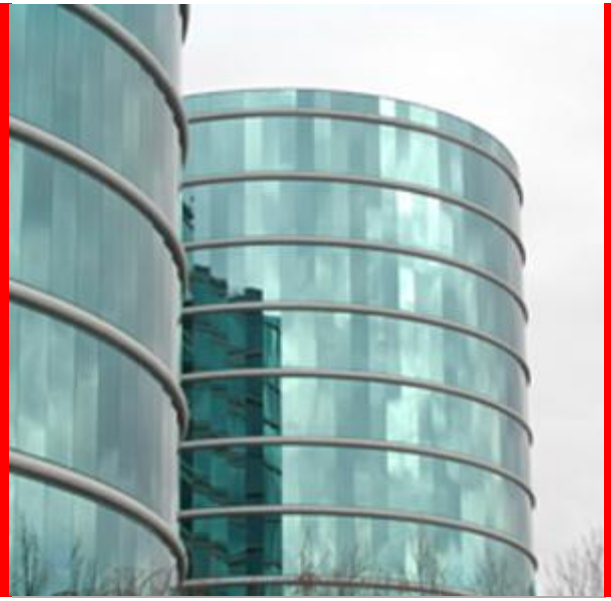


ORACLE

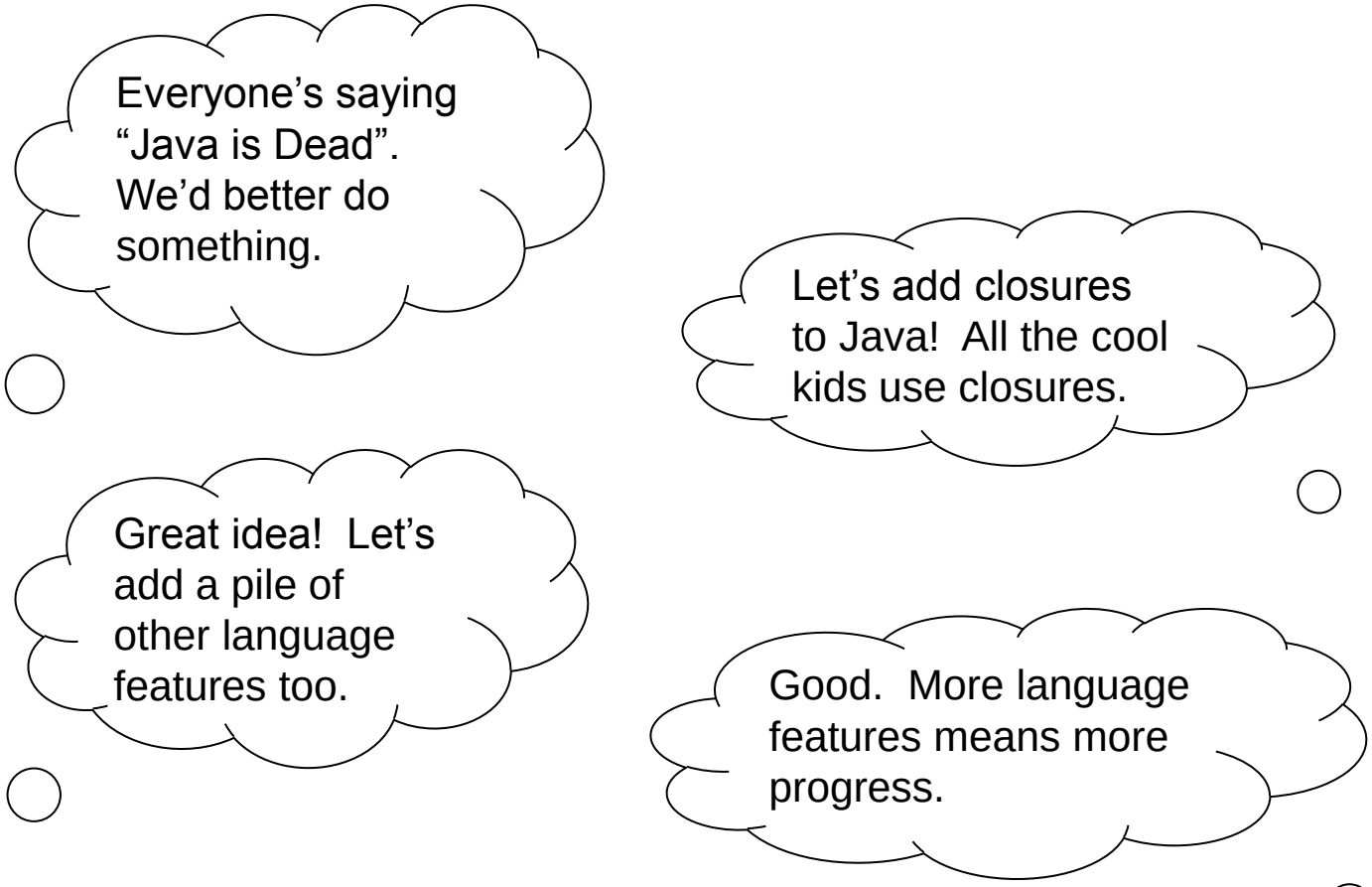
Virtual Extension Methods for Java

Alex Buckley, Java Language and VM Specification Lead, Oracle
Brian Goetz, Java Language Architect, Oracle

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.

BACKGROUND

(Not) heard in a hallway in Santa Clara



Why closures for Java?

- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point to not have closures
- Provide libraries a path to multicore
  - Today, a developer's primary tool for computing over aggregates is the for loop – which is *fundamentally serial*
  - Internal iteration needed to make data structures parallel-friendly
- Empower library developers
  - Easier to evolve the programming model through libraries than through language
- Hidden motivation: give developers a gentle push away from a side-effect-centric programming model

New language features for Java SE 8
Pulling on the string called closures

- Lambda expressions (closures)

```
(String x) -> x.length() == 0
```

- Functional interface conversion

```
Predicate<String> p = (String x) -> x.length() == 0;
```

- Type inference of lambda formal parameters

```
Predicate<String> p = x -> x.length() == 0;
```

- Method references

```
Predicate<String> p = String::isEmpty;
```

- Virtual extension methods

The real problem: interface evolution

- Adding closures is a big language change
  - If Java had closures from day 1, APIs would look different
  - Adding closures now makes aging APIs show their age even more
  - Most important APIs (Collections) are based on interfaces
  - Can’t add to interfaces without breaking source compatibility
- Adding closures, but not upgrading the APIs to use them effectively, would be silly
  - “What do you mean, I can’t say coll.forEach(lambda)? “

Know your inner poster child

- Most language evolution stories have a “poster child”
  - A canonical use case that can’t be supported in the “old” language
- For lambda: filter/map/reduce on collections

```
List<Student> students = ...
double highestScore =
  students.filter(s -> s.gradYear == 2011)
          .map(Student::getScore)
          .reduce(Integer::max);
```

- We need a mechanism for *library evolution*

LIBRARY EVOLUTION

Library-based evolution options

1. Don’t evolve existing collections, but add some new parallel implementation classes
   - Doesn’t do anything for serial programming model
   - Far removed from existing collections
2. Build “Collections 2”
   - Unrealistic – the Collections types infect the entire library
   - Huge design space
   - Still stuck with VM limitations (32-bit arrays, erasure)
3. Add aggregate operations to concrete classes
   - Hurts everyone who follows best-practice style advice
   - Code riddled with casts
   - Ties users to implementation instead of interface contracts
4. Add new interfaces (BetterList) with aggregate ops
   - More casts
   - Where does this lead to? NewBetterImprovedList2?

Language-based evolution options

1. Static extension methods
   - What appears to be an instance method call is really a static method call
   - Suppose static method `Collections.map(List, Mapper)` is available, and imported as a static extension – then `ls.map(…)` would compile to `Collections.map(ls, …)`
   - Simple to implement, and "proven" in C#
   - Programmer must reason about extensions in scope for each call
   - Subclass cannot provide a better implementation or covariant return
   - Poor interaction with instance methods of same name
   - Invisible to reflection

Language-based evolution options

2. Expanders, Classboxes
   - Unproven, risky
   - Expanders can't be applied to interfaces ☹ because a class might implement multiple interfaces and a client might "use" multiple expanders → call-site ambiguity.
3. Modularity-based solutions
   - Components in Jiazzi and Fortress
   - Information hiding techniques in Jigsaw and OSGi
   - Complex, risky
4. Traits
   - Too big a change

API evolution is a cross-cutting concern

- Library changes alone don’t help
- Nor do simple-minded language changes

- The fundamental problem is that we can’t evolve interface-based APIs
- This burden should fall to implementors, not users
  - Relatively fewer implementors
  - New methods often have an “obvious” implementation
  - Let the implementor provide a mostly reasonable default
  - Subclasses can override it at their leisure

Solution: virtual extension methods

- A virtual extension method is specified in the interface

```
interface Collection<T> {
    // existing methods, plus
    void forEach(Block<T> block) default Collections.<T>forEach;
}
```

- The forEach method is an extension method
  - From caller's perspective, an ordinary virtual method
- Collection provides a default implementation
  - Default is only used when implementation classes do not provide a body for the extension method
  - "If you cannot afford an implementation of forEach, one will be provided for you at no charge."

“Is this multiple inheritance in Java?!”

- Yes, but Java already has multiple inheritance of *types*
- This adds multiple inheritance of *behavior*, not state
- Multiple inheritance still a problem due to high complexity of separate compilation and dynamic linking

- Library evolution may be the primary motivator, but useful as an inheritance mechanism in itself
  - Almost all the way to stateless traits

METHOD RESOLUTION

Compatibility goals

- The point of this feature is being able to *compatibly* evolve APIs
- Compatibility has multiple faces
  - Source compatibility: Implementation class recompiles successfully
  - Binary compatibility: Callers continue to link correctly
- The key operation we care about is *adding new methods with defaults* to existing interfaces
  - *Without* necessarily recompiling the implementation class
- Also care about adding defaults to existing methods, and changing defaults on existing extension methods
  - Removals of most kinds are unlikely to be compatible

Compatibility is hard because of separate compilation

- Most interfaces are defined in libraries and consumed in applications
- Cannot count on application being recompiled
  - No guarantee that the same classes or symbols that were present at compile time will be there at runtime
- Compatibility means tolerating changes to interfaces without recompiling implementations and/or clients
  - Assumptions of uniqueness at compile-time must be verified at runtime, and if untrue, mitigated

Method resolution

- Key choice: treat inheritance of behavior from classes and interfaces separately
  - If there is an inheritance conflict between a superclass and a superinterface, the superclass takes priority
  - Defaults only come into play if there is no declaration for that method in the superclass chain
- Conflicts between superinterfaces may be resolved using subtyping
  - More specific interfaces take priority over less
- Invocation is resolved to a default if there is a *unique*, *most-specific default-providing interface*

Method resolution
Pruning less specific interfaces

- If interface B extends A, then B is *more specific* than A
  - If both A and B provide a default, we remove A from consideration because B is more specific
- Below, the fact that C<T> declares Collection<T> as an immediate supertype is irrelevant
  - Set is more specific and it provides a default, so it beats Collection

```
interface Collection<T> {
    public Collection<T> filter(Predicate<T> p) default …;
}
interface Set<T> extends Collection<T> {
    public Set<T> filter(Predicate<T> p) default …;
}
class D<T> implements Set<T> { ... }
class C<T> extends D<T> implements Collection<T> { … }
```

Method resolution
Handling diamonds

- We care about the identity of the interface providing the default, not the default itself

```
interface A { void m() default X.a; }
interface B extends A { }
interface C extends A { }
class D implements B, C { ... }
```

- When analyzing D, it is A that is the provider of the default, and it is unique
  - Therefore d.m(args) resolves to X.a(d, args)

Method resolution
Explicit disambiguation

- If ambiguous multiple inheritance is detected at compile-time, the programmer must provide a non-abstract method

```
interface A { void m() default X; }
interface B { void m() default Y; }

class C implements A, B {
  /* Required */ public void m() { A.super.m(); }
}
```

- New syntax *X*.super.*m*() to invoke the default in superinterface *X*

Modelling compilation and linking of virtual extension methods

- The type checking and method resolution rules are specified by a formal model
- Separated into type rules and resolution rules
  - Compiler would implement both
  - Runtime implements only resolution rules

Formal model: candidate computation

- We compute a set of candidate interfaces to provide a default for method m()
  - Pruning out less-specific candidates

$$\text{R-IntDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ \texttt{default}\ \langle\ k \mid \texttt{none}\ \rangle\ \}}{dcand(I) = \{\ I\ \}}$$

$$\text{R-IntInh}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ T\ m()\ ]\ \} \\ S = \bigcup_i\ dcand(I_i)\end{array}}{dcand(I) = \{\ W \in S\ :\ \forall_{V \in S}\ V <: W\ \Rightarrow\ V = W\ \}}$$

$$\text{R-ClassInh}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \} \\ S = \bigcup_{U \in \{\ D, I_1, ..., I_n\ \}} dcand(U)\end{array}}{dcand(C) = \{\ W \in S\ :\ \forall_{V \in S}\ V <: W\ \Rightarrow\ V = W\ \}}$$

Formal model: method resolution

- For each class, calculate:
  - Do I have or inherit a declaration for m() (C HasDefn)
    - If so, what class provides this (mprov(C))
  - Do I have a concrete body for m() (T HasBody)

$$\text{R-ClassBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ T\ m()\ \langle\ b \mid\ \texttt{abstract}\ \rangle\ \}}{mprov(C) = C \qquad C\ \texttt{HasDefn}}$$

$$\text{R-ClassInhBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \}\quad D\ \texttt{HasDefn}}{mprov(C) = mprov(D) \qquad C\ \texttt{HasDefn}}$$

$$\text{R-HasBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ T\ m()\ b\ \}}{C\ \texttt{HasBody}}$$

$$\text{R-HasDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ \ \texttt{default}\ k\ \}}{I\ \texttt{HasBody}}$$

Formal model: method resolution

- If the class or superclass has a declaration for m(), that wins
  - If that declaration is abstract, then there is no resolution
- Otherwise, if there is a unique most-specific default-providing interface, use that
  - If that interface has an “abstract default”, also no resolution

$$\text{R-ResolveImpl}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad C\ \texttt{HasDefn} \quad T = mprov(C) \quad T\ \texttt{HasBody}}{mres(C) = T}$$

$$\text{R-ResolveDef}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad \neg C\ \texttt{HasDefn} \quad |\ dcand(C)\ | = 1 \quad \exists_{J \in dcand(C)} \quad J\ \texttt{HasBody}}{mres(C) = J}$$

Formal model: limitations

- No method overloading (not really needed)
- No resolution of super invocations (can be added)
- No bridge methods
  - Irrelevant to general understanding
  - Big problem for implementation

Bridge methods: an inconvenient truth

- Java 5 added generics and covariant overrides
- These broke the 1:1 correspondence between methods in Java source and methods in class files
- Compiler generates “bridge methods” to make up for differences in the language and VM type systems
- Compiler knows that the two signatures are the “same” method, but the VM does not
  - A “semantic gap” in compile-time v. runtime resolution
  - Arguably bridges should have been a VM feature

Bridge methods

```
interface A<T> { void m(T t); }
interface B    { void m(String s); }
class C implements A<String>, B {
    public void m(String s) { … }
}
```

- Because of erasure, a variable of class C must respond to two signatures: m(String) and m(Object)
  - Compiler emits both methods
  - Object version “redirects” (with a cast) to the String version
- We need to defend (i.e. resolve defaults for) both
  - Need to know at runtime these are really the same method!
  - Compiler can’t generate the bridges for us, because methods with defaults might be added to superinterfaces *after C is compiled*

Compatibility: the good news

- The operations we care about are:
  - add-meth – add a method with a default to an interface
  - add-def – add a default to an existing interface method
  - mod-def – change the default of an existing method
- mod-def: Always source and binary compatible
- add-meth, add-def: Binary compatible if:
  - (Easy case) The resulting program is globally consistent (recompiling all sources would succeed)
  - (Hard case) The new interface method does not create conflicting defaults or invalid overridings

Compatibility: the hard cases

- Conflicting defaults: Add an extension method which is identical to an extension method in another interface, and classes exist that implement both interfaces

```
// red code added later with separate compilation
interface A { void m() default X.a; }
interface B { void m() default Y.b; }
class C implements A, B { }
```

- Invalid overriding: add an extension method whose signature matches one from a subclass, but whose return type is not compatible
  - This problem exists already in Java, where a new method is added to a superclass

Compatibility: further work needed

- Solutions to each of these problems involve tradeoffs against complexity of method resolution
  - Could impose a linearization order on candidate interfaces that could be used to resolve incompatibilities
  - Could use properties of the call site (e.g., the interface name referred to by an invokeinterface instruction)
  - Could store additional compile-time state in the class file
- We care more about avoiding binary incompatibilities than source incompatibilities
  - After-the-fact source incompatibilities can be mitigated by module dependencies or by explicit disambiguation (***`intf`*`.super.`*`m`*`()`**)

Implementation strategies

- Compiler techniques
  - Compile-time injection of default bodies into classes
    - Brittle, contradicts dynamic linking imperative
  - Translate invocations of extension methods using invokedynamic, and let bootstrap resolve default
    - Creates yet another way to invoke methods
    - Creates binary incompatibilities
- VM techniques
  - Classload-time injection of default bodies into classes
  - Integrated with vtable building
- Are virtual extension methods a language or VM feature?
  - Reality: everything else about inheritance is a VM feature
  - Trying to implement otherwise would cause visible seams

EVOLVING COLLECTIONS

Remember what we wanted to write?

```
List<Student> students = ...
double highestScore =
  students.filter(s -> s.gradYear == 2011)
          .map(Student::getScore)
          .reduce(Integer::max);
```

Need to add new methods to Collection / List

Lots of design choices here

- Eager vs lazy
- In-place vs create-new
- Serial vs parallel
- Where in the hierarchy to put new methods

Attractive target: Iterable

Add higher-order, streaming operations to Iterable

- All collections get these for free
- Default easily implemented in terms of iterator()

```
public interface Iterable<T> {
  Iterator<T>     iterator();
  boolean         isEmpty() default ...;
  void            forEach(Block<? super T> block)            default ...;
  Iterable<T>     filter(Predicate<? super T> predicate)     default ...;
  <U> Iterable<U> map(Mapper<? super T, ? extends U> mapper) default ...;
  T               reduce(T base, Operator<T> reducer)        default ...;
  Iterable<T>     sorted(Comparator<? super T> comp)         default ...;
  <C extends Collection<? super T>> C into(C collection)     default ...;
  // and more...
}
```

Eager v. lazy

- New methods are a mix of lazy and eager
  - Naturally lazy operations: filter, map, cumulate
  - Naturally eager operations: reduce, forEach, into
- Many useful operations can be represented as pipelines of source-lazy-lazy-eager
  - collection-filter-map-reduce
  - array-map-sorted-foreach
- In which case the laziness is mostly invisible
  - No new abstractions for LazyCollection, LazyList, etc

Going parallel

- A collection knows how to iterate itself
  - By augmenting Iterable, all Collections types now expose bulk data operations
  - Without changing the Collections implementations!
  - Individual classes can always override default implementations
- Can we do the same for parallel algorithms?
  - What is the parallel equivalent of Iterable?

Parallel Iterable is … Spliterable

A Spliterable can decomposed into two smaller chunks

- Small chunks can be iterated sequentially

Most data structures (arrays, trees, maps) admit a natural means of subdividing themselves

- Default implementations work in terms of Spliterable
- Add parallel() method to collections to dispense a Spliterable

```
public interface Iterable<T> {
  Iterator<T>     iterator();
  Spliterable<T>  parallel();
  // … Lots of extension methods follow …
}
```

Methods on Spliterable are analogous to Iterable

```
public interface Spliterable<T> {
  Iterator<T>     iterator();
  Spliterable<T>  left();
  Spliterable<T>  right();
  boolean            isEmpty() default ...;
  void               forEach(Block<? super T> block)            default ...;
  Spliterable<T>     filter(Predicate<? super T> predicate)     default ...;
  <U> Spliterable<U> map(Mapper<? super T, ? extends U> mapper) default ...;
  T                  reduce(T base, Operator<T> reducer)        default ...;
  Spliterable<T>     sorted(Comparator<? super T> comp)         default ...;
  <C extends Collection<? super T>> C into(C collection)        default ...;
  // and more...
}
```

Individual collection classes can always override default implementation

Explicit but unobtrusive parallelism

```
List<Student> students = ...
double highestScore =
  students.parallel()
          .filter(s -> s.gradYear == 2011)
          .map(Student::getScore)
          .reduce(Integer::max);
```

- More readable
- Better abstraction
- No reliance on mutable state
- Runs in parallel

- Implementation fuses three operations into one parallel pass
- Big win for data locality
- Works on any data structure that knows how to subdivide itself

Summary

- Java SE 8 will have closures for many reasons
  - Fundamental need to upgrade (not replace) libraries
  - No way to compatibly evolve interface-based APIs
- Java SE 8 adds virtual extension methods, a mechanism for compatible interface evolution
  - An upgrade to existing interface inheritance, so that classes may inherit behavior from interfaces
  - Looks like a language feature, but mplementation is in the JVM, reducing impact on class file consumers
- Java SE 8 evolves the language, libraries, and VM together