Java: Past, Present, and Future

Brian Goetz
Java Language Architect
Oracle Corporation

PAST

Background to Java

- The programming landscape in 1992-1995…
  - Fortran was dominant for scientific computing
  - C dominated just about everything else
  - Pascal on the wane
  - Perl used for small stuff
  - Niche enclaves of SmallTalk and Lisp
- Basically, all C, all the time

Background to Java

- But C sucked!
  - …at least for applications
  - Too low-level
  - Pervasive memory management tax
  - Portable, but not portable
  - Too hard to reuse code
- Most people wanted something better
  - But we were already struggling against “Yeah, but C runs everywhere”

Background to Java

- The hype landscape in 1992-1995…
  - Object-orientation
    - C++ not quite yet real
    - Design Patterns rising in the hype-o-sphere
  - Distributed Objects
    - CORBA (Object Management Group)
    - DOE (Sun)
    - DCE (Open Group)
  - The Web Will Change Everything
    - OK, this one really happened

Java, as self-described in 1995

- "A blue-collar language"
  - Opposite of "ivory tower"
- "A simple, object oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high performance, multithreaded, dynamic language"
  - OK, some hype here too

Static or Dynamic?

```
boolean isDynamic;

float howDynamic;

Map<Axis, Float> howDynamic;
enum Axis {
    TYPING,
    COMPILATION,
    DISPATCH,
    INTROSPECTION,
    LINKAGE,
    LOADING_AND_UNLOADING,
    ...
};
```

Static … and Dynamic

- Statically typed … and dynamically too
- Statically compiled … and dynamically too
- Static method overload resolution
  - But dynamically linked
  - …and verified
  - …with dynamic enforcement of accessibility
- Dynamic dispatch on receiver
  - Static dispatch on arguments
- Dynamic introspection (reflection)
- Dynamic loading and unloading

Radical … And Conservative

- An odd-seeming combination of risky and conservative
- Many features not yet proven in industrial languages
  - Garbage collection, Bytecode, JIT
    - Concepts well understood but performance not yet there
  - Dynamic linkage, loading, introspection
  - Integrated thread support
  - Cross-platform memory model!
  - Hybrid inheritance model
  - Unicode
  - Serialization (hey, can't get 'em all right)

Radical … And Conservative

- Fanatical about safety
  - No programmer access to pointers (!)
  - Runtime checks for nulls, array bounds
  - Runtime verification
- On the other hand, eschewed lots of features because of “complexity”
  - Operator overloading, macros, typedef, struct, union, enum, function pointers, multiple inheritance, automatic coercions

Method or Madness?

- We've got a grab bag of risky stuff from academic languages, and YAGNI-ism
  - At the same time, claiming "blue-collar language"
  - Was this just a random walk through the design space?
- Notice…
  - The risky stuff is mostly in the VM
  - The YAGNI stuff is mostly in the language

A Wolf in Sheep's Clothing

"It was clear from talking to customers that they all needed GC, JIT, dynamic linkage, threading, etc, but these things always came wrapped in languages that scared them."

– James Gosling (private communication)

Step 3 ... Profits!

- A few initial targeting mis-steps
  - Initially aimed at set-top boxes (oops)
  - Then, rotated to the client (oops)
  - Then rotated to the server – success!
- Credit the wolf
  - Fast GC and JIT
  - Reliable concurrency
  - Safety
  - Dynamic linkage, loading, introspection

PRESENT

Java Today

- The worlds most popular *programming language*
- The worlds most popular *deployment platform*
  - On 97% of enterprise desktops
  - On 3+ billion devices
- A community of over 9M developers

If Java Were A Country

	Java
Population	9M
GDP	USD 200B (?)

Design Tensions

- Reasons to change
  - Adapting to change
    - Changing hardware, attitudes, fashions, problems, demographics
  - Righting what's wrong
    - Inconsistencies, holes, poor user experience
- Reasons not to change
  - Maintaining compatibility
    - Low tolerance for change that will break anything
  - Preserving the core
    - Can't alienate user base in quest for "something better"
    - Easy to focus on cool new stuff, but there's lots of cool old stuff too

Core Language Principles

- Reading code is more important than writing code
- Simplicity matters
- One language, the same everywhere

Background to Java 8

- Developers have more choices in programming languages
  - Even without leaving the JVM
- Need to keep up with rising developer productivity expectations
- Need to offer a simpler programming model for data parallelism
  - While still being Java

Java 8

- Java SE 8 adds a small number of new features
  - Lambda expressions (and method references)
    - With a healthy dose of type inference
  - Default (and static) methods in interfaces
  - java.util.stream package for aggregate / data-parallel operations
- …that will have a big impact on developer productivity

Lambda Expressions

- A *lambda expression* is an anonymous method
  - Enables *behavior* to be manipulated as *data*
- Lambda expressions in the language…
  - …enable more powerful libraries
  - …enabling more expressive, more readable, less error-prone use code
  - …boosting developer productivity
- Also, key to an accessible parallelism strategy

Lambda Expressions

```
interface Predicate<T> {
    boolean test(T t);
}

class Collections {
    public static<T> void removeAll(Collection<T> c,
                                    Predicate<T> p) { … }

Collections.removeAll(people,
                      new Predicate<Person>() {
                          public boolean test(Person p) {
                              return p.getAge() > 18;
                          }
                      });
```

Lambda Expressions

```
interface Predicate<T> {
    boolean test(T t);
}

class Collections {
    public static<T> void removeAll(Collection<T> c,
                                    Predicate<T> p) { … }

Collections.removeAll(people, p -> p.getAge() > 18);
```

Aggregate Operations

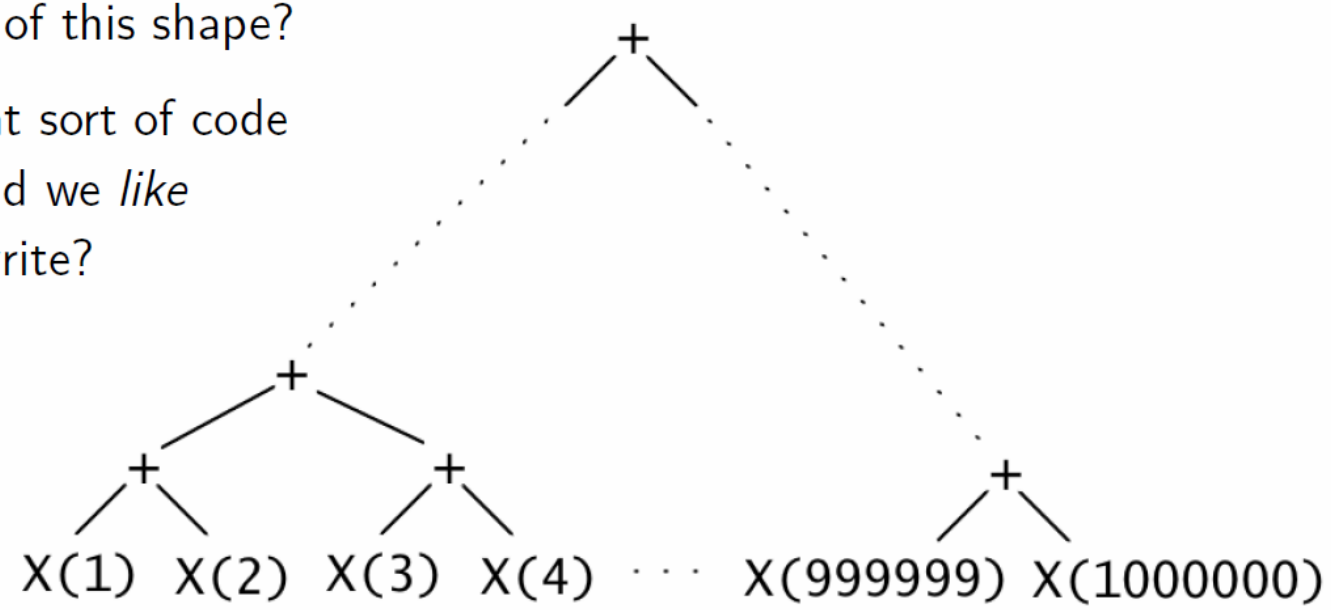
```
List<Person> people = ...

int heaviestGuy =
    people.parallelStream()
          .filter(p -> p.getGender() == MALE)
          .mapToInt(p -> p.getWeight())
          .max();
```

Brief Inspiration Break

Parallel Computation Tree

What sort of code should we write to get a computation tree of this shape?

What sort of code would we *like* to write?



ORACLE

Aggregate Operations

```
List<Person> people = ...

int heaviestGuy =
    people.parallelStream()
          .filter(p -> p.getGender() == MALE)
          .mapToInt(Person::getWeight)
          .max();
```

Retrofitted Collections for self-decomposition via Spliterator abstraction

Stream operations generic relative to Spliterator

Entire pipeline fused into single parallel pass

Source characteristics (sized-ness, distinct-ness, sorted-ness) used to optimize execution

Aggregate Operations

```
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());
```

- Operates on data set, not individual elements
- What, not how
- Code reads like the problem statement
- Doesn't leak extraneous details
- Well factored
- Free parallelism!

- Or…

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

Another Wolf In Sheep's Clothing

- Lambda expressions look like "just another language feature"
  - But really, Java has taken a gentle turn for the functional
  - A gentle push away from mutative / imperative
  - While still being Java
- Will have a huge impact on API design
  - Which in turn will improve how people program

Default Methods

- Java is a victim of its own success
  - Our Collections APIs are 15+ years old!
- Can't compatibly evolve interface-based APIs
  - Perverse result: adding lambdas to the language makes Collections look even older!
  - But replacing Collections would take years
- What we need is: interface evolution

Default Methods

- Add a new method to an interface
  - With a default implementation
  - Fully virtual, leaves API owners in control of their APIs
  - This is how we added the stream() method

```
interface Collection<T> extends Iterable<T> {
    ...
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }
}
```

Java SE 8

- A few new features
  - Lambdas (and method refs)
  - Default (and static) methods in interfaces
  - Possibly-parallel stream operations on Collections (and arrays, and anything else you want)
- Which will change the way 9M people program

FUTURE

Are We Done?

- Java SE 8 is a big step forward, is there more?
  - Still lots of pain points
    - Boxing
    - Tuples, records, multiple-return
    - Erasure
    - Where's List<int> ?
    - Pointer chasing
    - Array problems
      - 32 bit limitation, mutability, sparseness, heterogeneity
      - No deterministic inlining
  - And some new execution targets
    - GPUs, FPGAs

What's Next?

- Underlying most of these is … object identity
- Java's type system gives us:
  - A fixed set of primitive value types
  - Arrays – homogeneous aggregation
  - Classes – heterogeneous aggregation
- Nice things about primitive types
  - No identity
  - No object header
  - No indirection
  - Can store in registers
  - Can push on stack
- … But we can't make new ones

Value Types

- Our next big target is *value types*
  - Like classes, but without identity
  - No header, store fields in registers/stack
- Key enabler for
  - Tuples, records
  - User-defined primitive types
    - Can be packed efficiently into arrays, inlined into objects
  - Generification over primitives
- VM first, then language

Summary

- Lots of people thought Java was “done for” as recently as a few years ago
- Demonstrated that significant modernization is possible without compromising compatibility or principles!
- And we’re going to keep on doing that

Thank You

Brian Goetz
Java Language Architect
Oracle Corporation