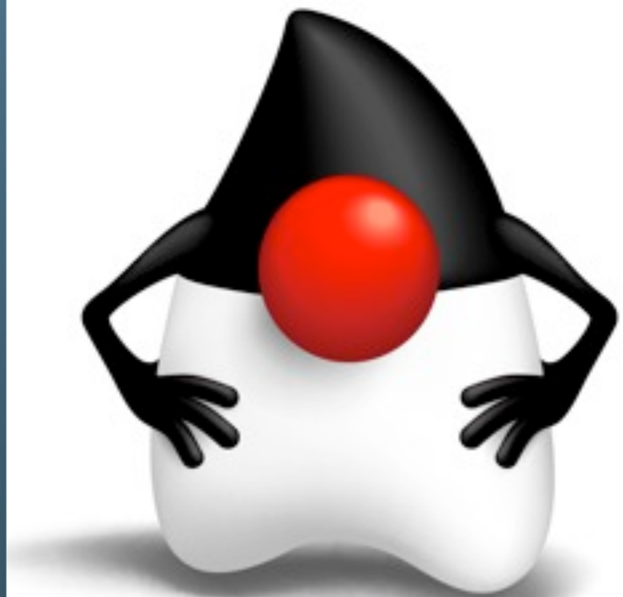
Project Lambda in Java SE 8

Daniel Smith
Java Language Designer



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.

The Java Programming Language

- Around 9,000,000 developers worldwide
- 17 years old
- 4 major revisions (1996, 2000, 2005, 2013...)
- [Insert staggering number] of companies very heavily invested
- Formally standardized and evolved via community

 | |

- The scope of the Java language is a huge opportunity for the forces of good to move the state of programming forward.
- But there's also a very strong commitment to legacy support, and a disincentive to messing things up.

Evolving a Major Language

- Adapting to change
- Righting what's wrong
- Maintaining compatibility
- Preserving the core

 |

Two ideas in this talk: 1) what we're doing in Java 8, along with 2) a meta discussion of how we arrived here
- "Change": we haven't discovered the perfect language yet, and when we do, conditions will change anyway
- "Wrong": the spec has warts, and they're not good for user experience or implementation consistency
- "Compatibility": unusually low tolerance for change between versions
- "Core": can't alienate the base in a quest for something better

Project Lambda: Function Values in Java

Code as Data

```
(define f
  (lambda (x) (* x x)))

(map nums f)
```

```
Object subclass: Widget [
  draw: canvas [ ... ]
  click [ ... ]
]

gui add:(Widget new).
```

 | |

- Both functional and object-oriented languages rely fundamentally on the “code as data” concept. (Here, passing a function to ‘map’ and an object to ‘gui’.)
- Compare and contrast...
- They have a lot in common, and each can be easily viewed from the other’s perspective.
- But the approaches are different: functions are small, classes are big.

Status Quo in Java 2

```
interface Runnable {
  void run();
}

Thread hello = new Thread(new Runnable() {
  public void run() {
    System.out.println(“Hello, world!”);
  }
});
```

 |

- Does Java already have functions?
- We’ve had Runnable since Java 1, and anonymous classes since Java 2. These combine to make it easy to pass a function to the Thread constructor.

Status Quo in Java 5

```
interface Predicate<T> {
  boolean accept(T arg);
}

lines.removeAll(new Predicate<String>() {
  public boolean accept(String line) {
    return line.startsWith("#");
  }
});
```

 | | Insert Information Protection Policy Classification from Slide 13

- Java 5 added generics, which make it easier to define interfaces representing general-purpose functions.
- But there is no standard Predicate interface, probably because you can't convince people to write code like this. Too little content in the boilerplate.

What We Wish It Looked Like

```
interface Predicate<T> {
  boolean accept(T arg);
}

lines.removeAll(line -> line.startsWith(“#”));
```

 | | Insert Information Protection Policy Classification from Slide 13

Problems with anonymous classes:
- Lots of boilerplate
- Everything is explicit
- Multiple lines
- Less obvious: puts stress on heap (class loading, object creation) and disk (lots of class files)

Why Functions in Java? Better Libraries

- *Lots* of applications...
- Our priorities:
  - Collections
  - Concurrency

```
public class ForkBlur extends RecursiveAction {
  private int[] mSource;
  private int mStart;
  private int mLength;
  private int[] mDestination;

  public ForkBlur(int[] src, int start, int length, int[] dst) {
    mSource = src;
    mStart = start;
    mLength = length;
    mDestination = dst;
  }

  // Average pixels from source, write results into destination.
  protected void computeDirectly() {
    for (int index = mStart; index < mStart + mLength; index++) {
      mDestination[index] = blur(index, mSource);
    }
  }

  protected static int sThreshold = 10000;

  protected void compute() {
    if (mLength < sThreshold) {
      computeDirectly();
      return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
              new ForkBlur(mSource, mStart + split, mLength - split, mDestination));
  }

}
```

 |

- Functional programmers know all sorts of situations where lightweight functions come in handy.
- As a start, we want our collections library to be more convenient/declarative and parallelizable.
- Example: fork-join is powerful, but a naive user is faced with tons of boilerplate just to express a simple parallel computation; we can’t really do much better without lightweight functions.

Java 8 Language Concepts & Features

- Lambda expressions
- Functional interfaces
- Target typing
- Method references
- Default methods

 | | Insert Information Protection Policy Classification from Slide 13

- Five major new language features & concepts that will facilitate powerful new Java programming patterns.
- Taking the best high-impact ideas we've seen or invented and fitting them into the Java language.

Lambda Expressions

Lambda Expressions

```
x -> x+1

(s,i) -> s.substring(0,i)

(Integer i) -> list.add(i)

() -> System.out.print("x")

cond -> cond ? 23 : 57
```

```
widget -> {
  if (flag) widget.poke();
  else widget.prod();
}

(int x, int y) -> {
  assert x < y;
  return x*y;
}
```

 |

- 0, 1, or multiple parameters
- Parameter types can be inferred or explicit
- Bodies can be expressions or blocks
- Block bodies are like methods -- local return
- Minimal delimiters

Variable Capture

- Lambdas can refer to variables declared outside the body
- These variables can be final or "effectively final"
  - Works for anonymous classes, too

```
void cut(List<String> l,
                  int len) {

    l.updateAll(s ->
       s.substring(0, len));

}
```

 | | Insert Information Protection Policy Classification from Slide 13

- Lambdas can refer to variables declared outside the body
- Example: declarations and uses of variables are bold
- This is one big reason you would want a local construct (rather than declaring a method)
- Anonymous classes have always required captured variables to be 'final'
- Captured vars still have to be fixed, but don't have to be declared final

Meaning of Names in Lambdas

- Anonymous classes introduce a new “level” of scope
  - ‘`this`’ means the inner class instance
  - ‘`ClassName.this`’ is used to get to the enclosing class instance
  - Inherited names can shadow outer-scope names
- Lambdas reside in the same “level” as the enclosing context
  - this refers to the enclosing class
  - No new names are inherited
  - Like local variables, parameter names can’t shadow other locals

 |

- Anonymous classes have a heavyweight resolution strategy
- Lambdas have a lightweight resolution strategy

Functional Interfaces

 |

Function Types in Java?

```
String -> int

(String, int, boolean) -> List<? extends Integer>

(String, Number) -> Class<?> throws IOException
```


- What is the type of a lambda expression? We need function types...
- But this isn't going to work!
- Imagine these types in a method signature or as a collection type argument

Function Types in Java: Functional Interfaces

java.util.concurrent

Interface Callable<V>

Type Parameters:

`V` - the result type of method `call`

All Known Subinterfaces:

JavaCompiler.CompilationTask

```
public interface Callable<V>
```

A task that returns a result and may throw an exception. Implementors define a single method with no arguments called `call`.

The `Callable` interface is similar to `Runnable`, in that both are designed for classes whose instances are potentially executed by another thread. A `Runnable`, however, does not return a result and cannot throw a checked exception.

The `Executors` class contains utility methods to convert from other common forms to `Callable` classes.

Since:

1.5

See Also:

`Executor`

Method Summary

Methods

Modifier and Type	Method and Description
`V`	`call()` Computes a result, or throws an exception if unable to do so.

 | | Insert Information Protection Policy Classification from Slide 13

Maybe we don’t need something new.

Common Existing Functional Interfaces

- java.lang.Runnable
- java.util.concurrent.Callable<V>
- java.security.PrivilegedAction<T>
- java.util.Comparator<T>
- java.io.FileFilter
- java.nio.file.PathMatcher
- java.lang.reflect.InvocationHandler
- java.beans.PropertyChangeListener
- java.awt.event.ActionListener
- javax.swing.event.ChangeListener

 |

- Already defined
- Already used extensively in APIs

Attributes of Functional Interfaces

- Parameter types
- Return type
- Method type arguments
- Thrown exceptions
- An expressive, reifiable type name (possibly generic)
- An informal contract

 | | Insert Information Protection Policy Classification from Slide 13

- An interface declaration takes up just enough space to give a name and a description to a function type
- Nominal typing is fundamental in Java

Shiny New Functional Interfaces*

- java.util.functions.Predicate<T>
- java.util.functions.Factory<T>
- java.util.functions.Block<T>
- java.util.functions.Mapper<T, R>
- java.util.functions.BinaryOperator<T>

* Names and concepts in libraries are still tentative

 |

We define some functional interfaces fitting basic shapes in our libraries, for both our own use and reuse by others.

Declare Your Own

```
/** Creates an empty set. */
public interface SetFactory {
  <T> Set<T> create();
}

/** Performs a blocking, interruptible action. */
public interface BlockingTask<T> {
  <T> T run() throws InterruptedException;
}
```

 | | Insert Information Protection Policy Classification from Slide 13

- The standard API can’t cover everything, and it doesn’t need to.
- Notice the informal contracts.

Target Typing

 |

Bridging the gap between lambda expressions and functional interfaces...

Assigning a Lambda to a Variable

```
// Runnable: void run()
Runnable r =
            () -> System.out.println("hi");

// Predicate<String>: boolean test(String arg)
Predicate<String> pred =
            s -> s.length() < 100;
```

 |

- A lambda can be assigned to a variable of a functional interface type.
- Bold highlights the target type and the matching expression.
- The type of the lambda expression IS the target type -- it's an implicit part of the expression.
- Implicit parameter types are inferred from the target type.

Target Typing Errors

```
Object o =
            () -> System.out.println(“hi”);

// Predicate<String>: boolean test(String arg)
Predicate<String> pred =
            () -> System.out.println(“hi”);
```

 | |

- Lambdas are meaningless without a functional interface target type.
- Parameters and return have to match the target type.

Target Typing in Java 7

```
long[][] arr =
            { { 1, 2, 3 }, { 4, 5, 6 } };

List<? extends Number> nums =
            Collections.emptyList();

Set<Map<String, Object>> maps =
            new HashSet<>();
```

 | | Insert Information Protection Policy Classification from Slide 13

- The idea of interpreting an expression based on context is NOT new.
- But it hasn't been formalized very well before.
- And we're stepping it up to a new level.

Target Typing for Invocations

```
class Thread {
  public Thread(Runnable r) { ... }
}

// Runnable: void run()
new Thread(() -> System.out.println("hi"));
```

 |

A method can ALSO provide a target type (combining information from a declaration and a use).

Target Typing for Invocations

```
interface Stream<T> {
  Stream<T> filter(Predicate<T> pred);
}

Stream<String> strings = ...;

// Predicate<T>: boolean test(T arg)
strings.filter(s -> s.length() < 100);
```

 |

The target type in a method might depend on generic instantiation (combining information from a declaration, a parameterized type, and a use).

A Recipe for Disaster
(Or: A Recipe for Awesome)

- Target typing
- Overload resolution
- Type argument inference

```
<T> int m(Predicate<T> p);
int m(FileFilter f);
<S,T> int m(Mapper<S,T> m);

m(x -> x == null);
```

 | | Insert Information Protection Policy Classification from Slide 13

- When we get a target type from a set of overloaded, generic methods, crazy stuff happens.
- Sometimes, it's just ambiguous, but sometimes we can (and should) do much better.
- This is probably where we've spent the majority of our language design time.
- Bonus: new and improved inference features.

Other Target Typing Contexts

```
Object o =
        (Runnable) () -> System.out.println("hi");

Runnable r =
        condition() ? null : () -> System.gc();

Mapper<String, Runnable> m =
        s -> () -> System.out.println(s);
```

 |

- A cast can provide an explicit target type.
- Conditional expression pass down target types.
- Lambdas can be nested.

Method References

Boilerplate Lambdas

```
(x, y, z) -> Arrays.asList(x, y, z)

(str, i) -> str.substring(i)

() -> Thread.currentThread().dumpStack()

(s) -> new File(s)
```

 |

- Sometimes, the function you want is just a method that's already defined somewhere.
- There's still some boilerplate involved in this usage.

Method (and Constructor) References

```
(x, y, z) -> Arrays.asList(x, y, z)
Arrays::asList
(str, i) -> str.substring(i)
String::substring
() -> Thread.currentThread().dumpStack()
Thread.currentThread()::dumpStack
(s) -> new File(s)
File::new
```

 | | Insert Information Protection Policy Classification from Slide 13

- Static method reference
- Instance method reference
- Bound method reference
- Constructor reference

Resolving a Method Reference

- Target type provides argument types
- Named method is searched for using those argument types
  - Searching for an instance method, the first parameter is the receiver
- Return type must be compatible with target return

 |

Since methods can be overloaded, the method being referenced depends on the target type.

Method References & Generics

```
Mapper<Byte, Set<Byte>> m1 = Collections::singleton;

// SetFactory: <T> Set<T> create()
SetFactory f2 = Collections::emptySet;

Mapper<Queue<Float>, Float> m2 = Queue::peek;

Factory<Set<String>> f3 = HashSet::new;
```

 |

- Type arguments are inferred, just like invocations.
- But methods can be referred to generically, too, given an appropriate target type.
- The class name part of the reference doesn't need type arguments.
- Similarly with constructor references: class type arguments can be inferred.

Default Methods

 | |

We’ve got all these great new features, now we need to get people to use them...

Evolving APIs

New concrete methods: Good

```
abstract class Widget {
  abstract double weight();
  abstract double volume();

  double density() {
    return weight()/volume();
  }
}
```

New abstract methods: Bad

```
interface Widget {
  double weight();
  double volume();

  double density();
}
```

 |

Workaround: Garbage Classes

- Not really a class
- Non-idiomatic invocation syntax
- Non-virtual

```
class Widgets {

  static double density(Widget w) {
    return w.weight()/w.volume();
  }

}
```

 | |

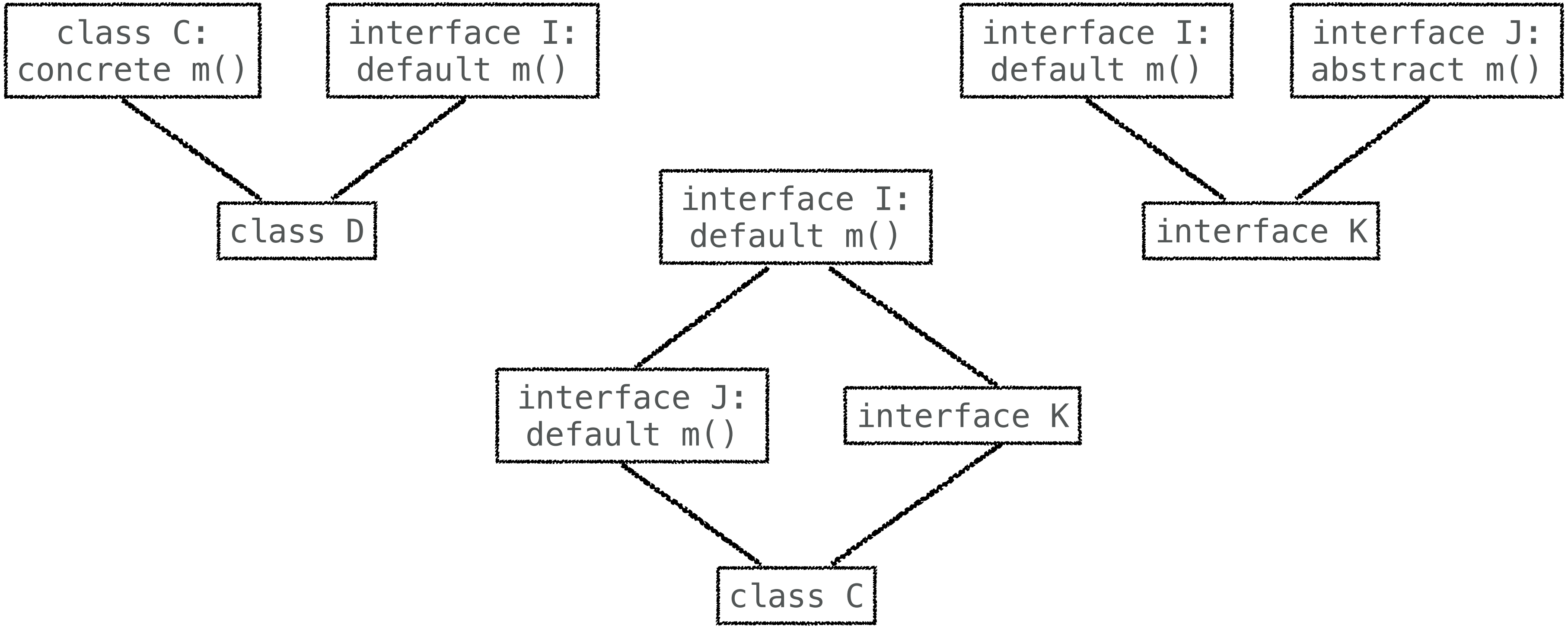
Default Methods: Code in Interfaces

```
interface Widget {
  double weight();
  double volume();

  default double density() {
    return weight()/volume();
  }
}
```

 | |

Multiple Inheritance?



- Multiple inheritance of _behavior_, but not _state_.
- Resolve in the “obvious” way whenever possible, but avoid surprises
- Intuitions: class beats interface; overrider beats overridden

Evolving the Java Standard API

```
interface Enumeration<E> extends Iterator<E> {
  boolean hasMoreElements();
  E nextElement();

  default boolean hasNext() { return hasMoreElements(); }
  default E next() { return getNext(); }
  default void remove() { throw new UnsupportedOperationException(); }

  default void forEachParallel(Block<T> b) { ... }
}
```

 |

Summary

 | |

Goals for Project Lambda

- Make dramatic & necessary enhancements to the programming model
- Smooth some rough edges in the language
- Preserve compatibility
- Maintain the essence of the Java language

 | |

- Enhancements: lambda expressions, target typing, default methods
- Rough edges: variable capture, type inference
- Essence: functional interfaces, default methods

Learning More

- **OpenJDK:** openjdk.java.net/projects/lambda
- **JSR 335:** www.jcp.org/en/jsr/detail?id=335
- **Me**: daniel.smith@oracle.com

- Download it and try it out!

 |

Now is a great time to get feedback from real-world usage.