ORACLE®

~~The Future of Java~~

ava: Present And Future
(With A Bit Of The Past)

mon Ritter
ead of Java Technology Evangelism
racle Corporation

Vith much thanks to Brian Goetz

ORACLE

afe Harbor Statement

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

ORACLE

ava SE 8

nhancing The Core Platform

ORACLE

Java 8

HTTP URL Permissions
Base64
Enhanced Verification Errors
Improve Contended Locking
DocTree API
Prepare for Modularization
Lambda (JSR 335)
Remove the Permanent Generation
Generalized Target-Type Inference
Date/Time API (JSR 310)
Bulk Data Operations
Parallel Array Sorting
Limited doPrivileged
Repeating Annotations
Compact Profiles
Parameter Names
Nashorn
Unicode 6.2
Configurable Secure-Random Number Generation
TLS Server Name Indication
Type Annotations (JSR 308)
Lambda-Form Representation for Method Handles
Fence Intrinsics

ORACLE

Modernising Java

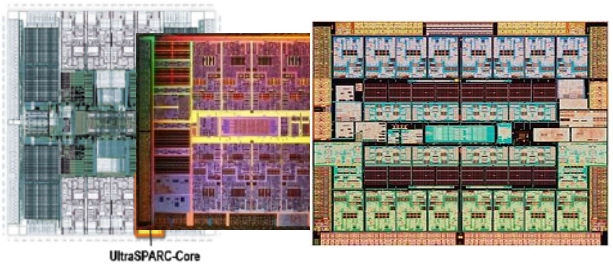
- Java SE 8 modernises the language
  - Lambda expressions (closures)
  - Interface evolution (default methods)
- Java SE 8 modernises the libraries
  - Bulk data operations on Collections
  - More library support for parallelism
  - Streams API
- Probably the biggest upgrade to the Java programming model

ORACLE

Lambda Expressions

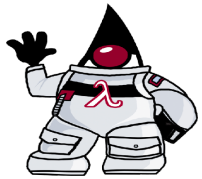
Why Do We Need Them?

- Multi-core, multi-processor machines are the norm
- We need to make writing parallel Java code easier
- To improve the libraries we need language changes

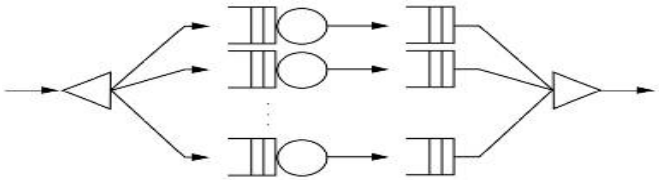


Concurrency in Java

Project Lambda



Fork/Join Framework
(jsr166y)



java.util.concurrent
(jsr166)

Phasers, etc
(jsr166)

lang.Thread

1.0 5.0 6 7 8

1995 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2

Lambda Expressions

What Are They?

- A Lambda expression (closure) is an anonymous function
  - Has an argument list, return type, body and thrown exceptions

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

  - Can refer to values in the surrounding 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
  - Simple way to store or pass behaviour as a parameter

Times Move On

Let's Go Way Back To 1995

- Most popular languages did not support closures
- Java was the last holdout that did not before Java SE 8
  - C++ added them in C++11
  - C# added them in version 3.0
  - All languages being designed today include them

ORACLE

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
- 2006-2008 – A vigorous community debate about Java closures
  - Multiple proposals, including BGGA and CICE
    - BGGA – Creating control abstraction in libraries
    - CICE – Reducing syntactic overhead of inner classes
  - No consensus, other distractions got in the way
- Little language evolution in the last ten years (Java SE 5)
  - Project Coin in Java SE 7

ORACLE

Closures For Java

A Long And Winding Road

- Dec 2009 – OpenJDK Project Lambda formed
- November 2010 – JSR-335 filed
- JSR-335 = Lambda expressions
  + Interface evolution
  + Bulk collection operations

ORACLE

Evolving A Mature Language

Key Forces

- Why change the language?
  - Adapt to external changes
    - hardware, attitudes, fashion, problems, solutions
  - Correcting errors
    - Inconsistencies, holes, poor user experience
- Why leave the language alone?
  - Maintaining compatibility
    - Nothing worse than changing working code to make it work
  - Keeping people happy
    - Different does not necessarily mean better
    - Why add a shiny new ball when you can polish the existing ball?

ORACLE

Adapting To External Changes

Moore's Law Had Different Effects In 1995

- In 1995, language design was all about sequentiality
  - For-loops are sequential and impose a specific order
    - Great feature! Why invite nondeterminism?
    - Determinism is convenient (when it's free)
    - The assumption of sequentiality propagated to libraries (e.g., Iterator)
  - Pervasive mutability
    - Mutability is convenient (when it's free)
    - Object creation was expensive, mutation was cheap
- For today's multicore world these assumptions are wrong
  - Can't outlaw for-loops and mutability
  - Need to offer a choice: Lambda expressions and Streams API

ORACLE

Problem: External Iteration

The For-Loop

- Take red blocks and colour them blue
- Uses foreach loop
  - Inherently sequential
  - Client has to manage iteration

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

- foreach loop hides complex interaction between library and client
  - `Iterable`, `iterator()`, `Iterator.next()`, `Iterator.hasNext()`

Using Internal Iteration

Using A Lambda Expression and Collection.forEach

- Code looks similar, but:
  - Not just a syntactic change
  - Now the library is in control
  - Separate the *how* from the *what*

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

- Library free to use parallelism, out-of-order execution, lazy evaluation
- Client passes behaviour (Lambda) as parameter
- Enables API designers to build more powerful, expressive APIs
  - Greater power to abstract over behaviour

ORACLE

Lambda Expressions

What Type Is It?

- Most languages with closures 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 a function type in VM type signatures
  - Obvious tool for function types would be generics
  - Is there a simpler alternative?

ORACLE

Functional Interfaces

- "Just add function types" was obvious… and wrong
  - Would have interacted badly with erasure and boxing
  - Would have introduced complexity and corner cases
  - Would have created two types of libraries: "old" style and "new" style
- Back to polishing the current ball rather than adding a shiny new one
- Bonus: existing libraries are *forward-compatibl*e with Lambdas
  - Libraries that never imagined Lambdas work with them!
  - Maintains significant investment in existing libraries
  - Fewer new concepts (easier to learn)

ORACLE

Problem: Interface Evolution

- Earlier example used a new Collection method, `forEach()`
  - This does not exist in Java SE 7
- Interfaces are a double-edged sword
  - Cannot compatibly change them unless you control all implementations
  - Lots of bad options for dealing with aging APIs
    - Let the API stagnate
    - Replace it entirely (every few years, e.g. nio)
    - Nail bits on the side (e.g. Collections.sort())

ORACLE

Default Methods

Mechanism For Compatibly Evolving APIs

- Virtual interface method with default implementation
- 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

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

ORACLE

Default Methods

Wait a minute!

- Isn't this adding multiple inheritance to Java
  - Java has always had multiple inheritance of *types*
  - This adds multiple inheritance of *behaviour*
  - But not of *state*, which is where most of the problems come from
- Primary goal is interface evolution
- Compared to C# extension methods
  - Java default methods are *virtual* and *declaration-site*, not *static* and *use-site*
- Compared to Scala traits
  - Java interfaces are stateless (more like Fortress traits)

ORACLE

Bulk Operations On Collections

Streams API

- Let's revisit the shape code
  - Further decomposition
  - Use `filter()` and `forEach()`

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

```
shapes.stream().
  filter(s -> s.getColour() == RED).
  forEach(s -> s.setColour(BLUE));
```

Streams API

Benefits

- 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 the client code is less brittle
  - Library can use parallelism, out-of-order execution and lazy evaluation to impro
    performance

ORACLE

Stream Example 1

Convert words in list to upper case

```
List<String> output = wordList.
  stream().
  map(String::toUpperCase).
  collect(Collectors.toList());
```

Stream Example 1

Convert words in list to upper case (in parallel)

```
List<String> output = wordList.
  parallelStream().
  map(String::toUpperCase).
  collect(Collectors.toList());
```

Stream Example 2

Count lines in a file

```
ong n = Files.newBufferedReader("foo.txt").lines().count()
```

Stream Example 4

List of words in lowercase, in alphabetical order

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(toList());
```

Stream Example 4

List of unique words in lowercase, sorted by length

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  map(String::toLowerCase).
  distinct().
  sorted((x, y) -> x.length() - y.length()).
  collect(toList());
```

To Java SE 9 and Beyond!

Project Sumatra – Java for GPUs

Improved Integration with Nat

ang Enhancements

Cloud

OpenJFX

esource Management

Self Tuning JVM

Optimizations

Generic Lang Interoperab

Multi-Tenancy Support

Ease of use

Jigsaw

Unified Type Syster

rts: Power PC/AIX

Data Structure Optimizations

Penrose

Hardware and Software
Engineered to Work Together

ORACLE