName().compareTo(b.getLastName());
    }
});
```

- Yuck!
  - (Worse if sort key is a primitive)

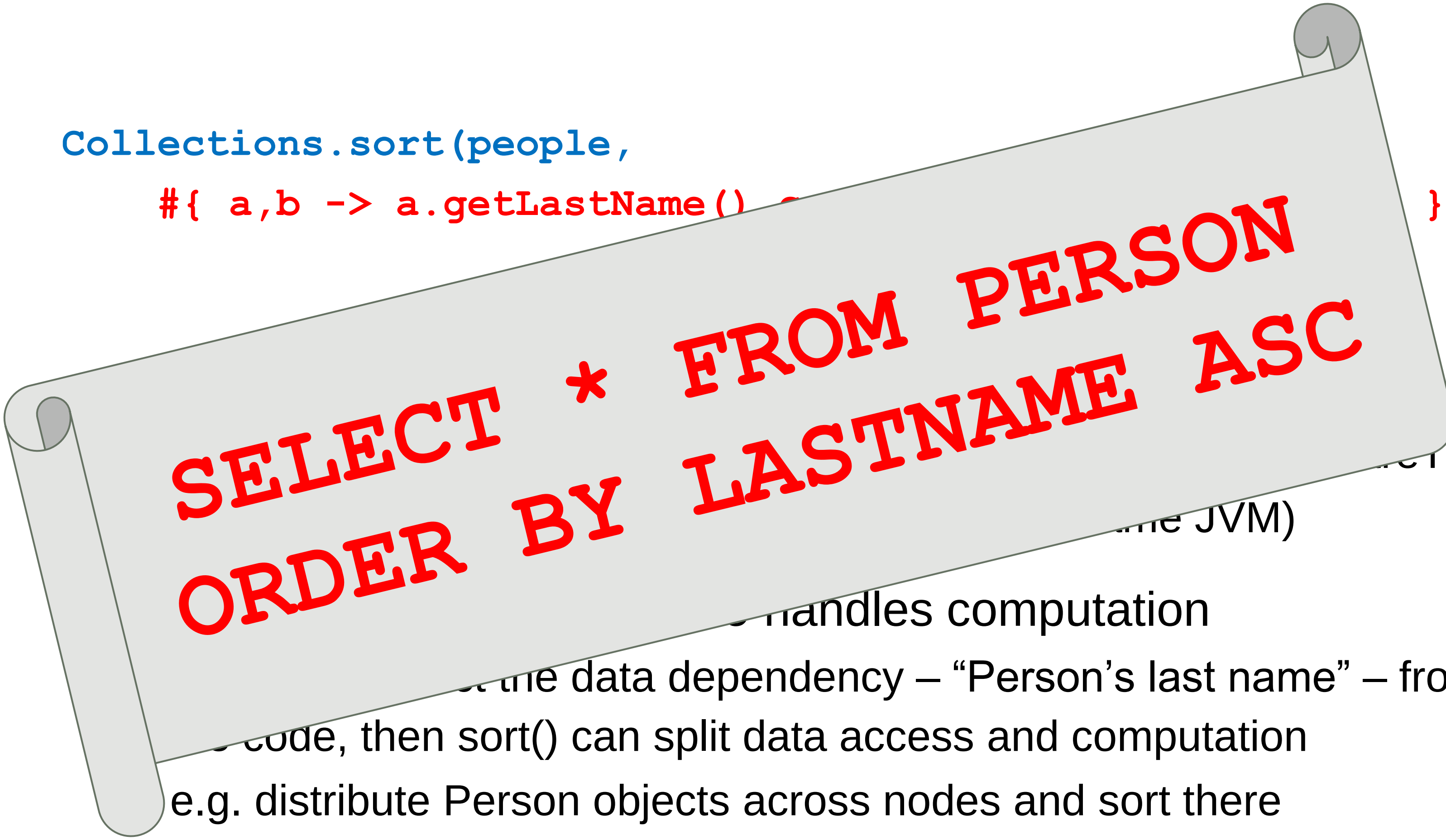
A lambda expression helps, but only so much

```
Collections.sort(people,
    #{ a,b -> a.getLastName().compareTo(b.getLastName()) });
```

- More concise, but not more abstract
  - Performs data access (getLastName) and computation (compareTo)
  - Assumes both Person objects are nearby (e.g. same JVM)
- More abstract if *someone else* handles computation
  - If we can extract the data dependency – “Person’s last name” – from the code, then sort() can split data access and computation
  - e.g. distribute Person objects across nodes and sort there

A lambda expression helps, but only so much

```
Collections.sort(people,
    #{ a,b -> a.getLastName() …                    });
```



…eTo)

…me JVM)

…handles computation

…t the data dependency – “Person’s last name” – from …e code, then sort() can split data access and computation

e.g. distribute Person objects across nodes and sort there

How to express “Person’s last name” in Java?

- Assume an interface to extract a value from an object:

```
interface Extractor<T, U> { U get(T element); }
```

- And a sortBy method keyed off an extractor:

```
public <T, U extends Comparable<…>>
    void sortBy(Collection<T> coll, Extractor<T,U> ex) {…}
}
```

- Then, pass a lambda expression that “wraps” a method call:

```
Collections.sortBy(people, #{ p -> p.getLastName() });
```

  – SAM conversion types the lambda as Extractor<Person,String>
  – sortBy() can pre-query last names, cache them, build indices…

Is that the best we can do?

```
Collections.sortBy(people, #{ p -> p.getLastName() });
```

- Writing little wrapper lambdas will be a pain
- If only we could reuse an existing method...

Is that the best we can do?

```
Collections.sortBy(people, #{ p -> p.getLastName() });
```

- Writing little wrapper lambdas will be a pain
- If only we could reuse an existing method

```
Collections.sortBy(people, #Person.getLastName);
```

- Method reference introduced with #
- No need for () or parameter types in simple cases

ORACLE

Recap: Method references

- When code outgrows a lambda expression, write a method and take a method reference
- Lambda expressions and method references have SAM types
- Work easily with existing libraries
- Can specify parameter types explicitly if needed
- Three kinds of method references (unbound/bound/static)
- No field references (use method references to getters/setters)

A word about implementation

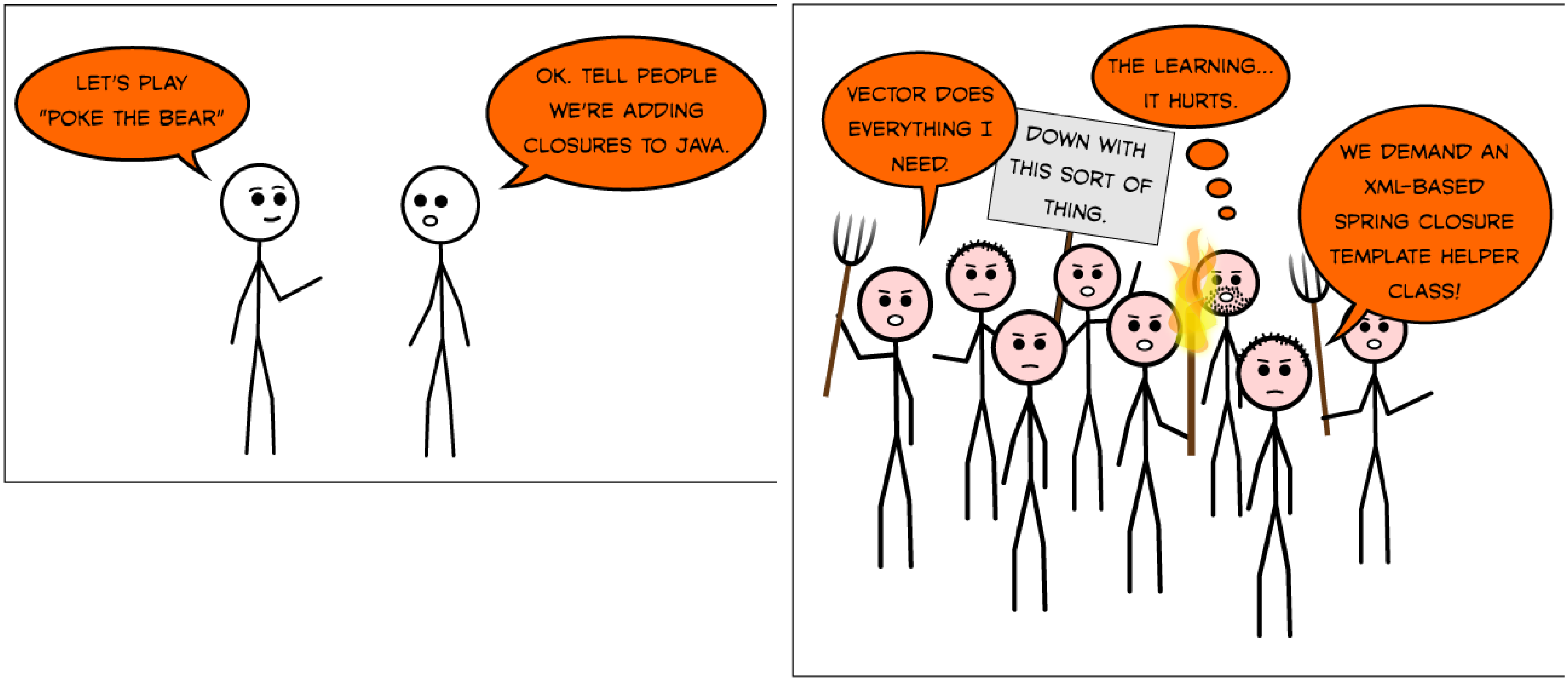
- Lambda expressions are not sugar for inner classes
  - Implemented with MethodHandle from JSR 292
- Method references are not sugar for wrapper lambdas
  - Implemented with enhanced ldc instruction from JSR 292
- See videos from 2010 JVM Language Summit for more
  - http://wiki.jvmlangsummit.com/
  - “Gathering the threads: JVM Futures”
  - “Efficient compilation of Lambdas using MethodHandle and JRockit”
  - “MethodHandles: an IBM implementation”

ORACLE

“But what about…”

- Function types
- Properties
- Curried functions
- Underscore placeholders
- Field references
- Partial application
- ‘var’
- Control abstraction
- ‘letrec’
- Tennent’s Correspondence Principle
- Dynamic typing
- Continuations

COMIC #61 – POKE THE BEAR



HTTP://TWITTCH.COM/61/

Twittch by Eric Burke

Our view

- Evolving a language with millions of developers is a fundamentally different task from evolving a language with thousands of developers
  - Adding features by the bucket is not good
  - Every feature adds conceptual weight
- We believe Project Lambda's changes are measured, and in the spirit of Java
  - Focus on readability and developer productivity
  - No new types to learn (compare with wildcards)
  - Respectful of existing idioms (SAM)

LIBRARY EVOLUTION

As the language evolves, the libraries should evolve with it

- Java collections do not support internal iteration largely because the language made it so clunky at the time
- Now the language can easily treat “code as data”, it’s crucial to support parallel/functional idioms in the standard libraries
- Continues a long theme of language/library co-evolution
  - synchronized {} blocks / Thread.wait()/notify()
  - for-each loop / Iterable<T>
  - Generic type inference / <T>Collection.toArray(T[] x)

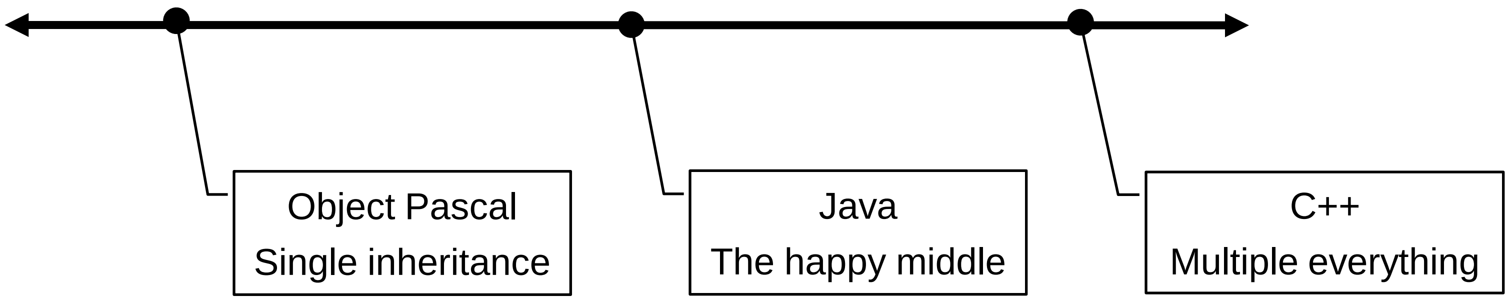
Without more library support for parallel idioms,
people will instinctively reach for serial idioms

Library support for internal iteration

- Sometimes, we want to add more types
  - Recall Java SE will likely define a “starter kit” of SAM types
- Sometimes, we want to augment existing interfaces
- No good way to add methods to existing interfaces today
  - Binary compatible: old clients continue to execute ☺
  - Source *in*compatible: old implementations fail to compile ☹
- Existing techniques for interface evolution are insufficient
  - Adding to j.u.Collections diminishes the value of interface contracts
  - Using abstract classes instead of interfaces
  - Interface inheritance and naming conventions (e.g. IDocument, IDocumentExtension, IDocumentExtension2, IDocumentExtension3)

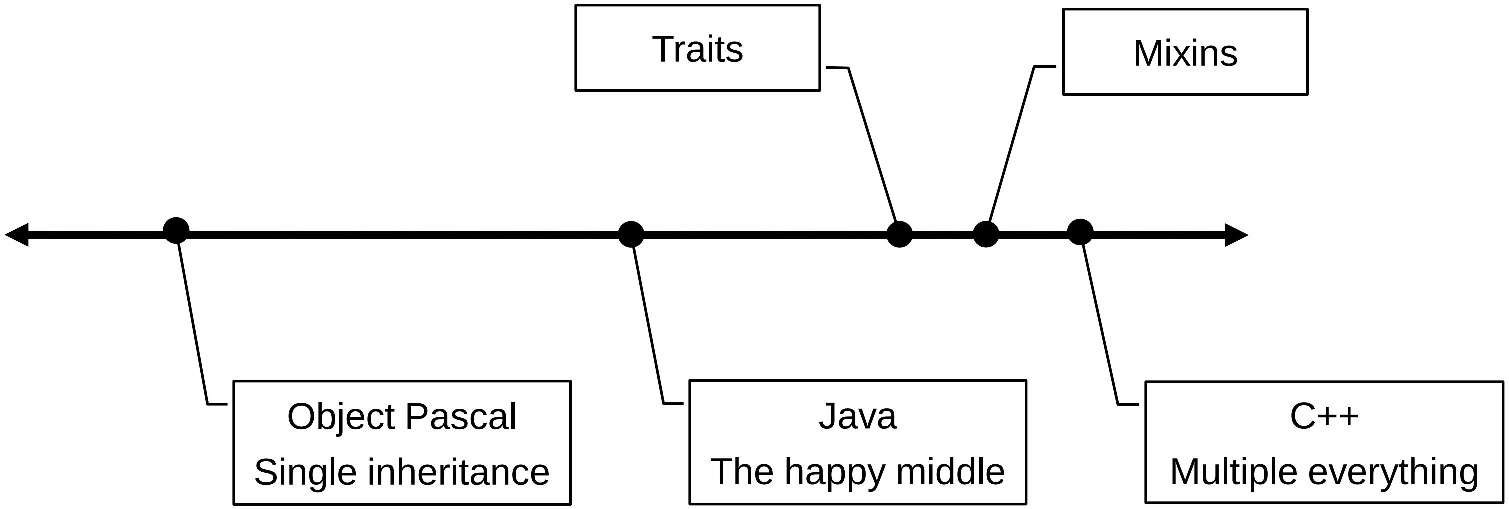
Interface evolution

- There is a spectrum of inheritance expressiveness



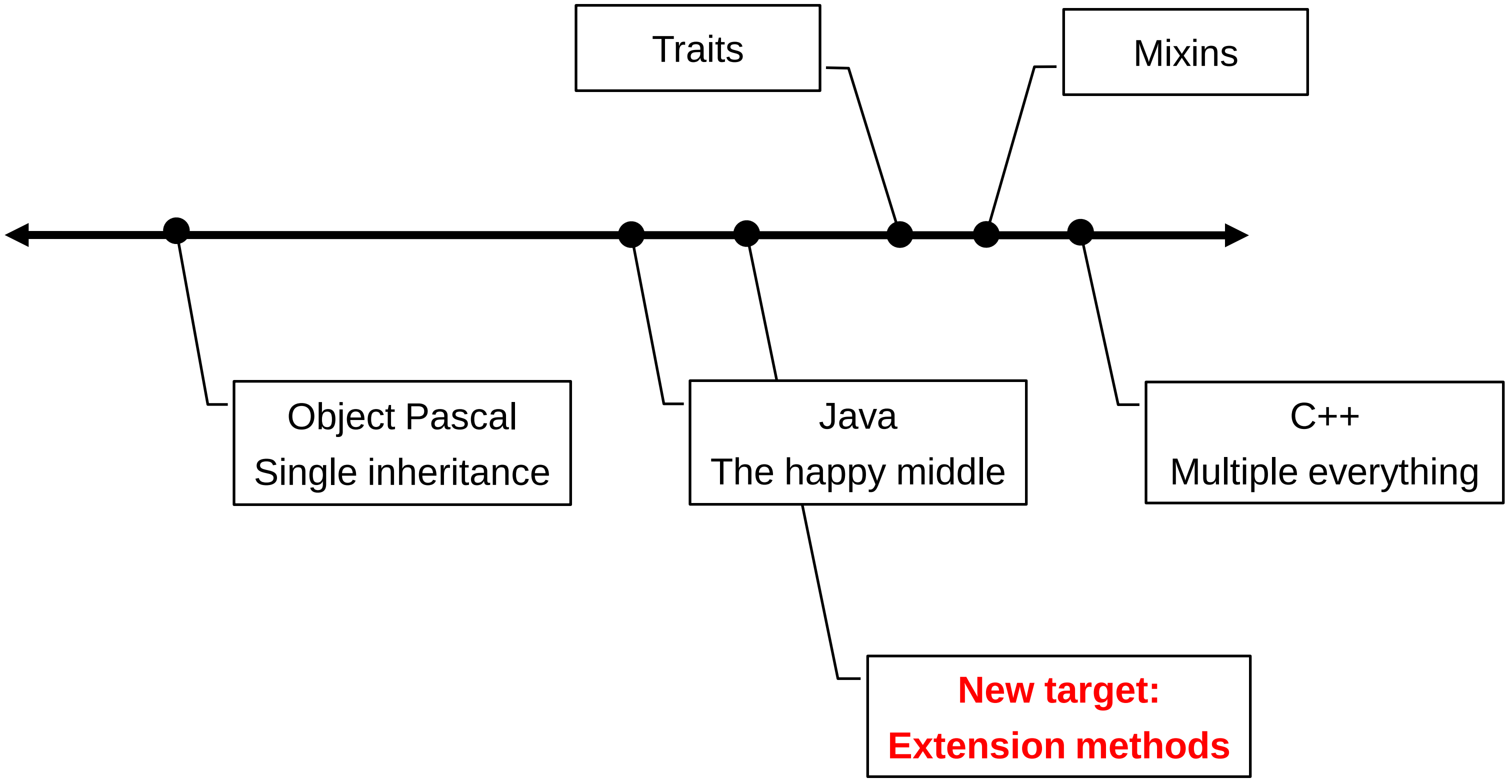
Interface evolution

- There is a spectrum of inheritance expressiveness



Interface evolution

- There is a spectrum of inheritance expressiveness



Extension methods: a measured step towards more flexible inheritance

```
public interface Set<T> extends Collection<T> {
    public int size();
    …
    public extension T reduce(Reducer<T> r)
        default Collections.<T>setReducer;
}
```

- Allows library maintainers to effectively add methods after the fact by specifying a default implementation
  - "If you cannot afford an implementation of reduce(), one will be provided for you"
- Less problematic than traits, mixins, full multiple inheritance

ORACLE

Extension methods in a nutshell

- An extension method is just an ordinary interface method
- For a client:
  - Nothing new to learn – calling the extension method works as usual, and the default method is linked dynamically if needed
- For an API implementer:
  - An implementation of an augmented interface may provide the method, or not
- For an API designer:
  - Default method can only use public API of augmented interface
- For a JVM implementer:
  - Lots of work

WRAP-UP

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions

```
Collections.sort(people, #{ Person x, Person y ->
    x.getLastName().compareTo(y.getLastName()) });
```

More essence,
less ceremony

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions

SAM conversion

```
Collections.sort(people, #{ Person x, Person y ->
    x.getLastName().compareTo(y.getLastName()) });
```

More essence,
less ceremony

Forward compatibility
– old API works with
new expressions

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions

SAM conversion

```
Collections.sort(people, #{ Person x, Person y ->
    x.getLastName().compareTo(y.getLastName()) });
```

Better libraries

```
Collections.sortBy(people, #{ Person p -> p.getLastName() });
```

More abstraction

ORACLE

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions *SAM conversion*

```
Collections.sort(people, #{ Person x, Person y ->
    x.getLastName().compareTo(y.getLastName()) });
```

Better libraries

```
Collections.sortBy(people, #{ Person p -> p.getLastName() });
```

Type inference

```
Collections.sortBy(people, #{ p -> p.getLastName() });
```

Static typing can
be fun too!

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions *SAM conversion*

```
Collections.sort(people, #{ Person x, Person y ->
    x.getLastName().compareTo(y.getLastName()) });
```

Better libraries

```
Collections.sortBy(people, #{ Person p -> p.getLastName() });
```

Type inference

```
Collections.sortBy(people, #{ p -> p.getLastName() });
```

Method references

```
Collections.sortBy(people, #Person.getLastName);
```

Don’t Repeat Yourself

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions *SAM conversion*

```
Collections.sort(people, #{ Person x, Person y ->
    x.getLastName().compareTo(y.getLastName()) });
```

Better libraries

```
Collections.sortBy(people, #{ Person p -> p.getLastName() });
```

Type inference

```
Collections.sortBy(people, #{ p ->
```

Method references

```
Collections.sortBy(people, #Per
```

Extension methods

```
people.sortBy(#Person.getLastName);
```

Migration compatibility –
Old class implements
new interface

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions

Better libraries

Type inference

Method references

Extension methods

```
people.sortBy(#Person.getLastName);
```

Project Lambda: A Journey

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Lambda expressions	*More concise*
Better libraries	*More abstract*
Type inference	*Less ceremony*
Method references	*More reuse*
Extension methods	*More object-oriented*

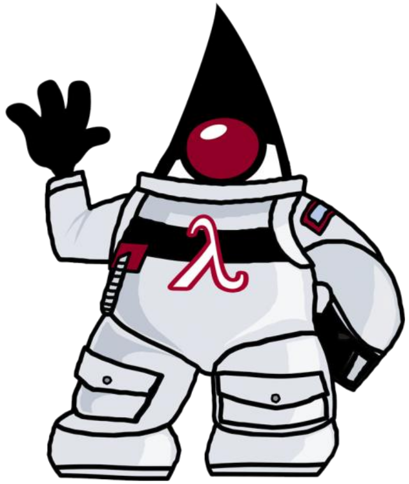
```
people.sortBy(#Person.getLastName);
```

Project Lambda: To Multicore and Beyond

- Project Lambda is not just “closures”
- A suite of features to support parallel/functional idioms
- With an eye on compatibility, as always
- Collections story is a work in progress
- JVM evolution in JSR 292 really helps the Java language
- Steady pipeline of measured innovation

ORACLE

Project Lambda @ OpenJDK



- http://openjdk.java.net/projects/lambda/
- 2461 mails (Dec 2009–Sep 2010)
- 292 subscribers (Sep 2010)
  - Acknowledgements: Alex Blewitt, Joshua Bloch, Nathan Bryant, Stephen Colebourne, Collin Fagan, Remi Forax, Neal Gafter, Thomas Jung, Lawrence Kesteloot, Peter Levart, Howard Lovatt, Mark Mahieu, Mark Thornton, Reinier Zwisterloot, and many more
- Special thanks: Maurizio Cimadamore
- Download javac and lambda-ify your code!
- Please report experiences to lambda-dev@openjdk.java.net

LIBRARY UPGRADES

The destination of the journey: Parallel Collections

```
double highestScore = 0.0;
for (Student s : students) {
    if (s.gradYear == 2010) {
        if (s.score > highestScore) {
            highestScore = s.score;
        }
    }
}
```

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

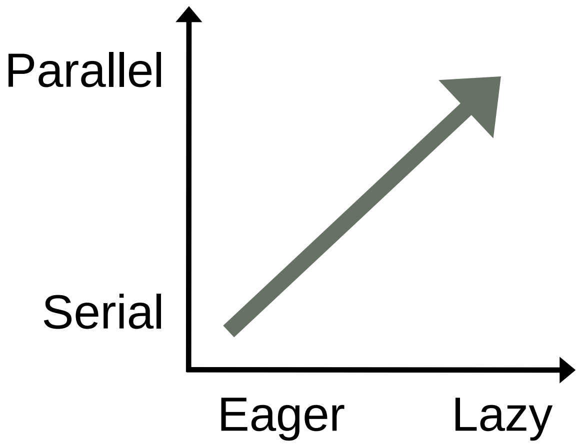
So we're done?

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

- What should students.filter() and map() return?
  - Obvious choice is Collection<Student>
- Intermediate collection objects are unnecessary overhead ☹
  - More abstract, but not more performant: Fail
- filter() and map() should return lazy streams
  - LazyCollection<Student> ?

It's <u>still</u> not parallel

- Laziness and parallelism are independent dimensions
  – Just adding a lazy map is not enough



- Collections must offer parallel "iterators" that encapsulate decomposition, a.k.a. "spliterators"
  – See Guy Steele's "Iterators Considered Harmful"
- Each sub-part of a decomposed collection evaluates lazily

A modest requirement list

- New bulk-data aggregate operations
  - Collection types must offer aggregate operations
    Collection.map(Mapper), reduce(Reducer), filter(Predicate)
  - Generic implementations over decomposable collections
    Utils.map(Mapper), reduce(Reducer), filter(Predicate)
- New abstractions for lazy streams
  - Collection types must auto-convert to lazy collections
    LazyCollection<?>
- New abstractions for parallel decomposition
  - Collection types must expose parallel decomposition
    LazyCollection.asParallelStream()
- What does this all mean for existing non-thread-safe collections?

From eager serial to lazy parallel collections

```
double highestScore =
   students.filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

```
double highestScore =
   students.asParallelStream()
           .filter(#{ Student s -> s.gradYear == 2010 })
           .map(   #{ Student s -> s.score })
           .max();
```

SOFTWARE. HARDWARE. COMPLETE.

ORACLE