Virtual Extension Methods
(or, wedging multiple inheritance into the JVM)

Brian Goetz
Java Language Architect, Oracle Corporation

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.

New language features for Java SE 8

- Lambda expressions (closures)

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

- SAM conversion

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

- More type inference, e.g. lambda formals

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

- Method references

```
Predicate<> p = String->isEmpty
```

- Exception transparency (maybe)
- Virtual extension methods (aka *defender methods*)

Why *these* features?

- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point
- Provide libraries a path to multicore
  - Internal iteration needed to make data structures parallel-friendly
  - Today, developer's primary tool for computing over aggregates is the (fundamentally serial) for loop
- Empower library developers
  - Easier to evolve the programming model through libraries than through language
  - Enable developers to evolve interface-based APIs over time

Goals

- Encourage the creation of more abstract, *high-performance* libraries
    - Secondary goal: encourage a more side-effect-free programming model
- Simplify the consumption of such libraries through a concise code-as-data mechanism
- Provide for better library evolution and migration
    - Collections are looking long in the tooth
    - Lambdas without broad library support would be disappointing
- Secondary goal: keep doors open
    - Function types (but requires reification)
    - Control abstraction (but lots of work needed to get there)

Why extension methods?

- Adding closures is a big language change
- If Java had closures from day 1, our APIs would definitely look different
  - So adding closures now makes our APIs show their age!
  - 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 collection.forEach(lambda)?
- Therefore we need a mechanism for *interface evolution*

Static extension methods

- C# has *static extension methods*
- A static extension method is a tuple (T, n, D, m)
  - Calls to t.n(args) rewritten at compile time as D.m(t, args)
- Advantages
  - Simple to implement
  - No VM changes
- Limitations
  - Brittle – if default changes, clients have to be recompiled
  - No covariant overrides
  - Not reflectively discoverable
  - Poor interaction with existing instance methods of same name
  - Extended class cannot provide a “better” implementation
  - Not very object-oriented

Solution: *virtual* extension methods

- Virtual extension methods 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."

Virtual extension methods

- Within I, extension methods are a tuple (n, D, m)
  - Calls to i.m(args) are rewritten *at run time* to D.m(i, args)
- Gack, is this multiple inheritance in Java?
  - Yes, but Java already has multiple inheritance of *types*
  - This adds multiple inheritance of *behavior* too
    - But not state!
    - Abstract classes still relevant for representation
  - Multiple inheritance still a source of complexity due to separate compilation and dynamic linking
- API evolution may be the primary motivator, but useful as an inheritance mechanism in itself

Method resolution

- The rules treat inheritance of behavior from classes and interfaces separately
- Declarations in classes always win over interfaces
  - Follow the implementation hierarchy upwards
  - If you find a concrete body, OR a declaration that the method is abstract, stop
  - Only then consider defaults provided by interfaces
- Declarations in more-specific (under subtyping) interfaces win over less-specific interfaces
- 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

```
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> { … }
```

- Here, the fact that C<T> declares Collection<T> as an immediate supertype is irrelevant
  - Set is more specific and also provides a default, so it wins over Collection

Method resolution
Handling diamonds

- We track not the identity of the default, but the interface that provides it

```
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)
    - Diamonds are a problem for state inheritance, not behavior

But wait, there’s math

- The type checking and method resolution rules are specified by a formal model (excerpts here)

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

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

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

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

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

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

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

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

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

Compatibility goals

- The whole point of this feature is being able to *compatibly* evolve APIs
- Compatibility has multiple faces
  - Source compatibility
  - Binary compatibility
- The key operation we care about is *adding new methods with defaults* to existing interfaces
  - 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 goals

- How to achieve source and binary compatibility for addition of extension methods is not fully solved
  - Almost there – solved for programs that are globally consistent (i.e., would compile if recompiled from scratch)
  - Damn that pesky separate compilation!
- Currently several vectors through which an “innocent” change to an interface can break code
  - Add an extension method whose signature matches that of another method but whose return type is not compatible
    - This problem existed before, but went untriggered because changes to interfaces in standalone libraries were rare
  - Add an extension method which is identical to an extension method in another interface, and classes exist that implement both interfaces

Compatibility goals

- The solutions to each of these problems involve tradeoffs between complexity of method resolution, and the set of incompatible changes
  - Three kinds of solutions
    - Storing additional as-compiled state in the classfile
    - Using properties of the call site (e.g., interface through which invokeinterface is invoked)
    - Imposing a linearization order on candidate interfaces that could be used to resolve incompatibilities
- We care more about avoiding binary incompatibilities than source incompatibilities
  - After-the-fact source incompatibilities can be mitigated by module dependencies

How to implement?

- There are many possible 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
- Big question: is this a language or VM feature?
  - Reality: everything else about inheritance is a VM feature
  - Trying to implement otherwise would cause visible seams

Bridge methods rear their ugly head

- In Java 5, we added generics and covariant overrides
  - These broke the 1:1 correspondence between methods in Java source code and methods in classfiles
  - Compiler needs to generate “bridge methods” to make up for differences between the language and VM type systems
  - This happens with both covariant overrides and with generic type substitution
- The compiler knows that the two signatures are the same method, but the VM does not
  - Arguably this should have been a VM feature, but we took the easy route and did it in the compiler

Bridge methods

- Example:

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

- Here, instances of C must respond to both signatures: `m(Ljava/lang/String;)` and `m(java/lang/Object;)`
  - Compiler generates the Object version which redirects to the String version
  - We need to be prepared to resolve defaults for both
  - Need to know at runtime these are really the same method!
    - “A simple matter of programming”
  - In the long run should probably push bridges into the VM
    - This problem also shows up with SAM conversion

Consequences for non-Java languages

- By making this a VM feature, non-Java languages can remain mostly ignorant of extension methods
  - Can invoke extension methods through invokeinterface without having to know that they are extension methods or how they are resolved
  - Can generate classes that implements an interface, and if new methods are added to the interface after compilation, defaults still work
  - Can generate interfaces with default implementations and use as a composition mechanism
  - Can package language-specific runtime functionality into interfaces that Java classes can “mix in”

Summary

- Virtual extension methods are an upgrade to existing interface inheritance, where classes can inherit behavior from interfaces
- Goal is to allow interfaces to be evolved without breaking existing implementations
  - Though also presents new options for composing functionality
- Implementation is as a VM feature, reducing impact on classfile consumers