{Code PaLOUsa}

IT'S SOFTWARE DEVELOPMENT MADNESS

Java.Next:
Java 8 Overview

Scott Seighman
Oracle
scott.seighman@oracle.com

Disclaimer

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.

Agenda

- Quick Java 7 Update
- Modernizing Java
  - Adapting & Changing
- Key Java 8 New Features
- Timeline/Roadmap
- Resources
- Q & A

Java 7 Update

- Java 7 Update 51
  - Contains several enhancements and changes, as well as important security fixes
  - Security focus
    - Includes two security changes designed to enhance authentication and authorization for RIA (Applets and Web Start)
  - JavaFX is now part of JDK
    - Release includes JavaFX version 2.2.51
- Download
  - http://goo.gl/TfUlf
- Release Notes
  - http://goo.gl/gf8WXq

Java SE 8 (JSR 337)

Component JSRs

- New functionality
  - **JSR 308: Annotations on types**
  - **JSR 310: Date and Time API**
  - **JSR 335: Lambda expressions**
- Updated functionality
  - JSR 114: JDBC Rowsets
  - JSR 160: JMX Remote API
  - JSR 199: Java Compiler API
  - JSR 173: Streaming API for XML
  - JSR 206: Java API for XML Processing
  - JSR 221: JDBC 4.0
  - JSR 269: Pluggable Annotation-Processing API

JDK Enhancement Proposals (JEPs)

- Regularly updated list of proposals
  - Serve as the long-term roadmap for JDK release projects
  - Roadmap extends for at least three years
- Uniform format and a central archive for enhancement proposals
  - Interested parties can find, read, comment, and contribute
- Process is open to every OpenJDK Committer
- Enhancement is a non-trivial change to the JDK code base
  - Two or more weeks of engineering effort
  - significant change to JDK or development processes and infrastructure
  - High demand from developers or customers

Modernizing JAVA

Language

Libraries

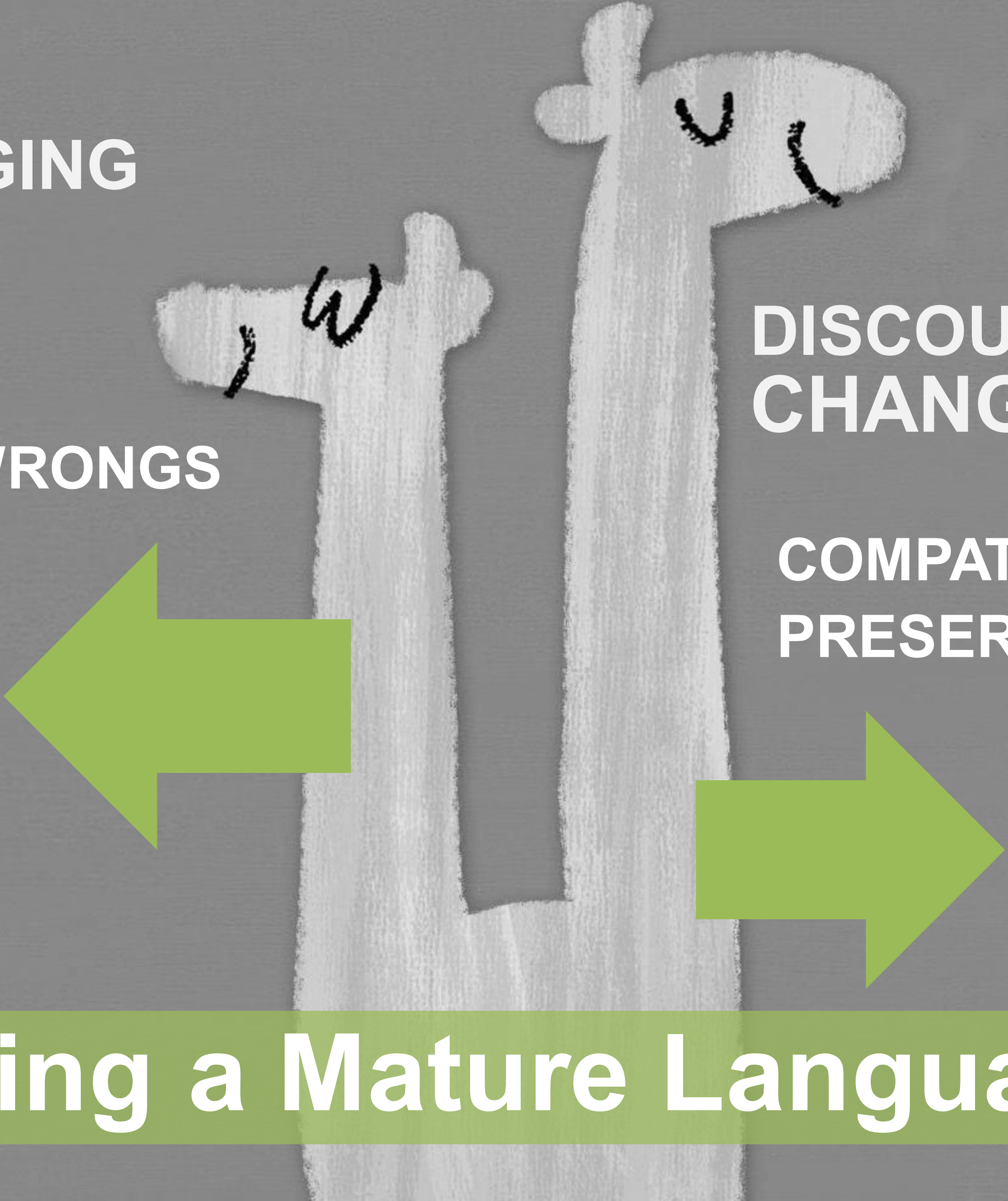
Programming Model

Evolving a Mature Language

ENCOURAGING CHANGE

ADAPTING

RIGHTING WRONGS



DISCOURAGING CHANGE

COMPATIBILITY

PRESERVE

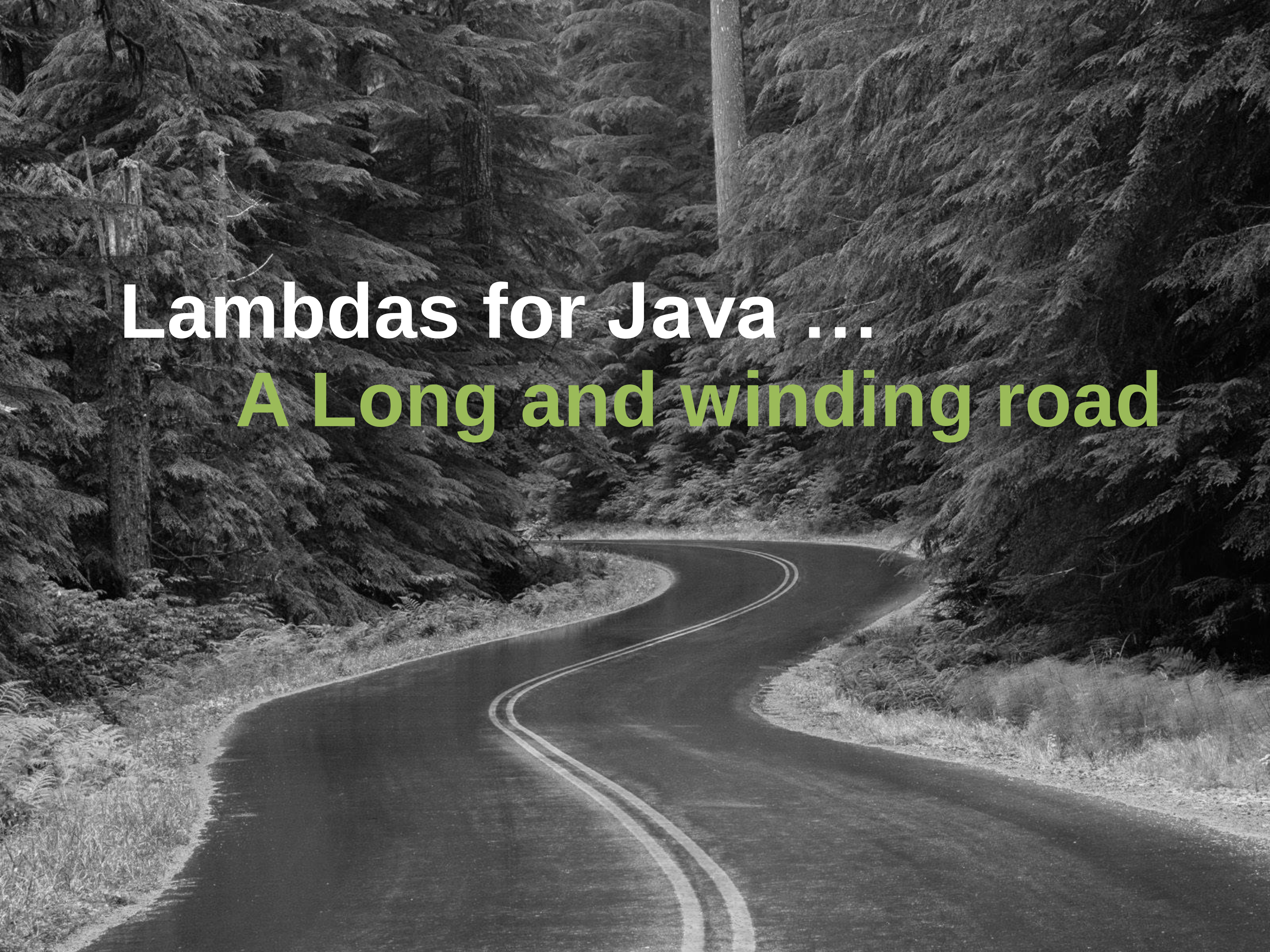
Lambdas
for Java

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

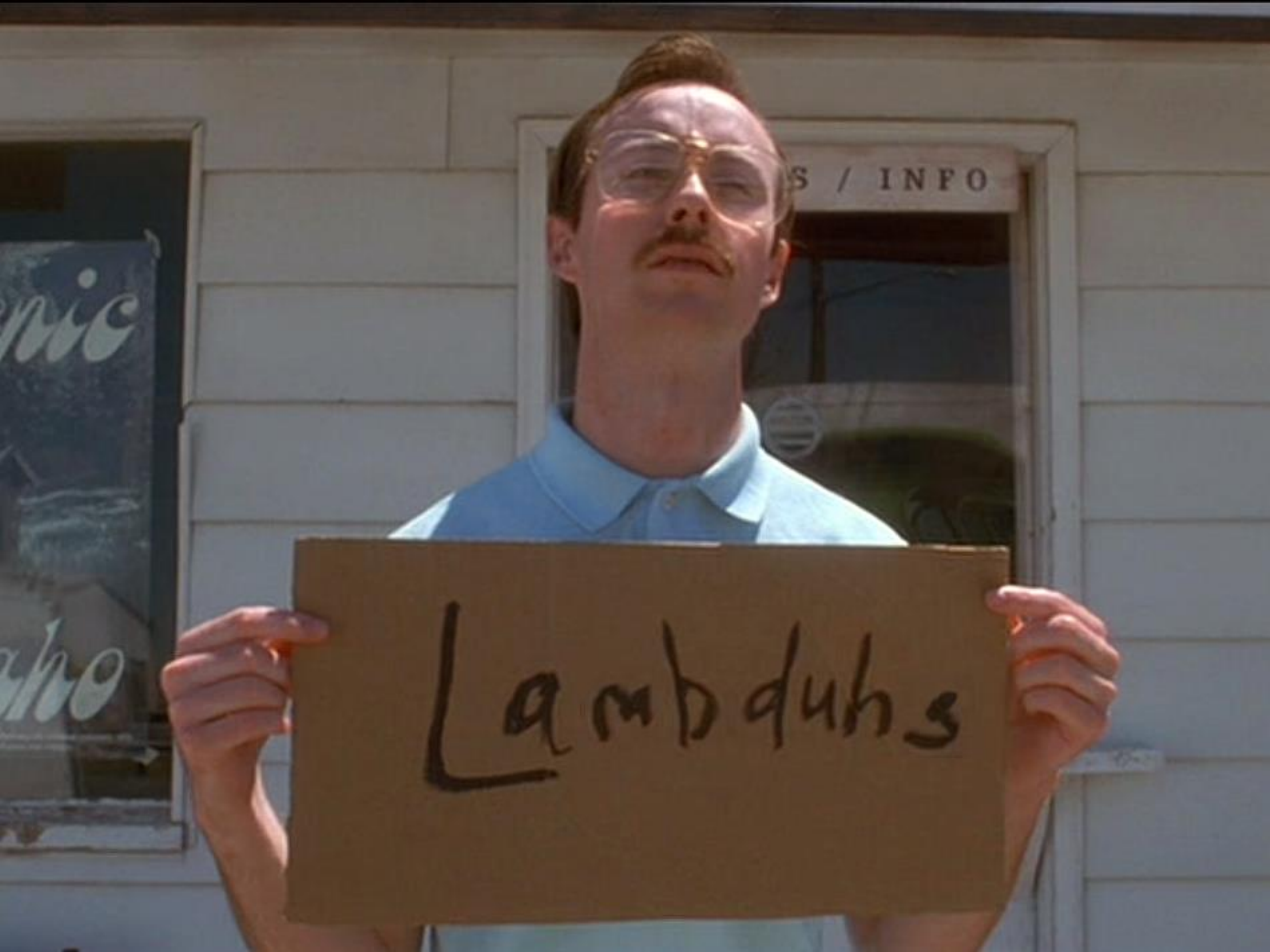
Adapting to Change

- In 1995, pervasive sequentiality infected programming language design
  - For-loops are sequential and impose a specific order
- Why wouldn’t they be? Why invite nondeterminism?
- Determinism is convenient – when free
- This sequentiality assumption propagated into libraries (e.g., Iterator)
  - Pervasive mutability
- Mutability is convenient – when free
- Object creation was expensive and mutation cheap
- In today’s multicore world, these are just the wrong defaults!
  - Can’t just outlaw for loops and mutability
  - Instead, gently *encourage* something better
- Lambda expressions are that gentle push



Lambdas for Java …

A Long and winding road


S / INFO
Lambduhs

Problem: External Iteration

- Snippet below takes the red blocks and colors them blue
- Uses `for` 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);
}
```

Imperative to Declarative

We could define the difference as follows:

- **Imperative programming**:
  - Telling the "machine" ***how*** to do something, and as a result *what* you want to happen will happen.
- **Declarative programming**:
  - Telling the "machine" ***what*** you would like to happen, and let the computer figure out *how* to do it.
- Analogy …

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*
- 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

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

What’s a Lambda Expression?

- Lambda expression is an anonymous function
- Think of it like a method
  - But not associated with a class
- Can be used wherever you would use an anonymous inner class
  - Single abstract method type
- Syntax
  - `([optional-parameters]) -> body`
- Types can be inferred (parameters and return type)

What's a Lambda Expression?

- 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)
```

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

```
Object::toString
```

- Allows you to ***treat code as data***
  - Behavior can be stored in variables and passed to methods

Lambda Typical Use Cases

- Anonymous classes (GUI listeners)
- Runnables / Callables
- Comparator
- Apply operation to a collection via `forEach` method

Effectively Final

- For both lambda bodies and inner classes, local variables in the enclosing context can only be referenced if they are final or *effectively final*
- A variable is *effectively final* if it is never assigned to after its initialization
- No longer need to litter code with **`final`** keyword

Lambdas

Demo

Convert Anonymous Class

// Anonymous inner class for event handling

```
button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.out.println(“Anonymous class");
            }
        });
```

Convert Anonymous Class to Lambda

```
button.addActionListener((ActionEvent evt) ->
            System.out.println("Lambda Example 1"));
```

- The lambda type is **inferred** by the compiler as **`EventHandler<ActionEvent>`** because the **`onAction()`** method takes an object of type **`EventHandler<ActionEvent>`**

Convert Anonymous Class to Lambda

```
button.addActionListener((evt) -> {
                System.out.println("Lambda Example 2");
          });
```

- The parameter in this lambda expression must be an **`ActionEvent`**, because that is the type specified by the **`handle()`** method of the EventHandler interface.

Convert Anonymous Class to Lambda

```
button.addActionListener(evt -> {
            System.out.println("Lambda Example 3");
        });
```

- When a lambda expression has a **single parameter** and its type is **inferred**, the parentheses are not required

Convert Anonymous Class to Lambda

```
button.addActionListener(evt ->
            System.out.println("Lambda Example 4"));
```

- Because the block of code in our lambda expression contains only one statement, we can simplify it even further

Lambda Expressions

```
(int a, int b) -> a + b
(int a) -> a + 1
() -> 42
```

Lambda Statements

```
(int a, int b) -> { println(a + b); return a + b; }

(int a) -> { println(a + 1); return a + 1; }

() -> { println("Returning 42."); return 42;
```

Lambda Syntax

```
(a, b) -> a + b

(a) -> a + 1

(a, b) -> { println(a + b); return a + b; }

(a) -> { println(a + 1); return a + 1; }
```

Types may be omitted if they can be inferred from the target type

Lambda Syntax

```
a -> a + 1

a -> { println(a + 1); return a + 1; }
```

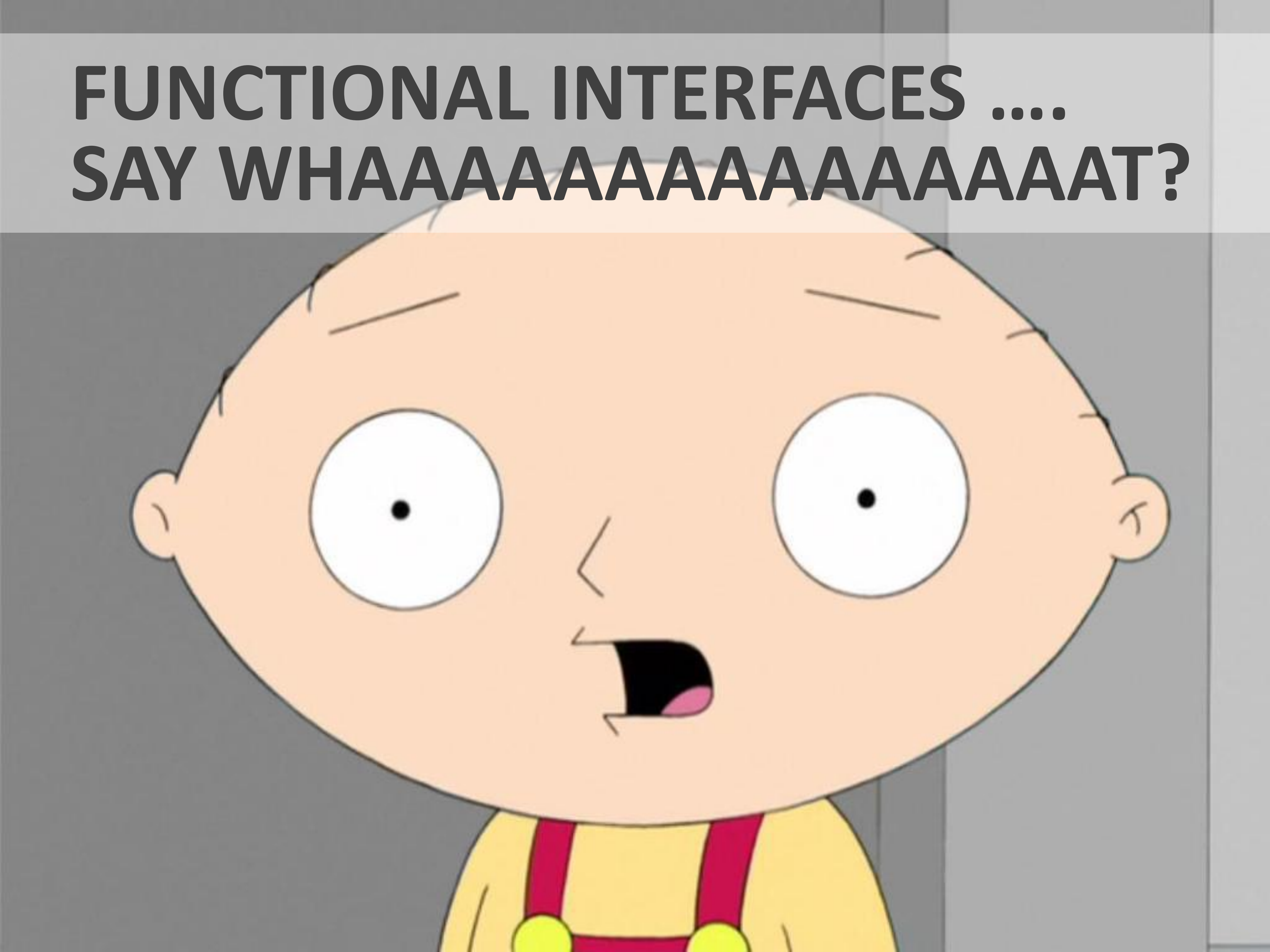
Parentheses may be omitted if the lambda accepts a single parameter

Lambda Expressions …

What’s the Type?

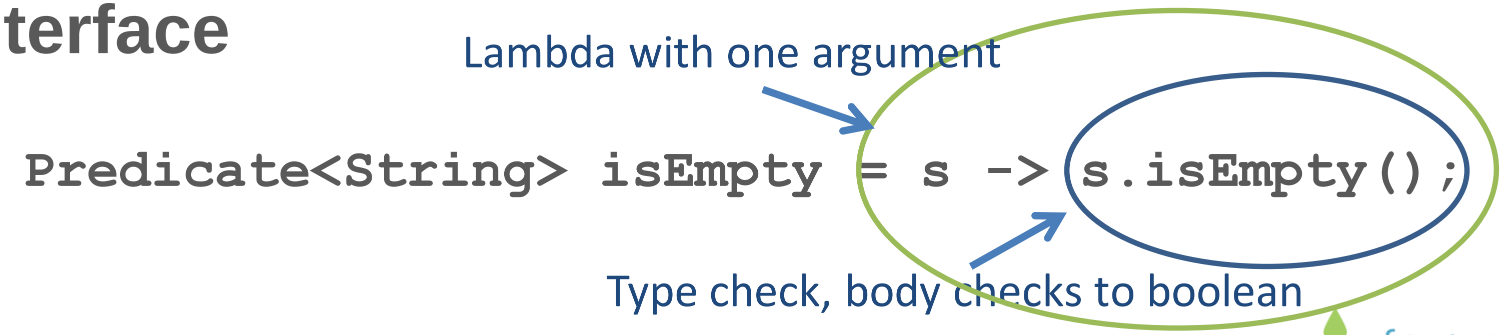
- Most languages with Lambdas have some notion of a *function type*
  - “Function from long to int”
  - Seemed reasonable (at first) to consider adding them to Java
- But…
  - JVM has no native representation of function type in VM type signatures
  - Obvious tool for representing function types is generics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

FUNCTIONAL INTERFACES …. SAY WHAAAAAAAAAAAAAAT?



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>
- The **type** of a Lambda expression is a **functional interface**



Lambda with one argument

```
Predicate<String> isEmpty = s -> s.isEmpty();
```

Type check, body checks to boolean

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

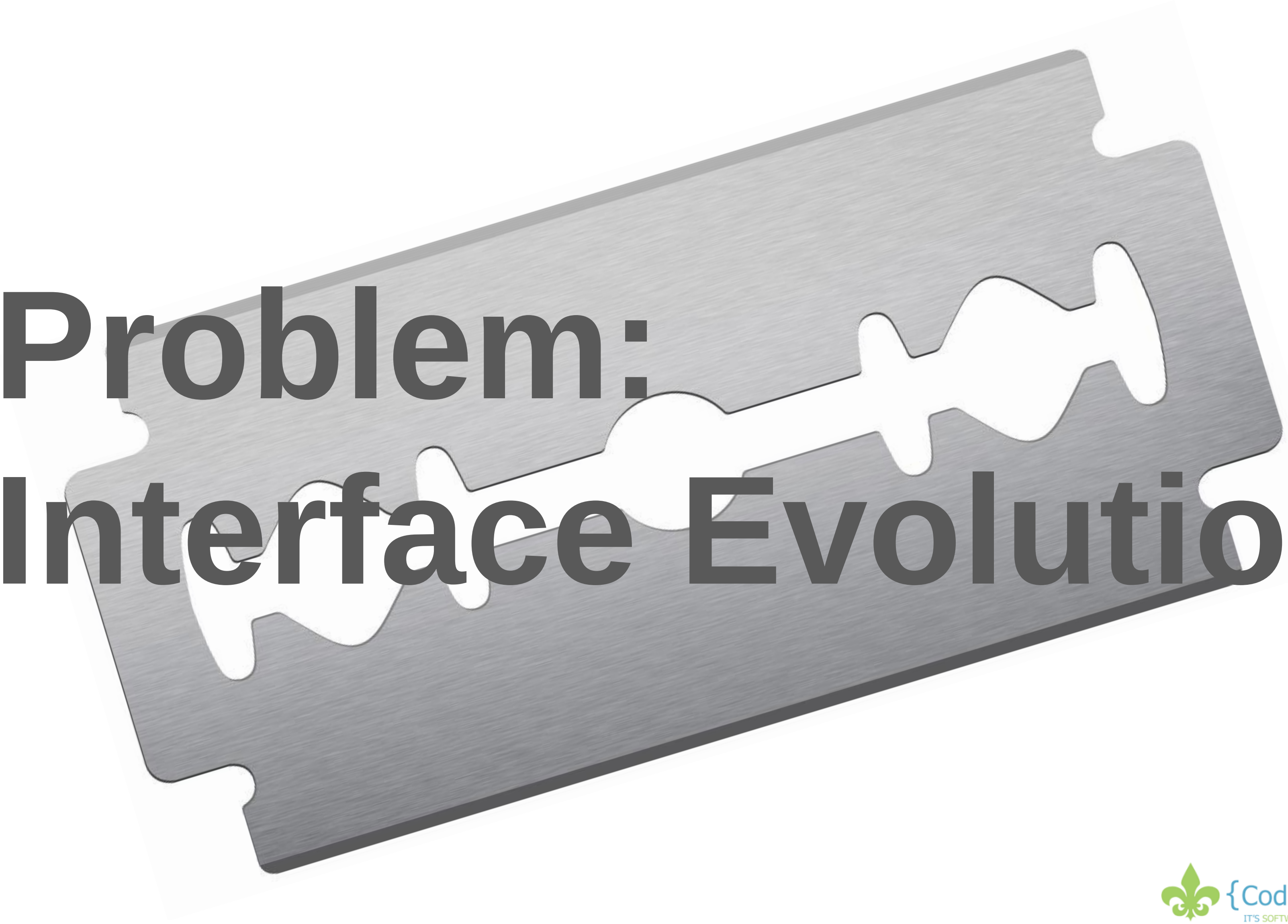
Lambdas Enable Better APIs

- 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

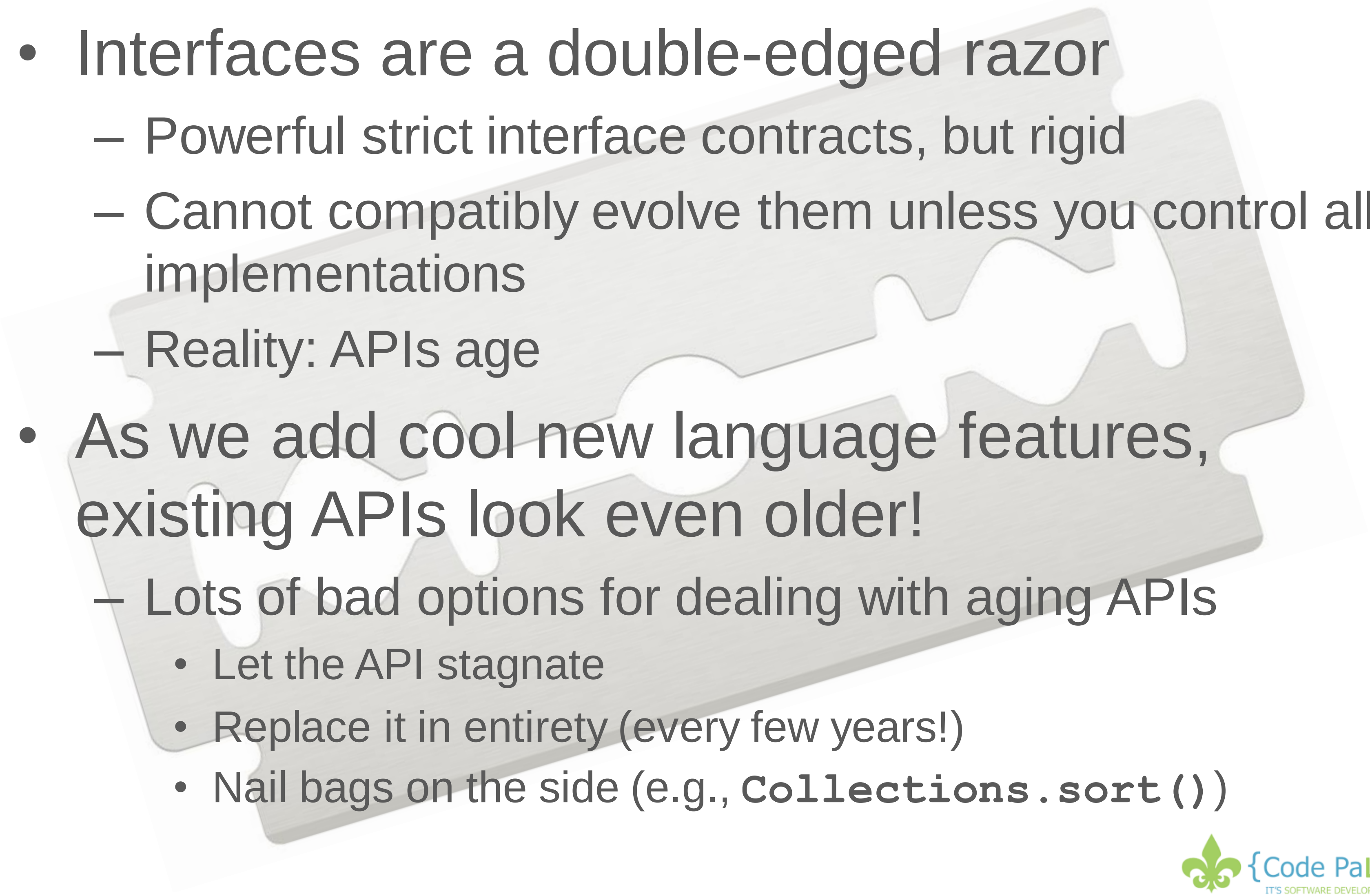
Lambdas Enable Better APIs

- 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
- But 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!**

Problem: Interface Evolution



Problem: Interface Evolution



- Interfaces are a double-edged razor
  - Powerful strict interface contracts, but rigid
  - 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()`)

Default Methods

- Libraries need to evolve, or they stagnate
  - Need a mechanism for compatibly evolving APIs
- New feature: *default methods*
  - OK to add a new method to an interface as long as you provide *some* implementation
  - Virtual interface method with default implementation
  - “default” is the opposite of “abstract”
- 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

- Wait! Isn't that multiple inheritance?
  - Adding multiple inheritance of behavior
  - Not adding multiple inheritance of state
    - Where most of trouble comes from …
- Primary goal is interface evolution
- Compare to Scala's Traits
  - Java interfaces are stateless (more like Fortress Traits)
- Compare to C# extension methods
  - Java defaults methods are virtual and declaration-site

Default Methods

- We all know that interfaces in Java contain only method declarations and no implementations and any non-abstract class implementing the interface had to provide the implementation
- To overcome this limitation a new concept is introduced in Java 8 called default methods
- Default methods are those methods which have some default implementation and helps in evolving the interfaces without breaking the existing code

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

Streams

Really efficient.

Streams

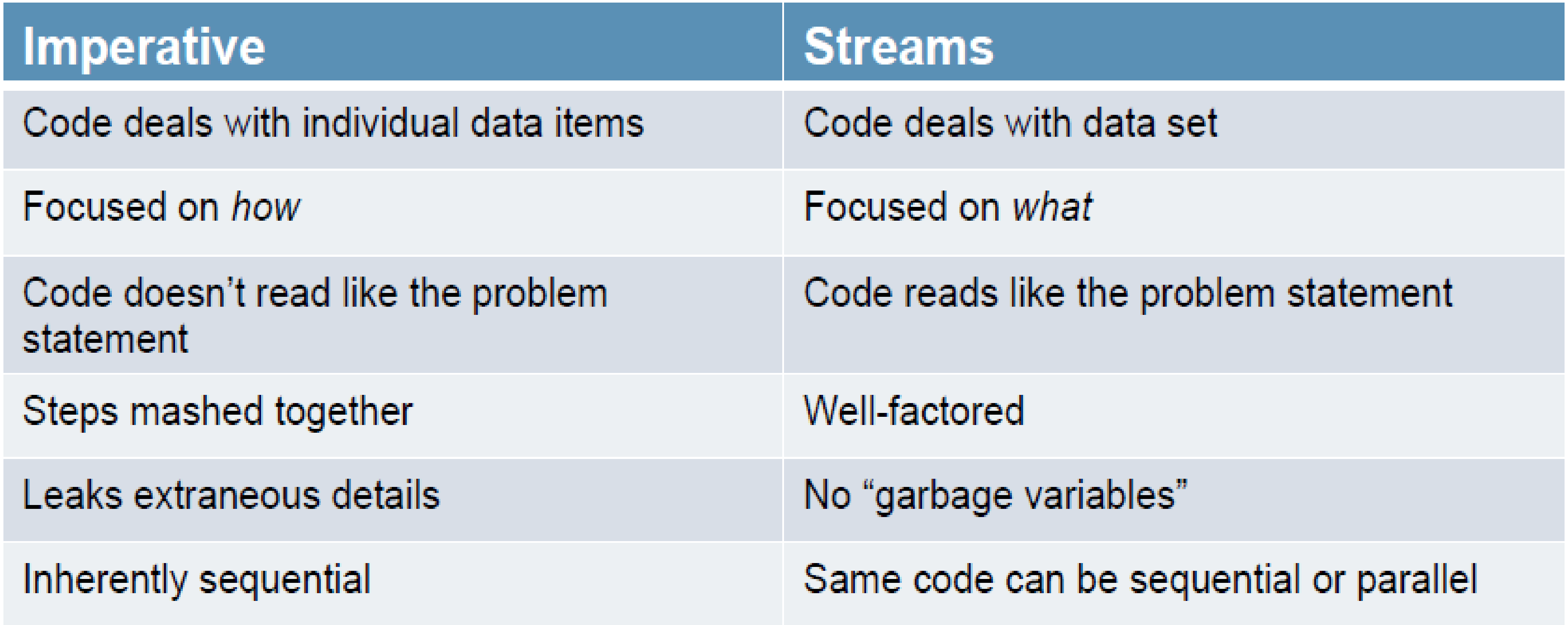
- To add bulk operations, we create a new abstraction, **Stream**
  - Represents a stream of values
- Not a data structure – doesn't store the values
  - Source can be a Collection, array, generating function, I/O…
  - Operations that produce new streams are lazy
  - Encourages a "fluent" usage style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

What is a Stream?

Stages of a stream pipeline

- This stream pipeline does three things: filter, map, sum
  - Under the hood, fused into one pass on the data
  - For both sequential or parallel execution modes
- The filter() and map() methods don’t actually do the work
  - Just set up the pipeline of operations, and return a new Stream
  - Filter and map are *lazy*
- All the work happens when we get to the sum() operation

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

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

Parallel Streams

Why go parallel?

- Performance
  - Many chips and cores per machine; each not getting much faster
  - GPUs (OpenJDK project Sumatra)
- Fork/Join is great, but too low-level
  - Stream framework uses Fork/Join under the hood
- Many factors will affect performance
  - Data size/decomposition/packing, #cores, cost-per-element
- Sadly, not a magic bullet
  - Just saying .parallel() is not always faster

Parallel Streams

Why go parallel?

- Map/reduce in the small
- Parallelism is explicit but unobtrusive
  - Sequential is the default
  - **Parallelism** may be faster but introduces non-determinism
- A stream pipeline is created with an orientation of serial or parallel
  - Can be changed with the methods `parallel()` or `sequential()`
  - Serial/parallel orientation applies to the entire pipeline
  - "Last call wins"

Streams

Summary

- Streams is a powerful framework for describing business logic on disparate data source
  - More compact, readable, performant, less error-prone
- Easy path to parallelism
  - But parallelism is not always easy
- Deeply integrated with Collections
  - But not restricted to Collections

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!

Other Notable Updates

Java Time

Java Time API

- It been a challenge to work with Date, Time and Time Zones in Java
- There was no standard approach or API in Java for date and time
- Nice addition in Java 8 is the java.time package that will streamline the process of working with time
- Sub-packages like java.time.format provide classes to print and parse dates and times
- java.time.zone provides support for time-zones and their rules
- The new Java Time API prefers enums over integer constants for months and days of the week
- One useful class is DateTimeFormatter for converting datetime objects to strings

Java Time API (cont)

- Replaces
  - `java.util.Date`
  - `java.util.Calendar`
  - `java.util.TimeZone`
  - `java.text.DateFormat`
- Immutable to work well with lambdas/functional
- No need to use Joda-Time
- Backport available in Maven Central for JDK 7

Java Time API (cont)

- // get the current date and time

  ```
  LocalDateTime now = LocalDateTime.now();
  ```

- // returns formatted date and time

  // “2013-10-21T20:25:15:16.256”

  ```
  now.toString();
  ```

- // add 5 hours

  ```
  LocalDateTime later = now.plus(5, HOURS);
  ```

- // subtract 2 days

  ```
  LocalDateTime earlier = now.minus(2, DAYS);
  ```

Java Time API (cont)

```
LocalDate today = LocalDate.now();

LocalDate bday = LocalDate.of(1979, 3, 10);

Period timeAlive = Period.between(bday, today);

int age = timeAlive.getYears();
```

// Periods can also be used as arguments in the **`.plus()`** and **`.minus()`** methods:

```
LocalDate calculatedBday = today.minus(timeAlive);
```

Compact Profiles

Compact Profiles

- **Project Jigsaw** will be included in **Java 9**
  - Modular Java platform
  - Enable developers to identify and isolate only those modules needed for their application
- Interim solution: **Compact Profiles**
- Java 8 will define subset profiles of the Java SE platform specification that developers can use to deploy.
- At the current time three compact profiles have been defined, and have been assigned the creative names *compact1*, *compact2*, and *compact3*
- Initial draft of profiles: http://openjdk.java.net/jeps/161

{Code PaLOUsa}
IT'S SOFTWARE DEVELOPMENT MADNESS

Compact Profiles (cont)

- Snapshot early access build of Java SE-Embedded 8 for ARMv5/Linux:
  - A reasonably configured Compact1 profile comes in at less than 14MB
  - Compact2 is about 18MB
  - Compact3 is in the neighborhood of 21MB
- For reference, Java 7u45 SE Embedded ARMv5/Linux environment requires ~45MB

semble
HIGHLIGHTS
NEW!
Search samples and docs here!
Searc
SAMPLES
SAMPLES
Animation
Interpolator
Timeline Events
Timeline
Fade Transition
Fill Transition
Parallel Transition
Path Transition
Pause Transition
Rotate Transition
Scale Transition
Sequential Transition
Stroke Transition
Translate Transition
Canvas

JavaFX

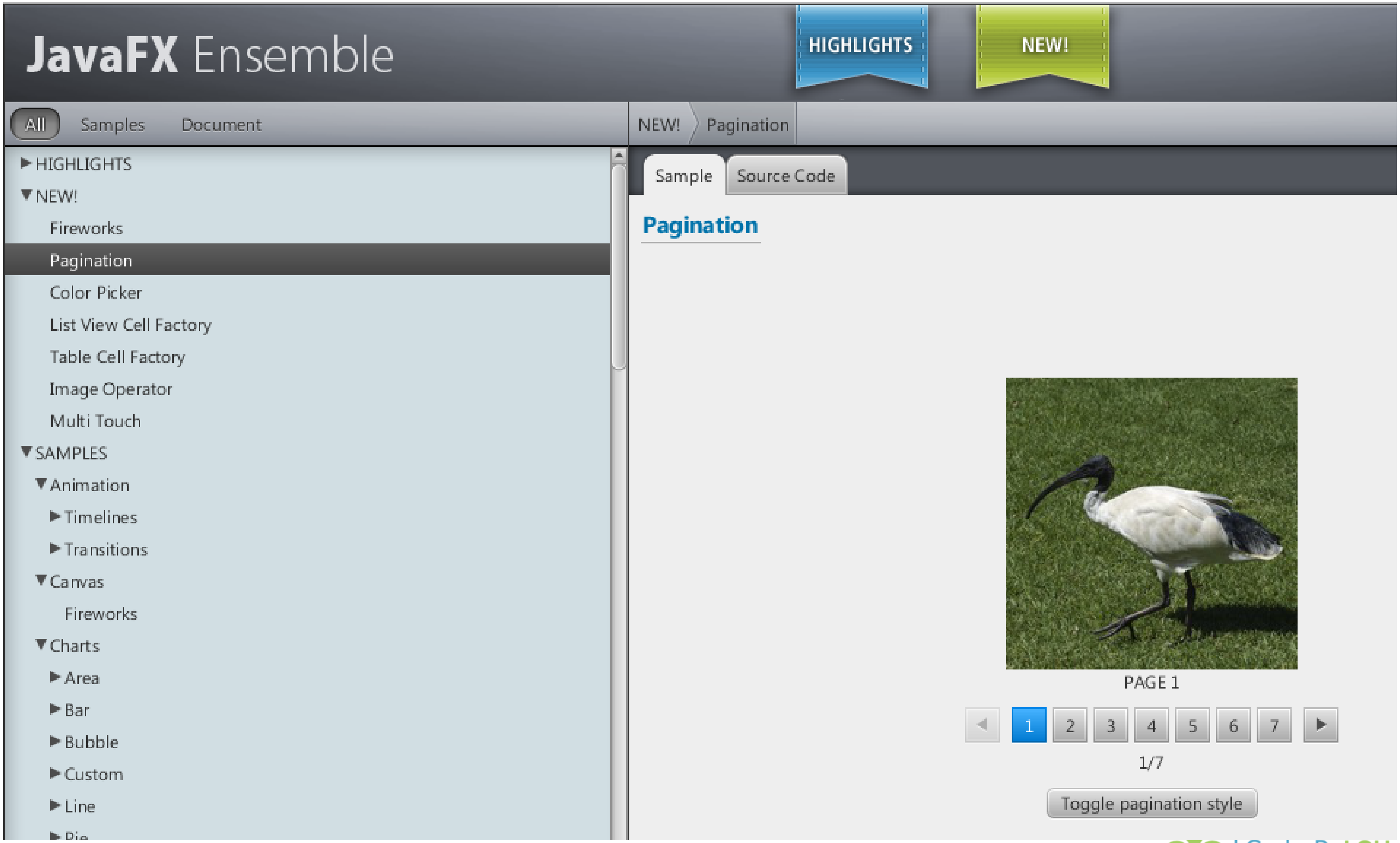
JavaFX

- Rich text support will be added via TextFlow class
- Swing node will allow you to embed swing components inside a javafx scene
- Support for video and audio recording will be added
- Printing support will be added
- New theme called Modena

JavaFX (cont)

- New DatePicker and TreeTable controls
- WebView Enhancements (Nashorn)
- Included in Oracle's Java SE Embedded 8
  - Will include a subset of features of the desktop version, namely it will not include:
    - WebView support
    - Media support
    - Workaround for media (Jasper Potts)
- Improved 3D support (more accurate to say true 3D support).

JavaFX Demos



Collections API

- We have already seen **`forEach()`** method and Stream API for collections. Some new methods added in Collection API are:
  - Iterator default method **`forEachRemaining(Consumer action)`** to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
  - Collection default method **`removeIf(Predicate filter)`** to remove all of the elements of this collection that satisfy the given predicate.
  - Collection **`spliterator()`** method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
  - Map **`replaceAll()`**, **`compute()`**, **`merge()`** methods.

Concurrency API

- Some important Concurrent API enhancements are:
  - Make CHM more useful as cache
  - Methods to support sequential and parallel bulk operations
    - **`compute()`**, **`forEach()`**, **`forEachEntry()`**, **`forEachKey()`**, **`forEachValue()`**, **`merge()`**, **`reduce()`** and **`search()`** methods.
  - Better support for maps with large numbers of elements

Concurrency API (cont)

- Fork/Join improvements:
  - Substantially better throughput when lots of clients submit lots of tasks
  - More cases are handled that allow threads to help others rather than generating compensation threads
  - Explicit support for task marking
  - Better tolerance for GC/allocation stalls

Concurrency API (cont)

- CompletionStage (Interface)
  - Stage of computation
  - Dependent functions and actions that trigger upon completion
  - Dependents may be executed sync or async
  - Computation expressed as a *Function*, *Consumer*, or *Runnable*
  - Triggered by completion of single stage, both of two stages, or either of two stage

Extensions on Collections

- Good for Lambda, great for you!
- Java 8 adds extensions to the Collection interfaces
- Added as default methods (new language feature)

```
default sort() { Collections.sort(this); }
```

- Optimized implementations for core classes

```
sort() { Arrays.sort(e, 0, size); }
```

Extensions on Collections

- So what do I get?
  - – Useful combinators on Comparator
  - – Couple of goodies for Collection, List, Set
  - – Most useful goodies on Map
    - ConcurrentMap methods
    - Useful stuff

```
map.computeIfAbsent(key, k -> new ArrayList<>())
    .add(elem);
```

Nashorn

(JavaScript Engine)

Nashorn (JavaScript Engine)

- JavaScript Engine for JVM
- 100% pure Java implementation
- 100% compiled to bytecode (no interpreter)
- 100% ECMAScript 5.1 compliant
- Ultimate **`invokedynamic`** consumer
- The only API is JSR-223: **`javax.scripting.*`**

Nashorn (Javascript Engine)

- JavaScript on Java use cases:
  - SHELL Scripting
  - Extend app functionality on runtime
  - Web-content generation
  - Provide extension points for application

Nashorn (JavaScript Engine)

- There are no browser APIs:
  - HTML5 canvas
  - HTML5 audio
  - WebWorkers
  - WebSockets
  - WebGL

Nashorn (JavaScript Engine)

- Simple HelloWorld example:

```
import javax.script.*;

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine nashorn = m.getEngineByName("nashorn");
try {
        nashorn.eval("print('Hello, world')");
     } catch (ScriptException e) {
}
```

Nashorn (JavaScript Engine)

- Simple HelloWorld example:

```
import javax.script.*;

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine nashorn = m.getEngineByName("nashorn");
try {
      nashorn.eval(new java.io.FileReader(“TestScript.js”));
     } catch (ScriptException e) {
}
```

Java I/O

- There are new IO/NIO methods, which are used to obtain `java.util.stream.Stream` from files and input streams
- Some IO improvements are:
  - **`Files.list(Path dir)`** that returns a lazily populated Stream, the elements of which are the entries in the directory.
  - **`Files.lines(Path path)`** that reads all lines from a file as a Stream.
  - **`Files.find()`** that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
  - **`BufferedReader.lines()`** that return a Stream, the elements of which are lines read from this BufferedReader.

JDeps Utility

- New utility called **`jdeps`**
- Looks through Class/JAR files and identifies which use internal APIs and then lists those APIs
- Answers the questions, “am I using these internal classes” and “if so, which ones.”
- Usage:

```
jdeps <options> <classes ..>
```

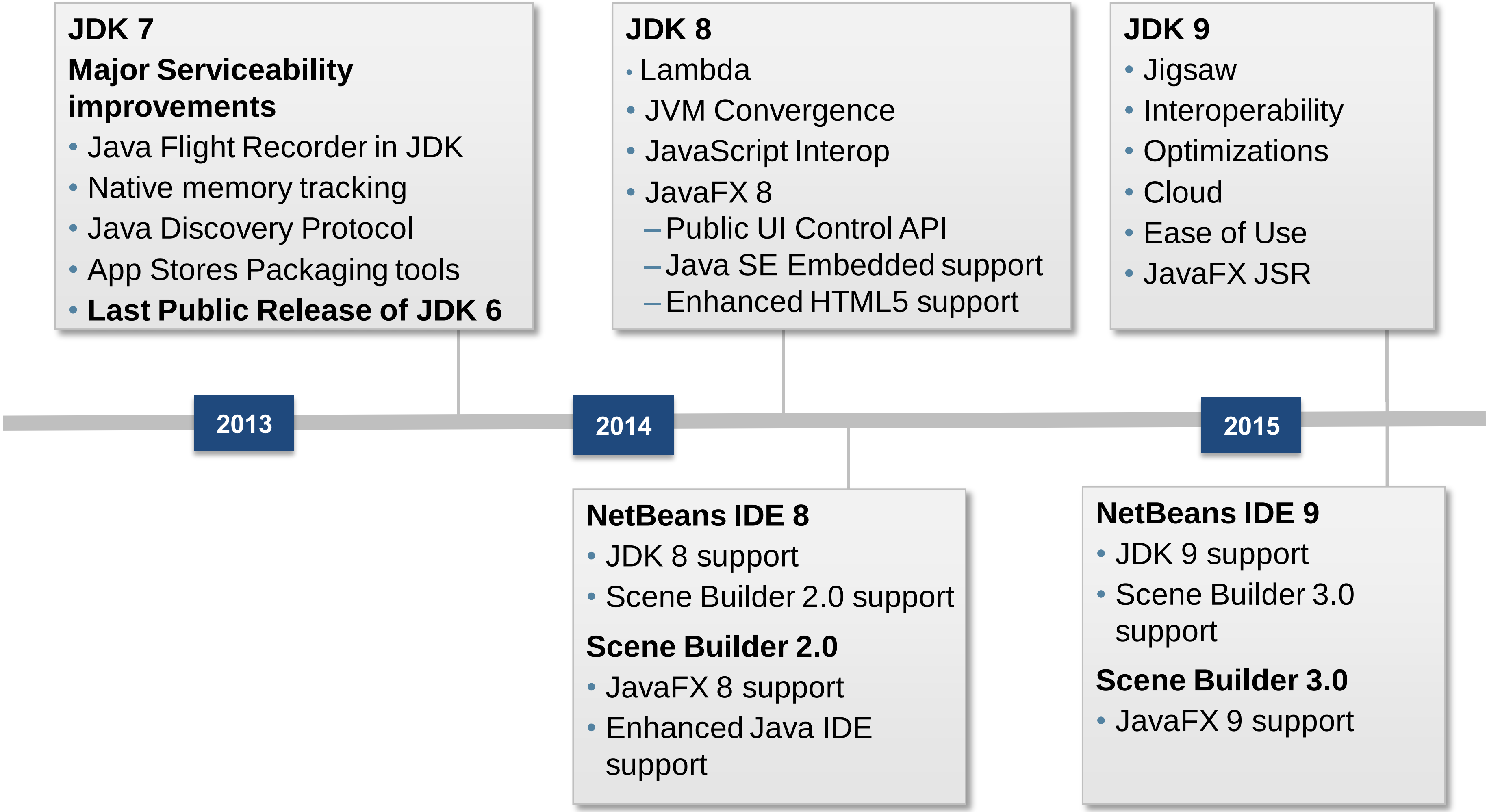
Removed: Permanent Generation

- Remove the permanent generation from the Hotspot JVM and thus the need to tune the size of the permanent generation
- Part of the JRockit and Hotspot convergence effort. JRockit customers do not need to configure the permanent generation (since JRockit does not have a permanent generation) and are accustomed to not configuring the permanent generation.
- Move part of the contents of the permanent generation in Hotspot to the Java heap and the remainder to native memory.

Current Java 8 Timeline

- Developer Release: Sept. 5, 2013
- Final Release Candidate: January 23, 2014
- General Availability: March 18, 2014

Java SE Roadmap*



2013

JDK 7

Major Serviceability improvements

- Java Flight Recorder in JDK
- Native memory tracking
- Java Discovery Protocol
- App Stores Packaging tools
- **Last Public Release of JDK 6**

2014

JDK 8

- Lambda
- JVM Convergence
- JavaScript Interop
- JavaFX 8
  - Public UI Control API
  - Java SE Embedded support
  - Enhanced HTML5 support

NetBeans IDE 8

- JDK 8 support
- Scene Builder 2.0 support

Scene Builder 2.0

- JavaFX 8 support
- Enhanced Java IDE support

2015

JDK 9

- Jigsaw
- Interoperability
- Optimizations
- Cloud
- Ease of Use
- JavaFX JSR

NetBeans IDE 9

- JDK 9 support
- Scene Builder 3.0 support

Scene Builder 3.0

- JavaFX 9 support

***Subject to change**

Resources

Links, Docs, Demos

- Download Java 8 RC2
  - https://jdk8.java.net/
- Lambda Project
  - http://openjdk.java.net/projects/lambda/
- JavaOne 2013 Content
  - http://www.oracle.com/javaone/index.html
- Java SE 8 Lambda Quick Start
  - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html?cid=7180&ssid=104133974809712#
- Lambda FAQ
  - http://lambdafaq.com/
- VirtualJUG
  - http://www.meetup.com/virtualJUG/

Conclusions/Summary

- Java SE 8 will add plenty of new features (and remove a few)
  - Language
  - Libraries
  - JVM
- Java continues to evolve!
  - jdk8.java.net
  - www.jcp.org
  - openjdk.java.net/jeps

Thanks!

Scott Seighman
Scott.seighman@oracle.com

Performance

Should I go parallel?

- Do not assume parallel is always faster
  - Going parallel is easy, but not always the right thing to do
  - Sometimes parallel is slower than sequential
- Qualitative considerations
  - How good is the stream source decomposition?
  - Does the terminal operation have a cheap or expensive merge step?
  - What are the stream characteristics?
- The primitive streams are included for performance reasons
  - Boxing hurts performance

A Simple Performance Model

Quantitative considerations

N = size of source data set

Q = cost per-element through the pipeline

N * Q ~= cost of pipeline

- Larger **N * Q** → higher chance of good parallel performance
- Easier to know **N** than **Q**
  - – But can reason qualitatively about it

Intuition and Measurement

- For small data sets, sequential usually wins
- Watch out for boxing
- Simpler pipelines are easier to "guesstimate"
  - **N > 10K**, **Q = 1**, "Do you have 10K elements or more?"
  - Reduction using **sum()**
- Complex pipelines are harder to reason about
  - Stream source is derived from an iterator
  - Pipeline contains a limit() operation
  - Complex reduction using **`groupingBy()`**
- If in doubt, measure!
  - Use a benchmark tool such as **jmh**