Featherweight Defenders: A formal model for virtual extension methods in Java

Brian Goetz and Robert Field, Oracle Corporation*

December 2, 2011

1 Introduction

As a means of modeling the semantics of *virtual extension methods* (also known as *defender methods*) in Java, we describe a lightweight model of Java in the style of *Featherweight Java* (Pierce et al.), called *Featherweight Defenders* (or FD).

This model retains many aspects of Java's class and method inheritance. We drastically simplify method overloading and naming (classes and interfaces can have only one method, named `m()`, with no arguments), but retain single inheritance of classes and multiple inheritance of interfaces, covariant overriding, abstract and concrete class methods, and reabstraction of concrete class methods (where a concrete method is overridden with an abstract one).

We add in a new feature not yet present in Java, *virtual extension methods*, where an interface can provide a default implementation for a method. Interface methods may be abstract definitions (no default clause), specify a concrete default (`default` k) or may explicitly cancel a default inherited from supertypes (`default none`), which is akin to reabstraction of concrete methods in classes.

The goal of this model is twofold: to capture the rules regarding well-formedness of types in the presence of extension methods, and to capture the runtime mapping of method names within a class to concrete method bodies which provide the implementation of that method for that class, which we call the *linkage* of the method `m()` in C.

We believe that this model includes the most significant inheritance features that are relevant to method linkage in the presence of extension methods. There are some additional features relevant to the implementation, such as bridge methods, that are beyond the scope of this model.

*The authors gratefully acknowlege the assistance of Sukyoung Ryu of KAIST in the refinement of this formal model.

$$
\begin{array}{rcl}
T & ::= & \texttt{Object} \mid C \mid I \\
K & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ [\ R\ m()\ \langle\ b\ \mid\ \texttt{abstract}\ \rangle\ ]\ \} \\
L & ::= & \texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ R\ m()\ [\ \texttt{default}\ \langle\ k\ \mid\ \texttt{none}\ \rangle\ ]\ ]\ \}
\end{array}
$$

Figure 1: FD language syntax

2 Syntax

The metavariables C, D, and E (and their derivatives) range over class names and the metavariable I, J, and K range over interface names. The metavariables T, R, U, V, and W range over all types (classes and interfaces). The metavariable S ranges over sets of types. The metavariable k ranges over a set of nominal identifiers, and the metavariable b ranges over a set of method bodies. Figure 1 shows the syntatic forms for FD.

3 Preliminaries

Figure 2 shows the subtyping judgements for classes and interfaces.

$$\text{S-Refl}\ \frac{}{T <: T}$$

$$\text{S-Trans}\ \frac{T <: V \qquad V <: W}{T <: W}$$

$$\text{S-Class}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\{...\}}{C <: D \qquad \forall_i\ C <: I_i}$$

$$\text{S-Intf}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\{...\}}{\forall_i\ I <: I_i}$$

Figure 2: Basic subtyping rules

As in Featherweight Java, we use "lookup functions" (such as $mtype(T)$) and "marker predicates" (such as T `OK`) in the inference rules to record information about types. These will be introduced as they are used by the inference rules.

4 Compile-time vs runtime

One of the goals of this model is to present a formal procedure for linking a method `m()` in a class C among the many candidate choices contributed by superclasses and interfaces. In languages like Java, method linkage is performed

both at compile time (to ensure that methods link uniquely, that required methods are implemented, and to reject source files that do not meet these requirements) and at run time (to perform linkage dynamically.) It is important to ensure that linkage decisions made at compile time and run time are consistent, but issues like separate compilation and dynamic linking pose challenges to this goal, since class files compiled separately may not provide a consistent view of the type hierarchy at run time, which may lead to runtime errors.

This model divides rules into two categories – typing-related (those beginning with T-) and linkage-related (those beginning with R-). A compiler would execute and enforce all the rules; the runtime would execute only the linkage rules. In a well-typed program, the compiler and runtime view of the world should be identical; in an inconsistently-typed program (say, due to separate compilation), it may still be possible to perform method linkage at runtime according to the R- rules. The treatment of separate compilation and dynamic linkage is outside the scope of this model.

5 Method typing

Figure 3 shows the typing rules for resolving the type of method `m()` in classes and interfaces. It defines the lookup function $mtype(T)$, which describes the return type of the method `m()` in T.

Central to the typing analysis is identifying whether a method `m()` in type T (whether defined explicitly or inherited) can legally override all signatures of `m()` from supertypes of T. Java supports covariant overrides – where a method in a subclass overrides a method in a supertype, but provides a narrower return type. If a method `m()` in T overrides one or more methods from supertypes, the return type of `m()` in T must be a subtype of each of the return types of `m()` from supertypes. If there is such a type, we say `m()` has a *well-defined return type* in T.

The value of $mtype(T)$ is a *set* of types, and is one of the following:

- The empty set $\{\}$, indicating that the method `m()` is not a member of T;
- The set $\{ U \}$, indicating that the method `m()` has a well-defined return type in T, and that return type is U;
- *undefined*, indicating that the method `m()` does not have a well-defined return type in T. The compiler should reject types for which $mtype(T)$ is *undefined*, but having a well-defined return type is not sufficient to declare that T is well-formed (for example, T could have conflicting defenders or problems with covariant overrides.)

Having defined *mtype* in this way, we define a relation for optional subtyping between a candidate type T and the return type of `m()` in U:

$$T \tilde{<:} mtype(U) \quad \equiv \quad mtype(U) = \{\} \quad \vee \quad [\, mtype(U) = \{\ V\ \} \ \wedge\ T <: V\ ]$$

Intuitively, if $T \tilde{<:} mtype(U)$, then T is a potentially valid return type for the method `m()` in a subtype of U.

We extend this concept by defining the function lbi_{mtype}, which computes an *inclusive lower bound* under optional subtyping for a projection under *mtype* of a set of types. (The inclusive lower bound for a set S is the lower bound of S if S contains its lower bound, and *undefined* otherwise.) We define $lbi_{mtype}(T_1, ..., T_n)$ as follows:

$$lbi_{mtype}(T_1, ..., T_n) = \begin{cases} \{\} & \text{if } \forall_i\ mtype(T_i) = \{\} \\ mtype(T_i) & \text{if } \exists_i \text{ such that } mtype(T_i) = \{\ U\ \}, \text{ and} \\ & \forall_{j \neq i}\ U <\tilde{:}\ mtype(T_j) \\ undefined & \text{otherwise} \end{cases}$$

We use lbi_{mtype} to determine whether there is a consistent return type for the method `m()` in T given the types of `m()` in the supertypes of T.

$$\text{T-ClassExpl}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \ T\ m()\ \langle\ b \mid \texttt{abstract}\ \rangle\ \} \\ T <\tilde{:}\ mtype(D) \qquad\qquad \forall_i\ T <\tilde{:}\ mtype(I_i)\end{array}}{mtype(C) = \{\ T\ \}}$$

$$\text{T-ClassImpl}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \}}{mtype(C) = lbi_{mtype}(D, I_1, ..., I_n)}$$

$$\text{T-IntExpl}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ [\ \texttt{default}\ \langle\ k \mid \texttt{none}\ \rangle\ ]\ \} \\ \forall_i\ T <\tilde{:}\ mtype(I_i)\end{array}}{mtype(I) = \{\ T\ \}}$$

$$\text{T-IntImpl}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ \}}{mtype(I) = lbi_{mtype}(I_1, ..., I_n)}$$

$$\text{T-Object}\ \frac{}{mtype(\texttt{Object}) = \{\} \qquad \texttt{Object OK}}$$

Figure 3: Method typing

6 Method inheritance

The rules so far only have to do with computing the membership of a method in a type, and the method's return type in that type, and do not consider where the method implementation codes from. We now explore the rules for inheriting methods, including inheritance of bodies from superclasses and inheritance of default implementations from superinterfaces.

All classes (except the root class `Object`) have a single superclass. We treat a concrete method body and a declaration that the method is abstract in the same way (collectively, we call these a method declaration). A class inherits method declarations from its superclass, unless the method is explicitly given a new declaration in the subclass.

A key aspect of inheritance of defender methods is *pruning* of less-specific default-providing interfaces from consideration in the linkage process. In Java, it is allowable for a class or interface to extend an interface both directly and indirectly, as in the following example:

```
interface A { Object m() default k }
interface B extends A { Object m() default l }
class C implements A, B { }
```

Here, `C` implements `A` both directly and indirectly. This idiom is common as a documentation device, and in Java 7 and earlier the additional declaration of `A` has no effect, because it is already implicit in the extension of `B`. This behavior should continue to hold true in the presence of extension methods.

The inheritance rules for extension methods calls for "redundant" inheritance from less-specific interfaces (such as `A` in the example above) to not be considered further in the inheritance decision, except inasmuch as the less-specific interface has already contributed to its subinterface.

We capture this pruning behavior by the function $prune(S)$:

$$prune(S) = \{\ W \in S\ :\ \forall_{V \in S}\ V <: W\ \Rightarrow\ V = W\ \}$$

Intuitively, the rules for method linkage behave as follows:

- A method defined in a type takes precedence over methods defined in its supertypes (R-IntDef, R-ClassBody).
- Abstract methods in interfaces (method declarations without default clauses) do not influence linkage (R-IntInh).
- A method declaration (concrete or abstract) inherited from a superclass takes precedence over a default inherited from an interface (R-LinkImpl).
- More specific default-providing interfaces take precedence over less specific ones (R-IntInh, R-ClassInh).
- If we are to link `m()` to a default method from an interface, there must a unique most specific default-providing interface to link to (R-LinkDef).

Figure 4 covers the rules for computing the candidates for inheriting methods from classes and interfaces. It defines the following lookup functions:

- $dcand(T)$ (defender candidates), which indicates the set of interfaces which could provide a default implementation for `m()` in T.
- $mprov(C)$ (method provenance), which indicates the most specific superclass from which C could inherit a method declaration for `m()`. Like *mtype*, the value of $mprov(C)$ is a set, which is either empty (there is no superclass that provides a declaration for `m()` in C), or a singleton set that contains the class providing the declaration for `m()` in C.

- T `HasBody`, which indicates that T declares either a concrete method body (for classes) or a default implementation (for interfaces). (It is not inherited; it is strictly a property of the class or interface declaration, and will be used in the final linkage decision.)

$$\text{R-Object}\ \frac{}{mprov(\texttt{Object}) = \{\} \qquad dcand(\texttt{Object}) = \{\}}$$

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

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

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

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

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

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

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

Figure 4: Inheritance candidates from classes and interfaces

7 Typechecking of interfaces and classes

We now consider the rules for when an interface or class should be accepted as well-typed. We use the marker predicate T `OK` to indicate that T is well-typed.

Figure 5 shows the typechecking rules for interfaces. For an interface I, I `OK` means:

$$\text{T-IntNoBody}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ ...\ \} \quad S = mtype(I) \quad \forall_i\ I_i\ \texttt{OK} \quad dcand(I) = \{\}}{I\ \texttt{OK}}$$

$$\text{T-IntInhBody}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ ...\ \} \quad \forall_i\ I_i\ \texttt{OK} \quad dcand(I) = \{\ J\ \} \quad mtype(I) = mtype(J)}{I\ \texttt{OK}}$$

Figure 5: Typechecking of interfaces

- All the supertypes of I are OK;
- If `m()` is a member of I, it has a well-defined return type in I;
- If `m()` is a member of I, then there is at most one defender candidate for `m()` in I, and if present, that defender candidate has the same return type as `m()` in I.

Figure 6 shows the typechecking rules for classes. For a class C, C `OK` means:

- All the supertypes of C are OK;
- If `m()` is a member of C, it has a well-defined return type in C;
- If C inherits a method declaration from a superclass, it has the same return type as `m()` in C.
- If C does not inherit a method declaration from a superclass, then there is at most one defender candidate for `m()` in C, and if present, that defender candidate has the same return type as `m()` in C.

Note that the typechecking rules do not reject classes which inherit abstract method definitions from superclasses (or "default none" methods from superinterfaces) and which do not provide an implementation. These classes are considered valid, but at runtime are identified as having no linkage. This is a consequence of simplifying the language model to not explicitly track classes as abstract vs concrete, and does not interfere with the goals of this model.

8 Method linkage

We are now able to define the linkage of `m()` in a class C which may inherit its implementation from a superclass or from a default method in an interface.

Figure 7 defines the lookup function $mres(C)$, which indicates the linkage or `m()` in class C. Its value is the type from which `m()` is inherited.

9 Superclass invocation linkage

Java provides a mechanism for a method overriding a method in a superclass to invoke the overridden method, using the `super.m()` syntax. We would wish

$$\text{T-ClassNone}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad D\ \texttt{OK} \quad \forall_i\ I_i\ \texttt{OK} \quad S = mtype(C) \quad mprov(C) = \{\} \quad dcand(C) = \{\}}{C\ \texttt{OK}}$$

$$\text{T-ClassFromSuper}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad D\ \texttt{OK} \quad \forall_i\ I_i\ \texttt{OK} \quad mprov(C) = \{\ E\ \} \quad mtype(C) = mtype(E)}{C\ \texttt{OK}}$$

$$\text{T-ClassFromDefault}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad D\ \texttt{OK} \quad \forall_i\ I_i\ \texttt{OK} \quad mprov(C) = \{\} \quad dcand(C) = \{\ J\ \} \quad mtype(C) = mtype(J)}{C\ \texttt{OK}}$$

Figure 6: Typechecking of classes

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

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

Figure 7: Linkage

$$\text{R-ClassSuper}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \\ mprov(D) = \{\} \qquad dcand(I_i) = \{\ J\ \} \\ J \in dcand(C) \qquad J\ \texttt{HasBody}\end{array}}{sres(C, I_i) = J}$$

$$\text{R-IntfSuper}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ ...\ \} \\ S = \bigcup_i\ dcand(I_i) \\ dcand(I_i) = \{\ J\ \} \quad J \in prune(S) \quad J\ \texttt{HasBody}\end{array}}{sres(I, I_i) = J}$$

Figure 8: Super-call linkage

to support a similar mechanism, where an interface or class overriding a default method from an interface I could invoke the overridden default as `I.super.m()`.

While the FD calculus does not provide syntax for method invocation, we can define a lookup function $sres(T, I)$ that would provide the desired linkage for a super-call to `m()` from T through interface I, just as we did for $mres(C)$. Figure 8 shows the calculation of $sres(T, I)$. For making a super-call from T through interface I, I must be an immediate super-interface of C, and the (possibly inherited) default provided by I must not be overridden by a more specific interface.