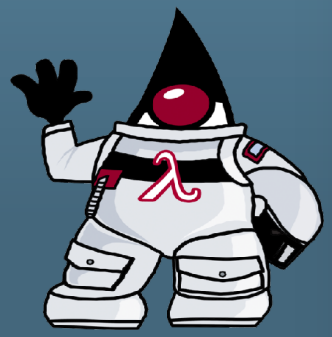

Java™

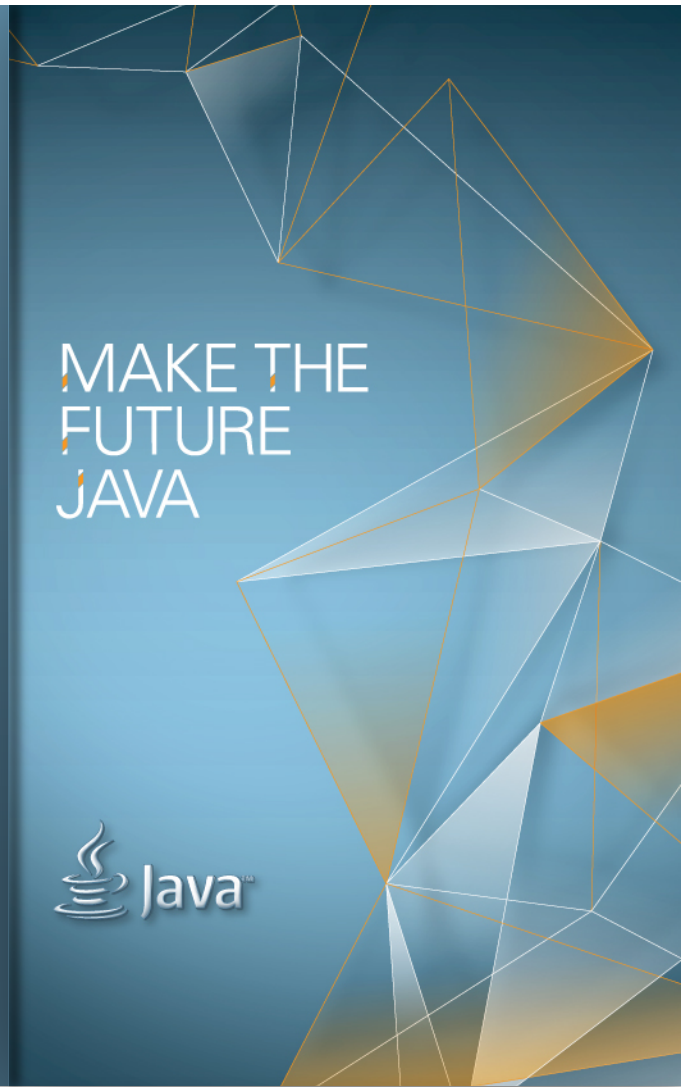
ORACLE

Lambda Expressions in Java

Simon Ritter
Java Technology Evangelist
Twitter: @speakjava

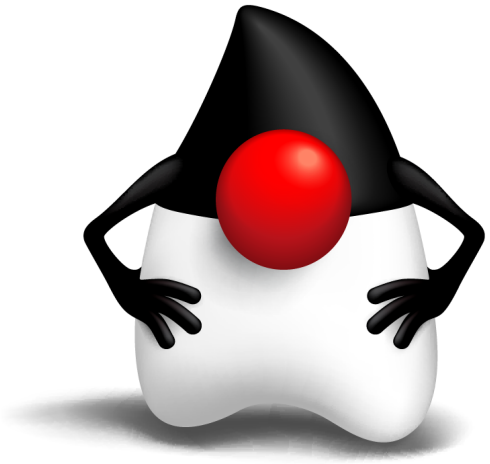


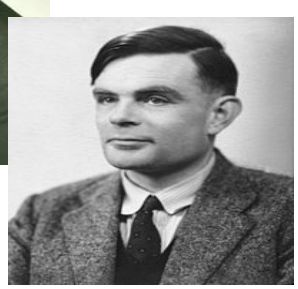
With thanks to Brian Goetz

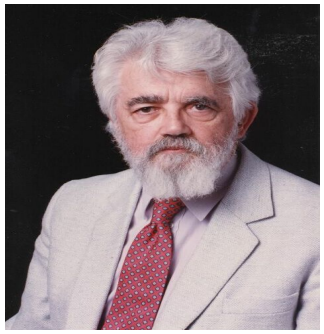


The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development,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

A Bit Of Background







1930/40’s

1950/60’s

1970/80’s

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 353 March 10, 1976

LAMBDA

THE ULTIMATE IMPERATIVE

by

Guy Lewis Steele Jr. and Gerald Jay Sussman

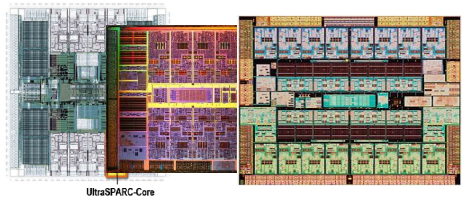
Images – wikipedia / bio pages

Java™ ORACLE®

Computing Today

- Multicore is now the default
  - Moore's law means more cores, not faster clockspeed
- We need to make writing parallel code easier
- All components of the Java SE platform are adapting
  - Language, libraries, VM

360 Cores
2.8 TB RAM
960 GB Flash
InfiniBand
..

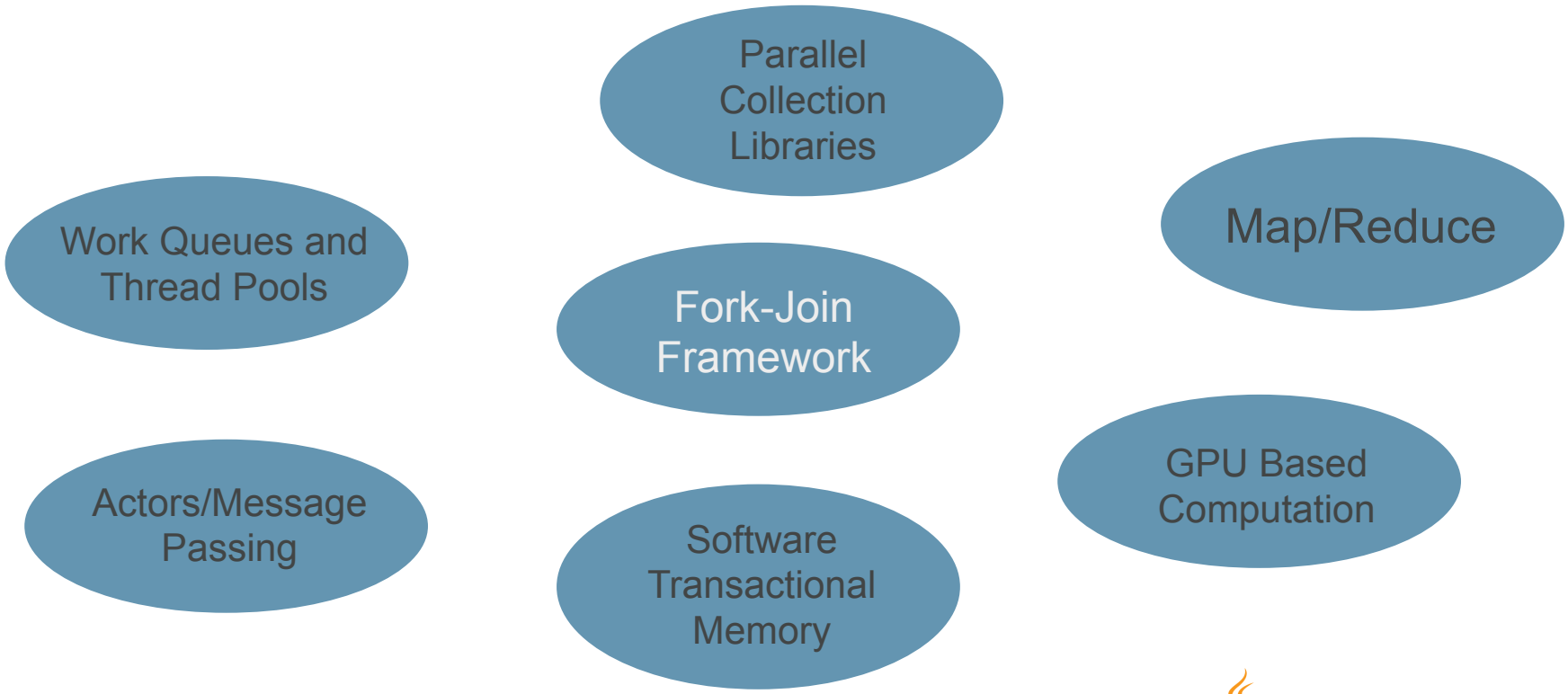


Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Herb Sutter
http://www.gotw.ca/publications/concurrency-ddj.htm
http://drdobbs.com/high-performance-computing/225402247
http://drdobbs.com/high-performance-computing/219200099

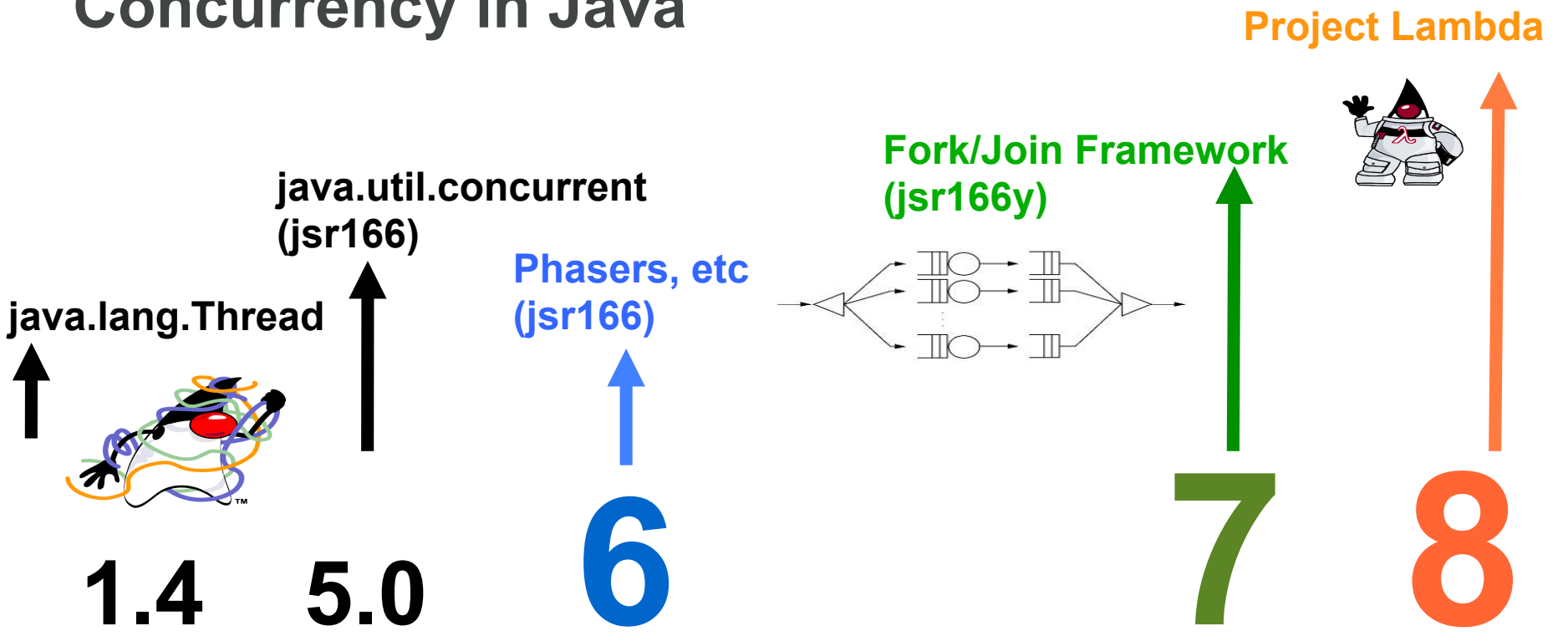
Java™ ORACLE

Data Parallelism Solutions



Java™ ORACLE®

Concurrency in Java



Java™ ORACLE®

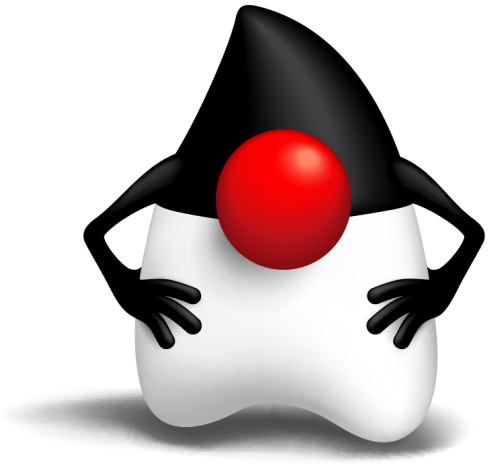
Goals For Better Parallelism In Java

- Easy-to-use parallel libraries
  - Libraries can hide a host of complex concerns
    - task scheduling, thread management, load balancing, etc
- Reduce conceptual and syntactic gap between serial and parallel expressions of the same computation
  - Currently serial code and parallel code for a given computation are very different
    - Fork-join (added in Java SE 7) is a good start, but not enough

It's All About The Libraries

- Most of the time, we should 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
- But sometimes we reach the limits of what is practical to express in libraries, and need some help from the language

Bringing Lambdas To Java



The Problem: External Iteration

```
List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.gradYear == 2011) {
    if (s.score > highestScore) {
      highestScore = s.score;
    }
  }
}
```

- Client controls iteration
- *Inherently serial:* iterate from beginning to end
- Not thread-safe because business logic is stateful (mutable accumulator variable)

Internal Iteration With Inner Classes

More Functional, Fluent and Monad Like

```
SomeList<Student> students = ...
double highestScore =
  students.filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011;
    }
  }).map(new Mapper<Student,Double>() {
    public Double extract(Student s) {
      return s.getScore();
    }
  }).max();
```

- Iteraction, filtering and accumulation are handled by the library
- Not inherently serial – traversal *may* be done in parallel
- Traversal *may* be done lazily – so one pass, rather than three
- Thread safe – client logic is stateless
- High barrier to use
  - Syntactically ugly

Internal Iteration With Lambdas

```
SomeList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

- More readable
- More abstract
- Less error-prone
- No reliance on mutable state
- Easier to make parallel

Lambda Expressions

Some Details

- Lambda expressions are anonymous functions
  - Like a method, has a typed argument list, a return type, a set of thrown exceptions, and a body

```
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

Lambda Expression Types

- Single-method interfaces used extensively to represent functions and callbacks
  - Definition: a *functional interface* is an interface with one method (SAM)
  - Functional interfaces are identified structurally
  - The type of a lambda expression will be a functional interface

```
interface Comparator<T>  { boolean compare(T x, T y); }
interface FileFilter     { boolean accept(File x); }
interface DirectoryStream.Filter<T> { boolean accept(T x); }
interface Runnable       { void run(); }
interface ActionListener { void actionPerformed(…); }
interface Callable<T>    { T call(); }
```

Target Typing

- A lambda expression is a way to create an instance of a functional interface
  - Which functional interface is inferred from the context
  - Works both in assignment and method invocation contexts
    - Can use casts if needed to resolve ambiguity

```
Comparator<String> c = new Comparator<String>() {
    public int compare(String x, String y) {
        return x.length() - y.length();
    }
};
```

```
Comparator<String> c = (String x, String y) -> x.length() - y.length();
```

Local Variable Capture

- Lambda expressions can refer to *effectively final* local variables from the enclosing scope
  - Effectively final means that the variable meets the requirements for final variables (e.g., assigned once), even if not explicitly declared final
  - This is a form of type inference

```
void expire(File root, long before) {
    ...
    root.listFiles(File p -> p.lastModified() <= before);
    ...
}
```

Lexical Scoping

- The meaning of names are the same inside the lambda as outside
  - A ‘this’ reference – refers to the enclosing object, not the lambda itself
  - Think of ‘this’ as a final predefined local

```
class SessionManager {
  long before = ...;

  void expire(File root) {
     ...
     // refers to ‘this.before’, just like outside the lambda
     root.listFiles(File p -> checkExpiry(p.lastModified(), before));
  }

  boolean checkExpiry(long time, long expiry) { ... }
}
```

Type Inferrence

- Compiler can often infer parameter types in lambda expression

```
Collections.sort(ls, (String x, String y) -> x.length() - y.length());
```

```
Collections.sort(ls, (x, y) -> x.length() - y.length());
```

- Inferrence based on the target functional interface's method signature
- Fully statically typed (no dynamic typing sneaking in)
  - More typing with less typing

Method References

- Method references let us reuse a method as a lambda expression

```
FileFilter x = new FileFilter() {
  public boolean accept(File f) {
    return f.canRead();
  }
};
```

```
FileFilter x = (File f) -> f.canRead();
```

```
FileFilter x = File::canRead;
```


Putting it all together

With a little help from the libraries

- Make common idioms more expressive, reliable, and compact

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

More concise
More abstract
More reuse
More object-oriented

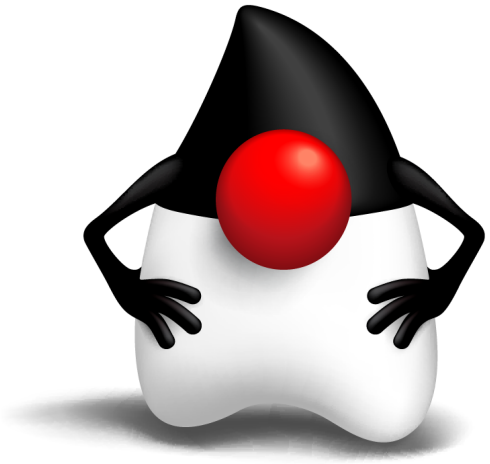
```
Collections.sort(people,
                 comparing(p -> p.getLastName()));
```


Lambda Expressions In Java

Advantages

- Developers primary tool for computing over aggregates is the for loop
  - Inherently serial
  - We need internal iteration
- Useful for many libraries, serial and parallel
- Adding Lambda expressions to Java is no longer a radical idea

Library Evolution



Library Evolution

The Real Challenge

- Adding lambda expressions is a big language change
  - If Java had them from day one, the APIs would definitely look different
  - Adding lambda expressions makes our aging APIs show their age even more
- Most important APIs (Collections) are based on interfaces
  - How to extend an interface without breaking backwards compatability
- Adding lamabda expressions to Java, but not upgrading the APIs to use them, would be silly
- Therefore we also need better mechanisms for *library evolution*

Library Evolution Goal

- Requirement: aggregate operations on collections
  - New methods on Collections that allow for bulk operations
  - Examples: filter, map, reduce, forEach, sort
  - These can run in parallel (return Stream object)

```
int heaviestBlueBlock =
    blocks.filter(b -> b.getColor() == BLUE)
          .map(Block::getWeight)
          .reduce(0, Integer::max);
```

- This is problematic
  - Can't add new methods to interfaces without modifying all implementations
  - Can't necessarily find or control all implementations

API Evolution Is A First-Class Problem

- Interfaces are a double-edged sword
  - Once published, cannot add to them without breaking existing implementations
- Fundamental problem: can't evolve interface-based APIs
  - The older an API gets, the more obvious the decay
  - We're a victim of our own success; Java has lots of old APIs
  - Lots of bad choices here
    - Let the API stagnate
    - Try and replace it in entirety – every few years!
    - Nail bags on the side (e.g., Collections.sort())
- Key Principle: burden of API evolution should fall to implementors, not users
  - Solutions that require users to permanently cruft up their code to use new features are undesirable

Solution: Virtual Extension Methods

AKA Defender Methods

- Specified in the interface
- From the caller's perspective, just an ordinary interface method
- List class provides a default implementation
  - Default is only used when implementation classes do not provide a body for the extension method
  - Implementation classes can provide a better version, or not
- Drawback: requires VM support

```
interface List<T> {
    void sort(Comparator<? super T> cmp)
        default { Collections.sort(this, cmp); };
}
```

Virtual Extension Methods

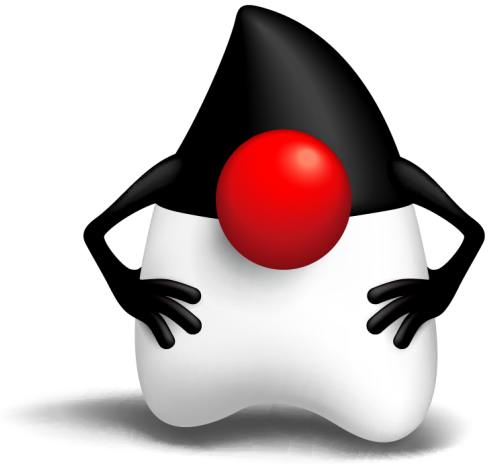
Stop right there!

- Err, isn't this implementing multiple inheritance for Java?
  - Yes, but Java already has multiple inheritance of *types*
  - This adds multiple inheritance of *behavior* too
  - But not *state*, which is where most of the trouble is
  - Though can still be a source of complexity due to separate compilation and dynamic linking

Compatibility Goals

- Extension methods are about being able to *compatibly* evolve APIs
  - Motivated by having APIs in serious need of evolution
- Compatibility has multiple faces
  - Source compatibility
  - Binary compatibility
- Primary concern is *adding new methods with defaults* to existing interfaces
  - *Without* necessarily recompiling the implementation class
- Secondary concerns
  - Adding defaults to existing methods
  - Changing defaults on existing extension methods
    - Removals of most kinds are unlikely to be compatible

Lambda Implementation
(Looking under the hood)



Lambda Implementation Approaches

- A lambda statement by definition can always be replaced by a single abstract method type
- Therefore there are several possible approaches to implementation
  - Desugar to an anonymous inner class
  - Use method handles
  - Use dynamic proxies
  - Use invokedynamic
  - Others

Anonymous Inner Class

Direct Compiler Translation

```
s -> s.getGradYear() == 2011

class StudentReduce$1 implements Predicate<Student> {
  public boolean apply(Student s) {
    return (s.getGradYear() == 2011);
  }
}
```

- Capture == invoke constructor
- One class per lambda expression (not nice)
- Burdens lambdas with identity
- No improvement in performance over current idiom

Translate Directly To Method Handles

- Compiler converts lambda body to a static method
  - Variable capture adds parameters to method signature
  - Capture == take method reference and curry the captured arguments
  - Invocation == MethodHandle.invoke

Translation Options

Issues

- Whatever translation is used is not just an implementation, but becomes a binary specification
  - Backwards binary compatability is important
  - Is the MethodHandle API ready to become a permanent binary specification
  - Performance of raw method handles compared to anonymous inner classes

More Translation Options

- Start with inner classes, switch to method handles later
  - Older compiled classes would still have inner classes
  - Java has never had “recomile your code for better performance”
  - We don’t want to start now
- We need a fixed solution
  - Old technology is bad
  - New technology is not mature enough
  - What to do?

Invokedynamic

Not just for dynamically typed languages

- Delay the translation strategy to runtime
- Invokedynamic embeds a recipe for constructing a lambda at the capture site
  - A declerative recipe, not an imperative recipe
  - Static bootstrap code: lambda meta-factory
  - At first capture a strategy is chosen and the call site linked
  - Subsequent captures use the method handle and bypass the slow path
  - Added bonus: stateless lambdas translate to static loads
    - Meta-factory returns reference to single instance

Lambda Performance Costs

- Linkage cost
- Capture cost
- Invocation cost

- The key cost to optimise is the *invocation* cost

Code Generation Strategy

- All lambda definitions are converted to static methods
  - For non-capturing lambdas the lambda signature matches the SAM signature exactly

```
s -> s.getGradYear() == 2011
```

  - Translated to Predicate<Student> becomes:

```
static boolean lambda$1(Student s) {
  return s.getGradYear() == 2011;
}
```

Code Generation Strategy

- For lambdas that capture variables from the enclosing context, these are prepended to the argument list
  - Only effectively final variables can be captured
  - Freely copy variables at point of capture

```
s -> s.getGradYear() == tagetYear
```

  - When translated to Predicate<String>:

```
static boolean lambda$1(int targetYear, Student s) {
  return s.getGradYear() == targetYear;
}
```

Code Generation Strategy

- At point of lambda capture compiler emits an invokedynamic call to create SAM (lambda factory)
  - Call arguments are captured variables (if any)
  - Bootstrap is method in language runtime (meta-factory)
  - Static arguments identify properties of the lambda and SAM

```
list.filter(s -> s.getGradYear() == tagetYear)
```

  - Becomes

```
list.filter(indy[bootstrapmethod=metafactory, args=…](targetYear));
```

Java™ ORACLE®

Runtime Translation Strategies

- Generate inner class dynamically
  - Same class that would be created by the compiler, bu generated at runtime
  - Probable initial strategy before optimisation
- generate per-SAM wrapper class
  - One per SAM type, not one per lambda expression
  - Use method handles for invocation
  - Use ClassValue to cache wrapper for SAM
- Use dynamic proxies
- Use MethodHandleProxies.asInterfaceInstance
- Use a VM private API to build object from scratch

Conclusions

- Java needs lambda statements for multiple reasons
  - Significant improvements in existing libraries are required
  - Replacing all the libraries is a non-starter
  - Compatibly evolving interface-based APIs has historically been a problem
- Require a mechanism for interface evolution
  - Solution: virtual extension methods
  - Which is both a language and a VM feature
  - And which is pretty useful for other things too
- Java SE 8 evolves the language, libraries, and VM together


MAKE THE
FUTURE
JAVA
Java
ORACLE