Interface evolution via “public defender” methods

Brian Goetz
Second draft, May 2010

1. Problem statement

Once published, it is impossible to add methods to an interface without breaking existing implementations. The longer the time since a library has been published, the more likely it is that this restriction will cause grief for its maintainers.

The addition of closures to the Java language in JDK 7 place additional stress on the aging Collection interfaces; one of the most significant benefits of closures is that it enables the development of more powerful libraries. It would be disappointing to add a language feature that enables better libraries while at the same time not extending the core libraries to take advantage of that feature[1].

V1 of the Lambda Strawman proposes static extension methods as a means of creating the *illusion* of adding methods to existing classes and interfaces, but they have significant limitations – for example, they cannot be overridden by classes that implement the interface being extended, so implementations are stuck with the “one size fits all” implementation provided as an extension[2].

2. Public defender methods (aka virtual extension methods)

In this document, we propose adding a mechanism for adding new methods to existing interfaces, which could be called *virtual extension methods*. Existing interfaces could be added to without compromising backward compatibility by adding *extension methods* to the interface, whose declaration would contain instructions for finding the default implementation in the event that implementers do not provide one. Listing 1 shows an example of the Set interface extended with a virtual extension method. The syntax is purely for illustrative purposes.

```
public interface Set<T> extends Collection<T> {
    public int size();
    // The rest of the existing Set methods

    extension T reduce(Reducer<T> r)
        default Collections.<T>setReducer;
}
```

Listing 1. Example virtual extension method.

[1] Obvious candidates for evolving the Collections classes include the addition of methods like forEach(), filter(), map(), and reduce().

[2] In general, static-ness is a source of all sorts of problems in Java, so adding more static mechanisms is likely a step in the wrong direction.

The declaration of reduce() tells us that this is an extension method, which must have a "default" clause. The default clause names a static method whose signature must match[3] that of the extension method, with the type of the enclosing interface inserted as the first argument[4].

Implementations of Set are free to provide an implementation of reduce(), since it is a virtual method just like any other method in the interface. If they do not, the default implementation will be used instead. You could call these "public defender" methods (or *defender methods* for short) since they are akin to the Miranda warning: "if you cannot afford an implementation of this method, one will be provided for you."

An interface that has one or more extension methods is called an *extended interface*.

3. Method dispatch

Runtime dispatch of defender methods is slightly different from that of ordinary interface method invocation. With an ordinary interface method invocation on a receiver of type D, first D is searched for a method implementation, then its superclass, and so on until we reach Object, and if an implementation is not found, a linkage exception is thrown. For a defender method, the search path is more complicated; first we search for actual implementations in D and its superclasses, and then we must search for the *most specific* default implementation among D's interfaces, failing if we cannot find a unique most specific default.

Method resolution is as follows:

1. First perform the standard search, from receiver class proceeding upward through superclasses to Object. If an implementation is found, the search is resolved successfully.
2. Construct the list of interfaces implemented by D, directly or indirectly, which provide a default for the method in question.
3. Remove all interfaces from this list which is a superinterface of any other interface on the list (e.g., if D implements both Set and Collection, and both provide a default for this method, remove Collection from the list.)
4. Construct the set of default methods corresponding to the list of interfaces arrived at in step (2). If the result has a single item (either because there was only one default or multiple interfaces provided the same default), the search is resolved successfully. If the resulting set has multiple items, then throw a linkage exception indicating conflicting defender methods.

Resolution need be performed only once per implementing class.

---

[3] Modulo allowed covariance in return type, contravariance in argument types, and commutativity and covariance in thrown exception types.

[4] The syntax of specifying the default method should match that of specifying method references, if method references are to be added to the language.

3.1. Resolving ambiguity

The last step in the resolution procedure above addresses the case of ambiguous default implementations. If a class implements two extended interfaces which provide the same default implementation for the same method, then there is no ambiguity. If one default method "shadows" another, then the most specific one is taken. If two or more extended interfaces provide different default implementations, then this is handled just as if the class did not provide an implementation at all, and a linkage exception is thrown indicating conflicting default implementations.

4. Classfile support

The compilation of extended interfaces and defender methods is very similar to ordinary interfaces. Defender methods are compiled as abstract methods just as ordinary interface methods. The only differences are:

- The class should be marked as an extended interface. This could be done by an additional accessibility bit (ACC_EXTENDED_INTERFACE), an additional class attribute, or simply inferred from the presence of defender methods.
- Defender methods should be marked as such and refer to their default implementation. This could be done by setting an additional accessibility bit (ACC_DEFENDER) in the access_flags field of the method_info structure, and/or accompanied by an additional attribute in the method_info structure which will store a reference to the default implementation:

```
Defender_attribute {
    // index of the constant string "Defender"
    u2 attribute_name_index;
    // Must be 4
    u4 attribute_length;
    // CP index of a MethodRef naming the default
    u2 default_name_index;
    // CP index referring to descriptor for default
    u2 default_descriptor_index;
}
```

5. Additional language and compiler support

It may be the case that a class implementing an extended interface wishes to call the default implementation, such as the case where the class wants to decorate the call to the default implementation. An implementation of a defender method can refer to the default implementation by "InterfaceName.super.methodName()", using syntax inspired by references to enclosing class instances of inner classes. If the class does not have multiple supertypes that specify this method name and a compatible signature, we may wish to additionally allow the simpler syntax "super.methodName()".

6. Implementation strategies

There are several possible implementation strategies, with varying impact on the VM and tools ecosystem. Sensible strategies include:

1. At class load time, for non-abstract classes that implement extended interfaces and do not implement all the extended methods of that interface, a bridge method is woven in that invokes the appropriate default implementation. This could be done by the class loader at execution time, or could be done before execution when a module is loaded into a module repository.
2. Like (1), but the bridge method invokes the default through an invokedynamic call which defers the target resolution to the first time the method is called (at which point the bootstrap handler gets out of the way and no further resolution is required.)
3. Like (2), but additionally the static compiler inserts similar bridge methods when compiling implementations of extended interfaces. This has the effect of (a) offloading effort from the VM and (b) reducing the impact on classfile-consuming tools, and can be done without changing the semantics of method dispatch at all.
4. At class load time, client code that *calls* defender methods is rewritten to use an invokedynamic call.

It is desirable that class modification behavior in the VM be implemented as far to the periphery as possible, to minimize the impact on core VM functions such as method dispatch, interpretation, and reflection. Doing so in the class loader (or at module deployment time) seems a sensible approach that minimize the impact on the VM.

7. Implementation option 1: load-time weaving with invokestatic

When a non-abstract class is loaded, the set of extended interfaces it implements (directly or indirectly) is constructed, and the class (including its superclasses) are searched for an implementation of each extension method. For each extension method that does not have an implementation in the class or superclasses, a search for the correct default is performed as per "Method Dispatch" above. If that search produces a unique result, then a bridge method is inserted into the newly loaded class which simply forwards the receiver and invocation parameters to the default implementation with an invokestatic. If the search does not produce a unique result, then the bridge method throws a linkage error (similar to the bridge methods inserted today throwing AbstractMethodError when a class is loaded that implements an interface which has methods not present in the implementation class.)

One complication is that the initial search for implementing methods needs to ignore the bridge methods previously inserted at load time on superclasses; since a superclass is loaded before a subclass, the superclass will already have had its bridge methods injected, but we do not want this to appear to a subclass that the superclass actually provides an implementation. This can be done by marking the bridge methods as being not present in the original implementation, and ignoring so-marked methods when searching superclasses for an actual implementation.

8. Implementation option 2: load-time weaving with invokedynamic

This approach is nearly identical to the previous option, except that the bridge methods do not need to resolve the linkage at class load time, they merely need to insert a bridge method that invokes the default through an invokedynamic call site. The bootstrap

handler implements the method resolution logic, which is executed on the first call to the bridge method. At that time, resolution is performed, and the bootstrap handler returns a DirectMethodHandle so that on subsequent calls the invocation can be optimized as if it were an invokestatic.

This approach has the benefit of not burdening class loading with the effort of defender method resolution, deferring the cost of resolution to the first time the method is invoked. (As many extension methods may never be invoked, this may be a desirable trade-off.) The disadvantage is that it uses a somewhat more complex mechanism – dynamic dispatch.

9. Implementation option 3: load-time + compile-time weaving

This approach supplements the previous approach (load-time weaving with invokedynamic) with compiler weaving which performs exactly the same process as the load-time weaving – insertion of a bridge method that calls the default through an invokedynamic invocation.

The benefit of adding compile-time weaving is (a) further offloading impact from the VM and (b) offloading impact from classfile-consuming tools. If done in conjunction with dynamically computing the call target via invokedynamic, any potential brittleness impact from compile-time weaving can be eliminated.

10. Implementation option 4: client-side weaving

This option takes a different strategy: invokeinterface calls to defender methods are rewritten to use an invokedynamic call site with similar resolution behavior as the previous strategies.

However, unlike the implementation-side weaving strategies, this strategy leaves a number of holes that then have to be patched, such as reflection support (reflective calls to defender methods would require special treatment in the runtime), dynamic proxy support, etc.

11. Conflicting methods

It is possible that two extended interfaces may provide incompatible extension methods (such as two interfaces providing methods with the same name and argument types but different return types.) At compile time, such files will be rejected (as they are today) by the static compiler. At load time, conflicting methods can be resolved as for nonconflicting methods – even though the Java source language does not permit conflicting methods, the class file format has no such restriction. Since virtual method invocation is done with explicit signatures, weaving of default implementations for conflicting methods can proceed as normally.

12. Source compatibility

It is possible that this scheme could introduce source incompatibilities to the extent that library interfaces are modified to insert new methods that are incompatible with methods in existing classes. (For example, if a class has a float-valued xyz() method and implements Collection, and we add an int-valued xyz() method to Collection, the existing class will no longer compile.)

13. Binary compatibility

Existing client classes compiled against previous versions of the interface still behave just as before, as implementations of the extended interface are still valid implementations of the pre-extension interface (because we can only add compatible method signatures, not change or remove method signatures.)

Calls to existing methods of existing implementations of extended interfaces behave as before.

14. Effect on the language

The intent of this feature is to render interfaces more malleable, allowing them to be extended over time by providing new methods so long as a default (which is restricted to using the public interface of the interface being extended) is provided. This addition to the language moves us a step towards interfaces being more like "mixins" or "traits". However, developers may choose to start using this feature for new interfaces for reasons other than interface evolution.

For example, it is quite conceivable that developers might choose to give up on abstract classes entirely, instead preferring to use defender methods for all but a few interface methods. This might result in an interface like Set looking like Listing 2.

```
public interface Set<T> extends Collection<T> {
    extension public int size()
        default AbstractSetMethods.size;

    extension public boolean isEmpty()
        default AbstractSetMethods.isEmpty;

    // The rest of the Set methods, most having defaults
}
```

Listing 2. Possible use of defender methods.

The prevailing wisdom in API design (see Effective Java) is to define types with interfaces and skeletal implementations with companion abstract classes. Defender methods allow users to skip the skeletal implementation. This has the disadvantage of adding another way to do something that is already well served, but the new way has advantages too, allowing the inheritance of behavior (but not state) from multiple sources, reducing code duplication or forwarding methods.

15. Syntax options

There are a number of possible ways to specify defender methods in addition to the syntax used in the examples.

One alternative that was explored was to put the default method body right into the interface source file. I think that this would be confusing for users; many already don't quite get the difference between interfaces and abstract classes. (This would be more appropriate if we went to full-blown traits.)

The use of the "extension" keyword (or similar, such as "optional") is not even needed; the "default" clause (which is already a Java keyword) may be sufficient.

16. Effect on dynamic proxies

Authors of dynamic proxies may well have assumed that the set of methods implemented by a given interface was fixed, and embodied this assumption into any dynamic proxies coded for that interface. Such dynamic proxy implementations may well fail when an extension method is called on the proxy. However, defensively coded dynamic proxies will likely continue to work, since most proxies intercept a specific subset of methods but pass others on to the underlying proxied object.

17. Restrictions

Defender methods will not be allowed on annotation interfaces (@interfaces.)

18. Possible generalizations

As we look towards language and library evolution, the evolution mechanism here for adding methods to interfaces may be generalized to a number of similar evolution problems. The mechanism we are proposing does similar classfile transformations at both static compilation and runtime class load time in aid of migration across incompatible API changes. Other possible applications of such a mechanism might include: supporting deprecation, method signature migration (allowing Collection.size() to return long instead of int), superclass migration (e.g., migrating from “class Properties extends Hashtable” to “class Properties implements Map<String, String>”), etc. Such a generalized mechanism would likely remain internal to the platform, but would provide a vehicle for solving other migration problems in the future.