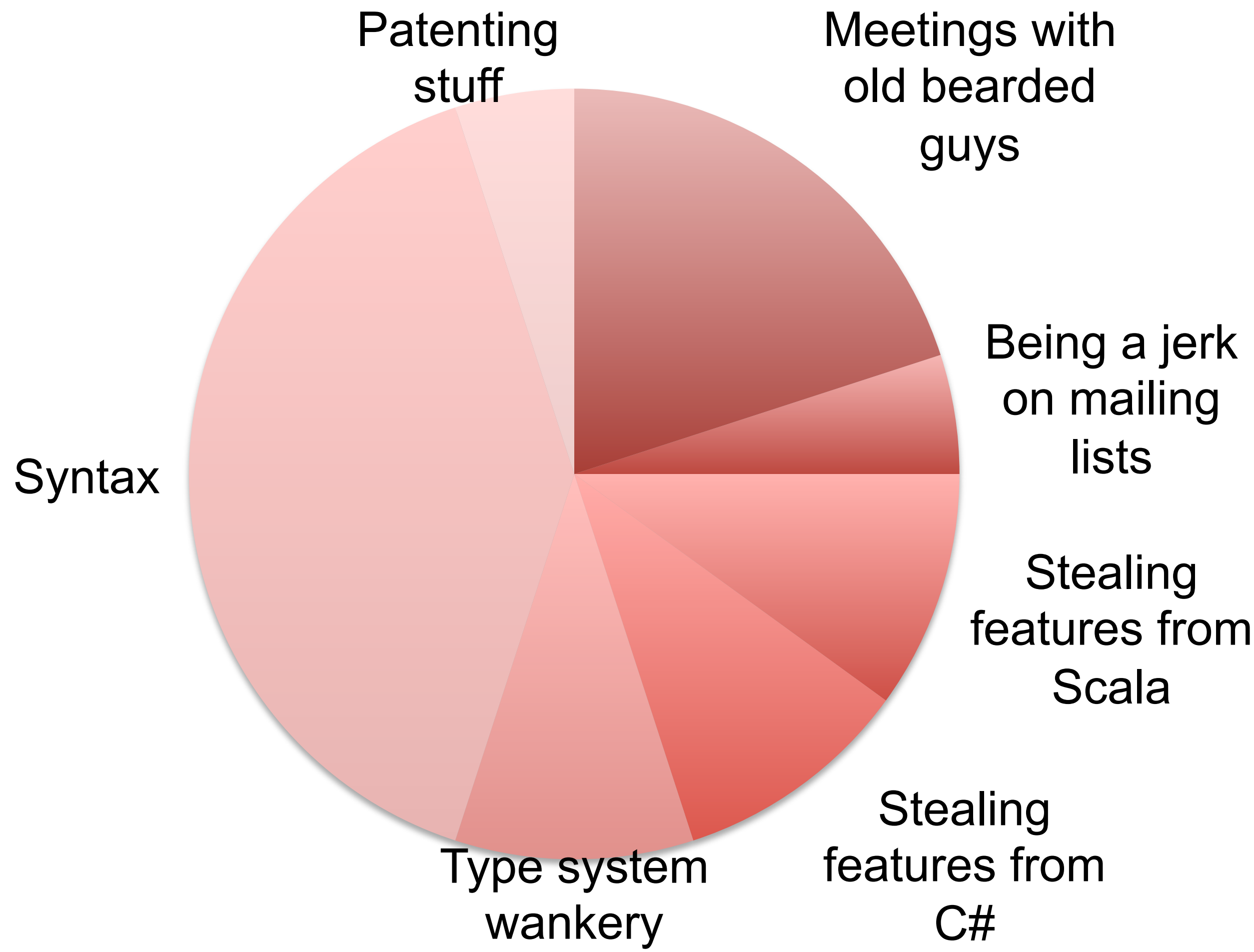
ORACLE

Evolving Java
Project Lambda, and Beyond

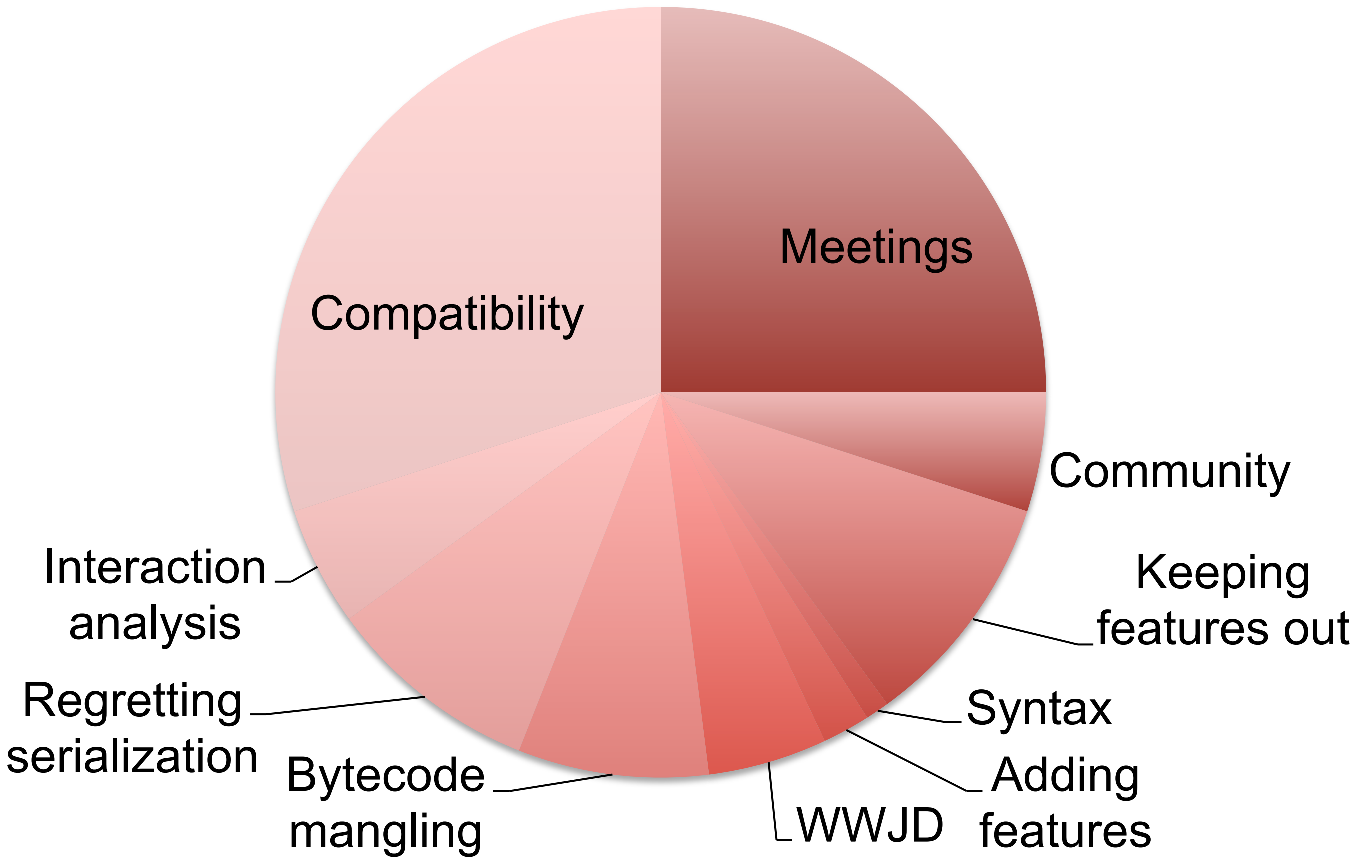
Brian Goetz
Java Language Architect
Oracle Corporation

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.

What people think I do

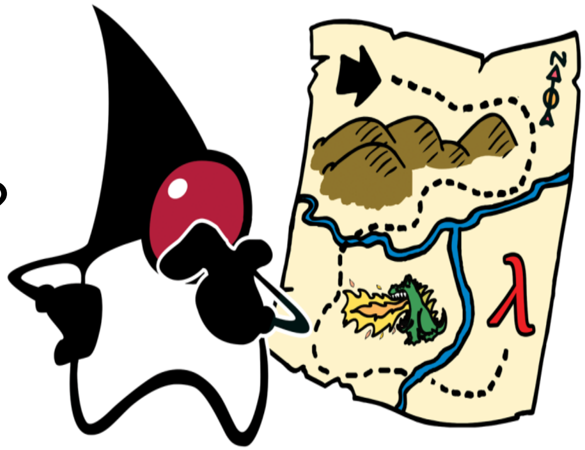


What (it feels like) I actually do



Modernizing Java

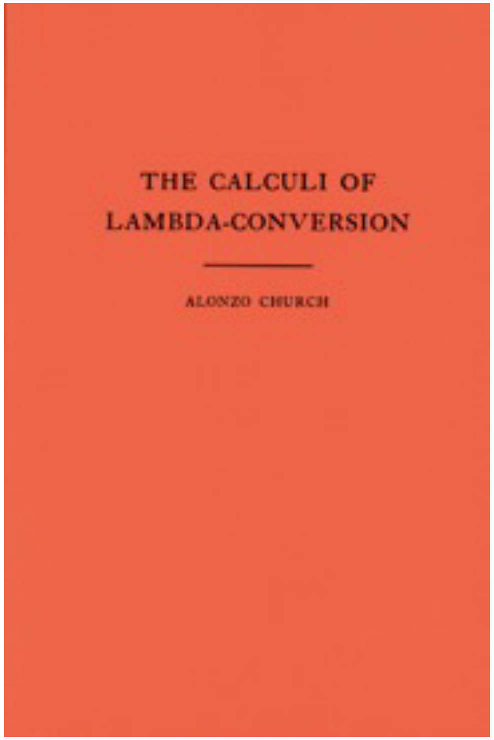
- Java SE 8 is a big step forward in modernizing the Java *Language*
  - Lambda Expressions (closures)
  - Interface Evolution (default methods)
- Java SE 8 is a big step forward in modernizing the Java *Libraries*
  - Bulk data operations on Collections
  - More library support for parallelism
- Why do we choose the features we do?
- How do we evolve a mature language?



Closures for Java – a long and winding road

- 1997 – Odersky/Wadler experimental “Pizza” work
- 1997 – Java 1.1 added inner classes – a weak form of closures
  - Too bulky, complex name resolution rules, many limitations
- In 2006-2008, a vigorous community debate about closures
  - Multiple proposals, including BGGA and CICE
  - Each had a different orientation
    - BGGA – creating control abstraction in libraries
    - CICE – reducing syntactic overhead of inner classes
  - Things ran aground at this point…
- Little evolution from Java SE 5 (2004) until now
  - Project Coin (Small Language Changes) in Java SE 7

Closures for Java – a long and winding road



Closures for Java – a long and winding road

- Dec 2009 – OpenJDK Project Lambda formed
- November 2010 – JSR-335 filed
- Current status
  - EDR specification complete
  - Prototype (source and binary) available on OpenJDK
  - Part of Java SE 8 (Summer 2013)
- JSR-335 = Lambda Expressions
  + Interface Evolution
  + Bulk Collection Operations

Evolving a major language

- Key evolutionary forces
  - Adapting to change
    - Everything changes: hardware, attitudes, fashions, problems, demographics
  - Righting what's wrong
    - Inconsistencies, holes, poor user experience
  - Maintaining compatibility
    - Low tolerance for change that will break anything
  - Preserving the core
    - Can't alienate user base in quest for "something better"
    - Easy to focus on cool new stuff, but there's lots of cool old stuff too

Adapting to Change

- In 1995, most mainstream languages did not support closures
  - Perceived to be "too hard" for ordinary developers
- Today, Java is just about the last holdout that doesn't
  - C++ added them recently
  - C# added them in 3.0
  - New languages being designed today all do

"In another thirty years people will laugh at anyone who tries to invent a language without closures, just as they'll laugh now at anyone who tries to invent a language without recursion."
-Mark Jason Dominus

Adapting to Change

- In 1995, pervasive sequentiality infected programming language design
  - For loops are sequential
    - Why wouldn't they be? Why invite nondeterminism?
    - Determinism is convenient – when free
    - Similarly, Iterator/Iterable is sequential
  - Pervasive mutability
    - Mutability is convenient – when free
    - Object creation was expensive and mutation cheap
- In today's world, these are just the wrong defaults!
  - Can't just outlaw for loops and mutability
  - Instead, gently *encourage* something better
- Lambda expressions is that gentle push

Problem – External Iteration

- “Take the red blocks and colors them blue”
- Typical solution with foreach loop
  - Loop is *inherently sequential*
    - Wasn’t a big problem 20 years ago, but times change
  - Client has to manage iteration
    - Conflates “what” with “how”
  - This is called *external iteration*
  - Hides complex interaction between library and client

```
for (Shape s : shapes) {
    if (s.getColor() == RED)
        s.setColor(BLUE);
}
```

Internal Iteration

- Re-written to use lambda and Collection.forEach
  - Not just a syntactic change!
  - Now the library is in control
  - *Internal iteration* – More *what*, less *how*
  - Client passes behavior into the API as data
- Library can use parallelism, out-of-order, laziness
- Also enable more powerful, expressive APIs
  - Greater power to abstract over behavior

```
shapes.forEach(s -> {
    if (s.getColor() == RED)
        s.setColor(BLUE);
})
```

Lambda Expressions

- A *lambda expression* is an anonymous method
  - Has an argument list, a return type, and a body

    ```
    (Object o) -> o.toString()
    ```
  - Can refer to values from the enclosing lexical scope

    ```
    (Person p) -> p.getName().equals(name)
    ```
  - Compiler can often infer parameter types from context

    ```
    p -> p.getName().equals(name)
    ```
- A method reference is a reference to an existing method

  ```
  Object::toString
  ```
- All of these forms allow you to *treat code as data*
  - Behavior can be stored in variables and passed to methods

What is the type of a lambda?

- Most languages with lambdas have some notion of a *function type*
  - Java language has no concept of function type
  - JVM has no native (unerased) representation of function type in VM type signatures
  - Adding function types would create many questions
    - How do we represent functions in VM type signatures?
    - How do we create instances of function types?
    - Want to avoid significant VM changes
  - Obvious tool for representing function types is generics
    - But then function types would be … erased

Functional Interfaces

- Historically used single-method interfaces to model functions
  - Runnable, Comparator, ActionListener
  - Let's just give these a name: *functional interfaces*
  - And add some new ones like Predicate<T>, Block<T>
- A lambda expression evaluates to an instance of a functional interface

```
Predicate<String> isEmpty = s -> s.isEmpty();
Predicate<String> isEmpty = String::isEmpty;
Runnable r = () -> { System.out.println("Boo!") };
```

Functional Interfaces

- “Just add function types” was obvious … and wrong
  - Would have introduced complexity and corner cases
  - Would have bifurcated libraries into “old” and “new” styles
  - Would have created interoperability challenges
- Preserve the Core
  - Stodgy old approach may be better than shiny new one
- Bonus: existing libraries are now *forward-compatible* to lambdas
  - Libraries that never imagined lambdas still work with them!
  - Maintains significant investment in existing libraries
  - Fewer new concepts

Problem – Interface Evolution

- Example used a new Collection method – forEach()
  - I thought you couldn't add new methods to interfaces?
- Interfaces are a double-edged sword
  - Cannot compatibly evolve them unless you control all implementations
  - Reality: APIs age
    - As we add cool new language features, existing APIs look even older!
  - Lots of bad options for dealing with aging APIs
    - Let the API stagnate
    - Replace it in entirety (every few years!)
    - Nail bags on the side (e.g., Collections.sort())

Interface Evolution

- Libraries need to evolve, or they stagnate
  - Need a mechanism for compatibly evolving APIs
- New feature: *default methods*
  - Virtual interface method with default implementation
  - “default” is the dual of “abstract”
- Three simple rules for resolving inheritance conflicts
  - Superclasses win over superinterfaces
  - More specific interfaces win over less specific
  - After that, concrete classes must override

```
interface Collection<T> {
    default void forEach(Block<T> action) {
        for (T t : this)
            action.apply(t);
    }
}
```

Default Methods

- Similar to, but different from, C# extension methods
  - Java's default methods are *virtual* and *declaration-site*
  - Core principle: API owners should control their APIs
- Primary goal is API evolution
  - Inheritance rules directed at this primary goal
  - But very useful as an inheritance mechanism on its own!
- Wait, is this multiple inheritance in Java?
  - Java always had multiple inheritance of *types*
  - This adds multiple inheritance of *behavior*
    - But not of *state*, where most of the trouble comes from

It’s All About The Libraries

- Generally, we prefer to evolve the programming model through libraries
  - Time to market – can evolve libraries faster than language
  - Decentralized – more library developers than language developers
  - Risk – easier to change libraries, more practical to experiment
  - Impact – language changes require coordinated changes to multiple compilers, IDEs, and other tools
- Sometimes we reach the limits of what is practical to express in libraries, and need a little help from the language
  - A little help, in the right places, can go a long way!

Lambdas Enable Better APIs

- Lambda expressions *enable more powerful APIs*
  - Boundary between client and library is more permeable
  - Client provides bits of behavior to be mixed into execution (“what”)
  - Library remains in control of the computation (“how”)
  - Safer, exposes more opportunities for optimization
- Key effect on APIs is: *more composability*
  - Leads to better factoring, more regular client code, more reuse
- Lambdas in the language

  → can write better libraries

  → more readable, less error-prone user code

Example: Sorting

- If we want to sort a List today, we write a Comparator
- Many layers of nastiness here!
  - Conflates extraction of sort key with ordering of that key
  - “Collections” class required for helper methods
  - Syntactically verbose
    - Could replace with a lambda, but only gets us so far
    - Better to untangle the intertwined aspects
  - Status quo reduces opportunities for reuse

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

Example: Sorting

- Lambdas encourage finer-grained APIs
  - We add a method that takes a “key extractor” and returns Comparator
  - The comparing() method is one built for lambdas
    - Higher-order function
    - Eliminates redundancy, boilerplate

```
Comparator<Person> byLastName
    = Comparators.comparing(p -> p.getLastName());
```

```
Class Comparators {
    public static<T, U extends Comparable<? super U>>
    Comparator<T> comparing(Mapper<T, U> m) {
        return (x, y) -> m.map(x).compareTo(m.map(y));
    }
}
```

Example: Sorting

```
Comparator<Person> byLastName
    = Comparators.comparing(p -> p.getLastName());
Collections.sort(people, byLastName);

Collections.sort(people, comparing(p ->
p.getLastName()));

people.sort(comparing(p -> p.getLastName()));

people.sort(comparing(Person::getLastName));

people.sort(comparing(Person::getLastName).reverse());

people.sort(comparing(Person::getLastName)
            .compose(comparing(Person::getFirstName)));
```

Example – Sorting

- Default methods can enhance composability
  - Comparator.reverse(), Comparator.compose()
  - Default methods offer a “right place” to put certain code

```
interface Comparator<T> {
    int compare(T o1, T o2);

    default Comparator<T> reverse() {
        return (o1, o2) -> -(compare(o1, o2));
    }

    default Comparator<T> compose(Comparator<T> other) {
        return (o1, o2) -> {
             int cmp = compare(o1, o2);
             return (cmp != 0) ? cmp : other.compare(o1, o2);
        }
    }
}
```

```
Comparator<Person> byFirst = ...
Comparator<Person> byLast = ...

Comparator<Person> byFirstLast = byFirst.compose(byLast);
Comparator<Person> byLastDescending = byLast.reverse();
```

Bulk operations on Collections

- Compute sum of weights of blue shapes
  - Compose compound operations from basic building blocks
  - Each stage does one thing
  - Client code reads more like the problem statement
  - Structure of client code is less brittle
  - Less extraneous "noise" from intermediate results
    - No "garbage variables"
  - Library can use parallelism, out-of-order, laziness for performance

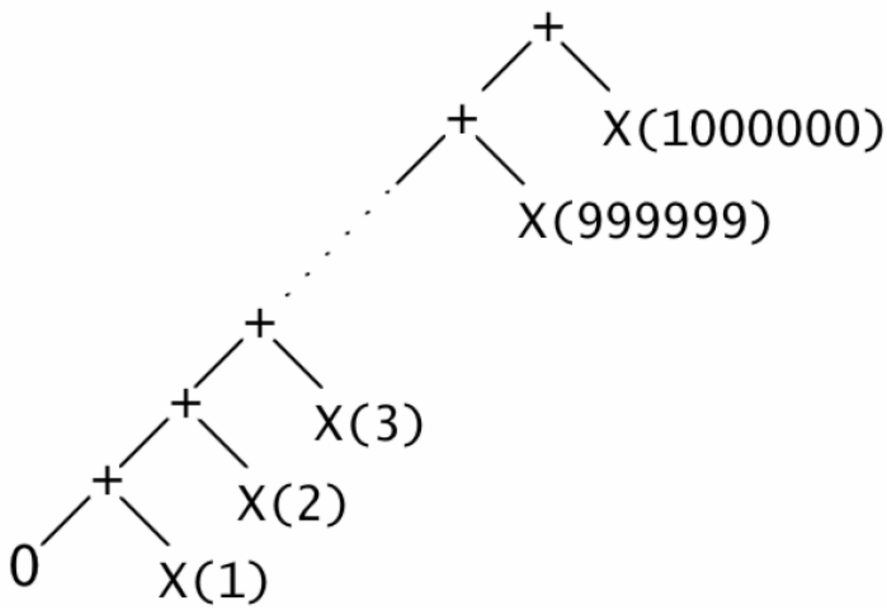
```
int sumOfWeight
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .map(Shape::getWeight)
            .sum();
```

Brief inspiration diversion

Brief inspiration diversion

Sequential Computation Tree

```
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```



ORACLE

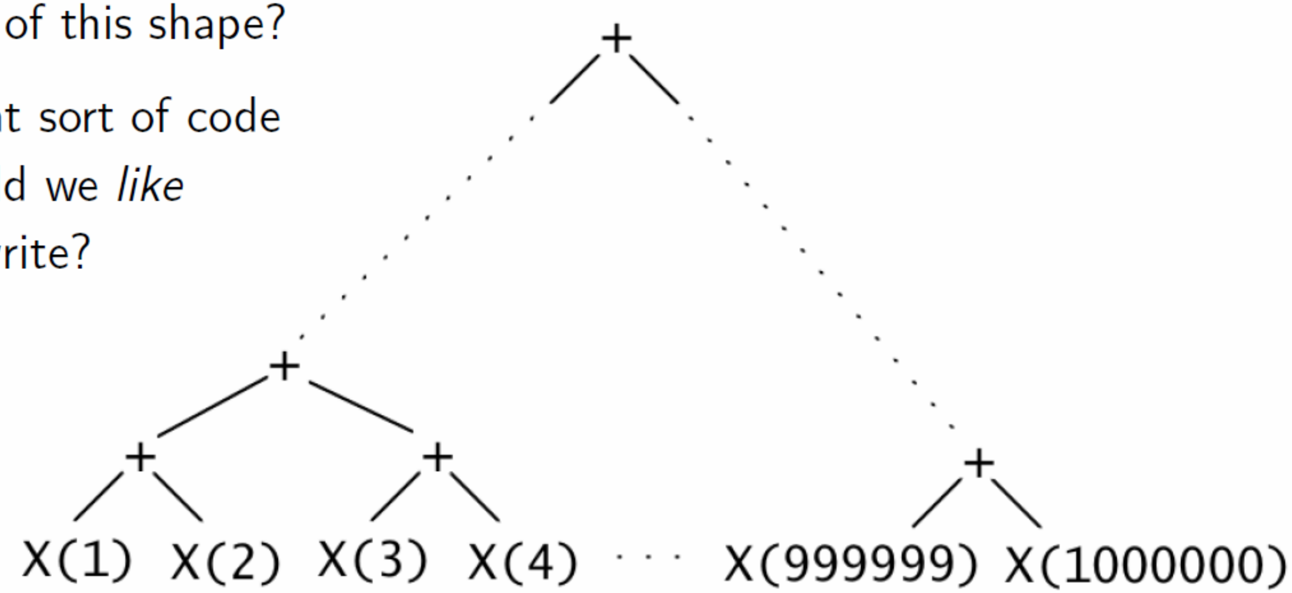
ORACLE

Brief inspiration diversion

Parallel Computation Tree

What sort of code
should we write
to get a computation
tree of this shape?

What sort of code
would we *like*
to write?



Streams

- To add bulk operations, we create a new abstraction, Stream (in package java.util.stream)
  - Key new library abstraction for JSR-335
  - Represents a stream of values
    - Not a data structure – doesn’t store the values
  - Source can be a Collection, array, generating function, IO
  - Encourages a “fluent” usage style
    - Supports operations like filter(), map(), reduce()
  - Retrofit stream() method on Collection
    - As well as: Reader.lines(), Random.ints(), String.chars(), etc
  - Easy to adapt any aggregate to be a Stream source

Streams

- What does this code do?

```
Set<Group> groups = new HashSet<>();
for (Person p : people) {
    if (p.getAge() >= 65)
        groups.add(p.getGroup());
}
List<Group> sorted = new ArrayList<>(groups);
Collections.sort(sorted, new Comparator<Group>() {
    public int compare(Group a, Group b) {
        return Integer.compare(a.getSize(), b.getSize())
    }
});
for (Group g : sorted)
    System.out.println(g.getName());
```

```
people.stream()
      .filter(p -> p.getAge() > 65)
      .map(p -> p.getGroup())
      .removeDuplicates()
      .sorted(comparing(g -> g.getSize())
      .forEach(g -> System.out.println(g.getName());
```

Parallelism

- Goal: easy-to-use parallel libraries for Java
  - Libraries can hide a host of complex concerns (task scheduling, thread management, load balancing)
- Goal: reduce conceptual and syntactic gap between serial and parallel expressions of the same computation
  - Right now, the serial code and the parallel code for a given computation don't look anything like each other
  - Fork-join (added in Java SE 7) is a good start, but not enough
- Goal: parallelism should be explicit, but unobtrusive

Fork/Join Parallelism

- JDK7 added general-purpose Fork/Join framework
  - Powerful and efficient, but not so easy to program to
  - Based on recursive decomposition
    - Divide problem into subproblems, solve in parallel, combine results
    - Keep dividing until small enough to solve sequentially
  - Tends to be efficient across a wide range of processor counts
  - Generates reasonable load balancing with no central coordination

Parallel Sum with Fork/Join

```
ForkJoinExecutor pool = new ForkJoinPool(nThreads);
SumProblem finder = new SumProblem(problem);
pool.invoke(finder);

class SumFinder extends RecursiveAction {
  private final SumProblem problem;
  int sum;

  protected void compute() {
    if (problem.size < THRESHOLD)
      sum = problem.solveSequentially();
    else {
      int m = problem.size / 2;
      SumFinder left, right;
      left = new SumFinder(problem.subproblem(0, m))
      right = new SumFinder(problem.subproblem(m, problem.size));
      forkJoin(left, right);
      sum = left.sum + right.sum;
    }
  }
}
```

```
class SumProblem {
  final List<Shape> shapes;
  final int size;

  SumProblem(List<Shape> ls) {
    this.shapes = ls;
    size = ls.size();
  }

  public int solveSequentially() {
    int sum = 0;
    for (Shape s : shapes) {
      if (s.getColor() == BLUE)
        sum += s.getWeight();
    }
    return sum;
  }
  public SumProblem subproblem(int start, int end) {
    return new SumProblem(shapes.subList(start, end));
  }
}
```

Parallel Sum with Streams

- Explicit but unobtrusive parallelism
  - All three operations fused into a single parallel pass
  - Works with ordinary, non-thread-safe collections
  - Extensible mechanism to work with any bulk container

```
int sumOfWeight
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .map(Shape::getWeight)
            .sum();
```

```
int sumOfWeight
    = shapes.parallelStream()
            .filter(s -> s.getColor() == BLUE)
            .map(Shape::getWeight)
            .sum();
```

Aggregation

- The sum() example is a special case of *reduction*
  - Elements $t_1, t_2, \ldots t_n$
  - An associative operator $\times$
  - Computes $t_1 \times t_2 \times \ldots \times t_n$
- We can extend the notion of reduction to mutable aggregations
  - Accumulate elements into a List
  - Concatenate strings into a StringBuffer
  - Classify elements and group into a Map
- Reductions work in parallel as well as sequentially
  - (As long as your operator is associative)

Aggregation

```
// Accumulate elements into a List
List<Person> list = people.stream()
                          .collect(Collectors.toList());

// Accumulate elements into a TreeSet
TreeSet<Person> list =
    people.stream()
          .collect(toCollection(TreeSet::new));

// Convert elements to comma-separated list
String joined =
    stream.map(Object::toString)
          .collect(toStringJoiner(", "))
          .toString();

// Find highest-paid employee
Employee highestPaid =
    employees.stream()
             .collect(maxBy(comparing(Employee::getSalary)));
```

Aggregation

```
// Group employees by department
Map<Department, List<Employee>> byDept =
    emps.stream()
        .collect(groupingBy(Employee::getDepartment));

// Find highest-paid employee by department
Map<Department, Employee> highestPaidByDept =
    emps.stream()
        .collect(groupingBy(Employee::getDepartment,
                            maxBy(comparing(Employee::getSalary))));
```

So … Why Lambda?

- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point to not have closures
  - Adding closures to Java is no longer a radical idea
- Provide libraries a path to multicore
  - Parallel-friendly APIs need internal iteration
  - Internal iteration needs a concise code-as-data mechanism
- Empower library developers
  - More powerful, flexible libraries
  - Higher degree of cooperation between libraries and client code
- Encourage better idioms
  - Gentle push towards a more functional style of programming

What's Next?

- There's plenty more work to do!
  - Dealing with primitive-reference divide
  - More parallel libraries
  - Value types and tuples
  - Computation on GPUs

Evolving Java
Project Lambda, and Beyond

Brian Goetz
Java Language Architect
Oracle Corporation