Evolving Java

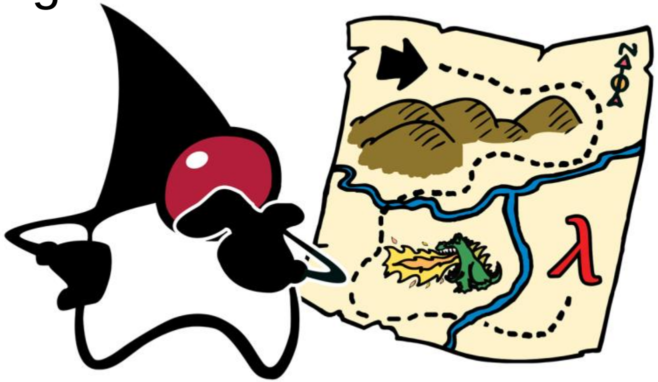
Brian Goetz
Java Language Architect, Oracle

ORACLE®

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

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
- Together, perhaps the *biggest upgrade ever* to the Java programming model



What is a Lambda Expression?

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

    ```
    (Object o) -> o.toString()
    ```

  - A *method reference* is a reference to an existing method

    ```
    Object::toString
    ```

  - Lambdas can refer to (capture) values from the enclosing lexical scope

    ```
    (Person p) -> p.getName().equals(name)
    ```

  - Compiler can often *infer* argument types from context

    ```
    p -> p.getName().equals(name)
    ```

- Lambdas and method refs allow you to *treat code as data*
  - Behavior can be stored in variables and passed to methods

ORACLE

Times Change

- In 1995, most popular languages did *not* support closures
- Today, Java is just about the last holdout that does not
  - 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

Problem: External Iteration

- Snippet takes the red blocks and colors them blue
- Uses foreach loop
  - Loop is *inherently sequential*
  - Client has to manage iteration
- This is called *external iteration*
- Foreach loop hides complex interaction between library and client
  - Iterable, iterator(), Iterator.next(), Iterator.hasNext()

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


ORACLE

Internal Iteration

- Re-written to use lambda and Collection.forEach
  - Not just a syntactic change!
  - Now the library is in control
  - This is *internal iteration*
  - More *what*, less *how*

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

- Library free to use parallelism, out-of-order execution, laziness
- Client passes behavior (lambda) into the API as data
- Enables API designers to build more powerful, expressive APIs
  - Greater power to abstract over behavior

What is the Type of a Lambda Expression?

- Most languages with lambdas have some notion of a *function type*
  - "Function from long to int"
  - Seemed reasonable (at first) to consider adding them to Java
- But…
  - JVM has no native representation of function type in VM type signatures
  - Obvious tool for representing function types is generics
    - But then function types would be erased (and boxed)
  - Is there a simpler alternative?

ORACLE

Functional Interfaces

- Historically have used single-method interfaces to represent functions
  - Runnable, Comparator, ActionListener
  - Let's give these a name: *functional interfaces*
  - And add some new ones like Predicate<T>, Consumer<T>, Supplier<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!"); };
```

- Compiler recognizes functional interfaces structurally
  - No syntax needed

Functional Interfaces

- “Just add function types” was obvious … and wrong
  - Would have interacted badly with erasure
  - 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

Lambdas Enable Better APIs

- Lambda expressions *enable delivery of more powerful APIs*
- The client-library boundary is more permeable
  - Client can provide bits of functionality to be mixed into execution
  - Client determines the *what*
  - Library remains in control of the *how*
- Safer, exposes more opportunities for optimization

Example: Sorting

- If we want to sort a List today, we'd write a Comparator
  - Comparator conflates *extraction of sort key* with *ordering* of that key
- Could replace Comparator with a lambda, but only gets us so far
  - Better to separate the two aspects

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


ORACLE

Example: Sorting

- Added static method Comparator.comparing(f)
  - Takes a “key extractor” function from T to some Comparable key
  - Returns a Comparator<T>
  - This is a *higher-order function* – functions in, functions out

```
interface Comparator {
    public static<T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> f) {
        return (x, y) -> f.apply(x).compareTo(f.apply(y));
    }
}
Comparator<Person> byLastName
      = Comparators.comparing(Person::getLastName);
```

Lambdas Enable Better APIs

- The comparing() method is one built for lambdas
  - Consumes an “extractor” function and produces a “comparator” function
  - Factors key extraction from comparison
  - Eliminates redundancy, boilerplate
- Key effect on APIs is: *more composability*
  - Centralize manipulation of Comparators in one place
  - Leads to better factoring, more regular client code, more reuse
- Lambdas in the languages

  → can write better libraries

  → more readable, less error-prone user code

 | |

ORACLE

Problem: Interface Evolution

- The 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())

ORACLE

Default Methods

- 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”

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

- Lets us compatibly evolve libraries over time
  - Default implementation provided in the interface
  - Subclasses can override with better implementations
  - Adding a default method is source- and binary-compatible

Default Methods

- Huh? 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
- Primary goal is interface evolution
- Compared to C# extension methods
  - Java's default methods are *virtual* and *declaration-site,* not *static* and *use-site*
- Compared to Scala's Traits
  - Java interfaces are stateless (more like Fortress' Traits)
- How do we resolve conflicts between declarations in multiple supertypes?
  - Three simple rules

Rule #1: Class Wins

- If a class can inherit a method from a superclass and a superinterface, prefer the superclass method
  - Defaults *only* considered if no method declared in superclass chain
  - True for both concrete and abstract superclass methods
- Ensures compatibility with pre-Java-8 inheritance
  - Any call site that linked under previous rules links to the same target
- Otherwise…

Rule #2: Subtypes Win

- If a class can inherit a method from two interfaces, and one more specific than (a subtype of) the other, prefer the more specific
  - An implementation in List would take precedence over one in Collection
- The shape of the inheritance tree doesn't matter
  - Only consider the set of supertypes, not order in which they are inherited
- Otherwise…

Rule #3: There is No Rule 3

- If rule #1 does not apply, and rule #2 does not yield a *unique, most specific default-providing interface…*
  - Implement the method yourself (or explicitly reabstract it)
  - Implementation can delegate to inherited implementation with new syntax **`A.super.m()`**

```
interface A {
    default void m() { ... }
}
interface B {
    default void m() { ... }
}
class C implements A, B {
    // Must implement/reabstract m()
    void m() { A.super.m(); }
}
```


Diamonds – No Problem

- Diamonds do not pose a problem for behavior inheritance
  - More problematic for state inheritance
- For D, there is a unique, most-specific default-providing interface – A
  - D inherits m() from A, via two paths
  - "Redundant" inheritance does not affect the resolution

```
interface A {
    default void m() { ... }
}
interface B extends A { }
interface C extends A { }
class D implements B, C { }
```


Example – Evolving Interfaces

- Default methods are instance methods
  - Type of 'this' is the declaring interface
  - So default implementation can invoke methods from enclosing interface
    - Such as iterator()
  - Adding a new method with default is source- and binary-compatible

```
interface Collection<E> {
    default boolean removeIf(Predicate<? super E> filter) {
        boolean removed = false;
        Iterator<E> it = iterator();
        while(it.hasNext()) {
            if(filter.test(each.next())) {
                it.remove();
                removed = true;
            }
        }
        return removed;
    }
}
```


Example – “Optional” Methods

- Default methods can reduce implementation burden
- Most implementations of Iterator don’t provide a useful remove()
  - So why make developer write one that just throws?
  - In this way, default methods can be used to declare “optional” methods
  - Adding a default to an existing method is source- and binary- compatible

```
interface Iterator<T> {
    boolean hasNext();

    T next();

    default void remove() {
        throw new UnsupportedOperationException();
    }
}
```


Example – Combinators

- Comparator.reversed() – reverses sort order of a Comparator
  - Instance (default) method on Comparator
    - Invokes compare() with arguments in reverse order
  - Also added Comparator.thenComparing() instance methods

```
interface Comparator {
    default Comparator<T> reversed() {
        return (o1, o2) -> compare(o2, o1);
    }
}
Comparator<Person> byLastNameDescending
      = Comparator.comparing(Person::getLastName)
                  .reversed();
```


Putting it together: Sorting

```
Comparator<Person> byLastName
    = Comparator.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).reversed());

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

Bulk operations on Collections

- The lambda version of the "shapes" code can be further decomposed
  - Using Streams framework (java.util.stream) for aggregate operations
- "Color the red blocks blue" can be decomposed into filter+forEach

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

⬇

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


Bulk operations on Collections

- Collect the blue Shapes into a List

```
List<Shape> blueBlocks
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .collect(Collectors.toList());
```

- If each Shape lives in a Box, find Boxes containing a blue shape

```
Set<Box> hasBlueBlock
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .map(Shape::getContainingBox)
            .collect(Collectors.toSet());
```


ORACLE

Bulk operations on Collections

- Compute sum of weights of blue shapes

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

Bulk operations on Collections

- The new bulk operations are expressive and composable
  - 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
  - Library can use parallelism, out-of-order, laziness for performance

ORACLE

Streams

- To add bulk operations, we create a new abstraction, Stream
  - Represents a stream of values
    - Not a data structure – doesn’t store the values
  - Source can be a Collection, array, generating function, I/O…
  - Operations that produce new streams are lazy
  - Encourages a “fluent” usage style
  - Efficient – does a single pass on the data

```
collection.stream()
          .filter(f -> f.isBlue())
          .map(f -> f.getBar())
          .forEach(System.out::println);
```

Streams

```
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Group>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

- Or…

```
txns.stream()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sort(comparing(Seller::getName))
    .forEach(s -> System.out.println(s.getName());
```


Comparing Approaches

Imperative	Streams
Code deals with individual data items	Code deals with data set
Focused on *how*	Focused on *what*
Code doesn’t read like the problem statement	Code reads like the problem statement
Steps mashed together	Well-factored
Leaks extraneous details	No “garbage variables”
Inherently sequential	Same code can be sequential or parallel

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 has a 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 Collections

- Parallel sum-of-sizes with bulk collection operations

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

  – Explicit but unobtrusive parallelism
  – All three operations fused into a single parallel pass

So … Why Lambda?

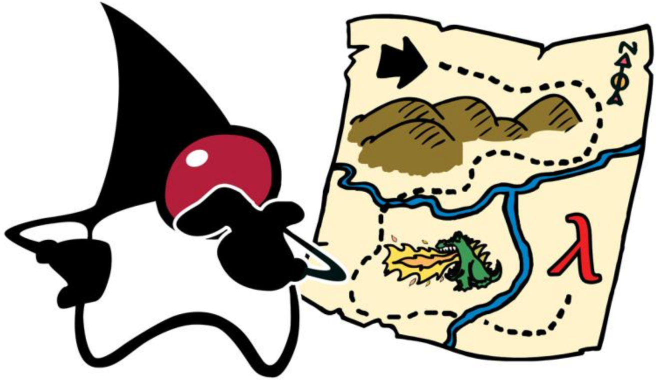
- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point to not have closures
  - Adding closures to Java is no longer a radical idea
- Provide libraries a path to multicore
  - Parallel-friendly APIs need internal iteration
  - Internal iteration needs a concise code-as-data mechanism
- Empower library developers
  - More powerful, flexible libraries
  - Higher degree of cooperation between libraries and client code
  - Better libraries means more expressive, less error-prone code for users!

 | |

ORACLE

Where are we now?

- Developer Preview available at https://jdk8.java.net/download.html
  - Download and try it out!
- In JCP Public Review
- Shipping with Java SE 8



ORACLE