Interface evolution via “public defender” methods

Brian Goetz
Third draft, August 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 proposed C#-style 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 outline 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. Extension methods are virtual methods just like other interface methods, but are specially tagged as extension methods and provide a reference to a default implementation (which is a static method whose signature matches the signature of the extension method, with the type of the extended interface prepended to the argument list). Listing 1 shows an example of the Set interface extended with a virtual extension method.

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

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

[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.

```
}
```
Listing 1. Example virtual extension method.

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]. It is an error if the default method cannot be found at compile time.

Implementations of Set are free to provide an implementation of reduce(), since it is a virtual method just like any other interface method. 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*.

It is a deliberate choice to not simply allow the programmer to include the code of the default inline within the interface. Not only would this violate the long-held rule that "interfaces don't have code", but by specifying the default by name rather than by value, it is easier for the runtime to identify whether two defaults are in fact the same method, which is important for conflict resolution.

3. Method dispatch

Runtime dispatch of extension 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 an extension method, the search path is more complicated; first we search for actual implementations in D and its superclasses just as before, and if we do not find an implementation of the method in the implementation hierarchy, then we must search for the *most specific* default implementation among D's interfaces (which is not a linear search), failing if we cannot find a unique most specific default.

Method resolution for extension methods 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.)

[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.

4. Construct the set of distinct 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 extension methods.

Resolution need be performed only once per extension method per implementing class. (This makes it an ideal candidate for dynamic linkage with invokedynamic.)

3.1. Conflicting or redundant defaults

The last step in the resolution procedure above addresses the case of ambiguous default implementations. If a class implements two or more extended interfaces which provide the same default implementation for the same method, then there is no ambiguity. If one default method "shadows" another (where a subinterface provides a different default for an extension method declared in a superinterface), then the most specific one is taken and again there is no ambiguity. 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 extension methods is very similar to ordinary interfaces. Extension methods are compiled as abstract methods just as ordinary interface methods. The only differences are:

- The class should be identifiable 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 extension methods. (Because classfile bits are in short supply, we may wish to infer that a method is an extension method or that an interface is an extended interface entirely from the presence of extension method attributes rather than by accessibility bits for the class or method.)
- Extension methods should refer to their default implementation. This requires an additional attribute in the method_info structure which will store a reference to the default implementation (see below) and could also could include setting an additional accessibility bit (ACC_DEFENDER) in the access_flags field of the method_info structure:

```
struct Defender_attribute {
   u2 class_index;
   u2 method_index;
}
```

5. Reflection support

Adding extension methods creates the need for several additional reflective methods in java.lang.Class and java.lang.Method to identify extended interfaces, extension methods, and to find the default for an extension method.

```
class Class {
    // Is this class an extended interface?
    public boolean hasExtensionMethods();
}

class Method {
    // Is this method an extension method?
    public boolean isExtensionMethod();

    // For an extension method, what is its default?
    public Method getDefaultImplementation();
}
```

Listing 2. Reflection support.

The implementations for these methods map straightforwardly to the existence and contents of Defender attributes in the class. Because the default method exists in another class than the specifying interface, the getDefaultImplementation() method may trigger loading of the class hosting the default method.

6. 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 an extension 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()”.

7. Invocation of extension methods

Extension methods should be invoked like any other virtual method, with invokevirtual or invokeinterface. When invoking an extension method defined on interface A implemented by class C, the caller should see the same result whether the method is invoked with invokevirtual or invokeinterface. (Because the default is specified by the interface, it is theoretically possible to disambiguate between two conflicting extension methods when the method is invoked via invokeinterface rather than invokevirtual. However, this would be confusing; historically, Java objects have only had one implementation of a method even when it is specified in multiple interfaces.) Therefore all invocation paths to a method on a given receiver (whether invokevirtual, or invokeinterface through any interface) should arrive at the same target method.

Direct invocation of default implementations from implementing classes (using the InterfaceName.super.methodName() syntax) should be implemented as invokedynamic calls, where the method name and argument list encodes the desired interface, method name, and method signature. The bootstrap method for this call site should link the call site by determining the default method for that method for the specified interface (likely via reflection.) An invokedynamic is used to defer resolution until runtime, to avoid inlining the compile-time default into implementing classes (because this could cause brittleness if the default is changed and the interface recompiled but not the implementor.)

The below listing shows pseudo-code for the resolution process embodied in the bootstrap for super.extensionMethod() calls.

```
public void resolveSuper(Class desiredInterface,
                         String methodName, String methodSig) {
        // May need to search entire super-interface tree of desiredInterface
        try {
            Method m = desiredInterface.getMethod(methodName,
toParameterTypes(methodSig));
            if (m != null) {
                Method def = m.getDefaultImplementation();
                if (def != null) {
                    // link call site to def
                }
                else { /* linkage error */ }
            }
            else { /* linkage error */ }
        }
        catch (NoSuchMethodException e) {
            // linkage error
        }
    }
```

8. Implementation strategies

There are several possible implementation strategies, with varying impact on the VM and tools ecosystem. Possible 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* extension methods is rewritten to use an invokedynamic call. (This strategy has been deemed undesirable because it is important that extension methods be invocable via invokevirtual or invokeinterface.)

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.

8.1. Implementation recommendation: load-time weaving with invokedynamic

The implementation approach we recommend is to weave in stubs for extension methods at class load time, but defer resolution until the method is called the first time (by having those stubs lazily perform method resolution using invokedynamic.)

When a class C is loaded, the set of interfaces *directly* implemented by C is examined to see if there are any extension methods. (The reason that interfaces implemented by superclasses need not be scanned is that the superclass will already have stubs for those methods. Therefore we only need to introduce stubs at the points where the method dispatch behavior could possibly have been changed.) For each extension method, a stub implementing that method is added to C, whose body consists of an invokedynamic whose bootstrap does the method resolution and then links the call site. (Once the call site is linked, it will never be invalidated, so there is no need to ever re-run the resolution.)

The resolution process involves first scanning the superclass hierarchy for a (non-stub) implementation, and then scanning the interface hierarchy for the best default if the implementation hierarchy does not find an implementation. This is complicated by the fact that C's superclasses may have stubs woven in when the superclass was loaded, and the resolution process must be able to distinguish between such stubs and methods that were created by the user.

By inserting stubs that implement the extension method directly into the class implementing the interface, from the perspective of other VM functionality (method dispatch, reflection) it is as if the implementor had provided these methods when writing the class. When executing an invokeinterface or invokevirtual on C, callers will see the stub as if it had been present from the outset.

The resolution process must take into account whether there are implementations of the method in C or its superclasses and the complete set of interfaces implemented by C, directly or indirectly. (While concrete implementations should be preferred over defaults, the stubs inserted into superclasses when they were loaded should not be considered a concrete implementation; the stubs must be marked in such a way that the search of the implementation hierarchy not be confused into thinking these methods were originally part of the class. Otherwise we may fail to select the most specific default.)

This approach is chosen because of its overall nonintrusiveness: VM method dispatch and core reflection need not be changed at all; the impact on class loading is not huge since no resolution need take place at load time; and once the class is loaded, performance is quite good as the call sites are linked once and then remain valid thereafter.

In the case where two interfaces provide inconsistent defaults for the same method, the bootstrap should link the call site to a method that throws a linkage error. The linkage error should provide a diagnostic naming the concrete class and the interfaces providing conflicting defaults.

The following is pseudo-code that illustrates the bootstrap process for linking an extension method. The algorithm shown is clearly inefficient.

```
public void resolveDefault(Class implementationClass,
                              String methodName, String methodSig) {
        // First try inheritance hierarchy
        for (Class c=implementationClass;
             c != Object.class;
             c=c.getSuperclass()) {
            try {
                Method m = c.getMethod(methodName,
                                       toParameterTypes(methodSig));
                if (m != null && !isDefenderStub(m)) {
                    // link call site to m
                    return;
                }
            }
            catch (NoSuchMethodException ignored) {
                continue;
            }
        }

        // Now search for a default in the interface hierarchy
        Set<Class> interfaces = new HashSet<Class>();
        accumulateInterfaces(implementationClass, interfaces,
                             methodName, methodSig);
        filterInterfaces(interfaces);
        Method answer = null;
        for (Class candidate : interfaces) {
            Method m = candidate
                    .getMethod(methodName, toParameterTypes(methodSig))
                    .getExtensionMethod();
            if (answer == null)
                answer = m;
            else if (answer != m) {
                // linkage error
                return;
            }
        }
        if (answer != null) {
            // link call site to answer
        }
        else { /* linkage error */ }
    }

    private void accumulateInterfaces(Class c, Set<Class> interfaces,
                                      String methodName, String methodSig) {
        for (Class d : c.getInterfaces()) {
            Method m = d.getMethod(methodName, toParameterTypes(methodSig));
            if (m != null && m.isExtensionMethod())
                interfaces.add(d);
            accumulateInterfaces(d, interfaces, methodName, methodSig);
        }
        Class d = c.getSuperclass();
        if (d != Object.class)
            accumulateInterfaces(d, interfaces, methodName, methodSig);
    }

    private void filterInterfaces(Set<Class> interfaces) {
        for (Iterator<Class> it=interfaces.iterator();
                it.hasNext(); ) {
            Class c = it.next();
            for (Class d : interfaces) {
                if (d != c && c.isAssignableFrom(d))
                    it.remove();
            }
        }
    }
```

8.2. Compatibility and adaptation of default implementation

The signature of the default method must be compatible with that of the extension method. Informally, the argument list of the default will be compatible with the arguments to the extension method with the receiver prepended onto the argument list.

We define compatibility in terms of existing method resolution rules. If the return type of the extension method on interface I is R, the arity *n*, the argument types A_i, and throwing checked exceptions E_j, then the following must be true of the signature of the default:

- A call corresponding to m(I, A_1, A_2, …, A_n) must type-check according to the method resolution rules against the default.
- The return type R of the extension method must be a supertype of the return type of the default (covariant return), or the return type the extension method must be void
- For each checked exception type of the default, it must be a subtype of some checked exception type of the extension method

These rules allow for not only covariant return types and contravariant argument types, but also conversions such as such as widening conversions (int to long), boxing conversions (int to Integer), and even varargs. In the case that conversion is needed, when linking the call site the bootstrap method must identify any needed conversions and use appropriate method handle combinators to adapt the types in the signature of the extension method to the types in the signature of the default.

9. 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.

10. 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.)

11. Binary compatibility

When adding an extension method to an interface, existing client classes compiled against the old version 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.)

If an interface is recompiled to turn an ordinary abstract interface method into an extension method, this should be considered a binary compatible change (this is analogous to turning an abstract method into a non-abstract one.)

Removing the default on an interface method should not be considered a binary compatible change; this is analogous to turning a concrete method into an abstract one.

A default may be added to an ordinary interface method not only through recompilation of the interface, but through extension; if interface A declares ordinary method M, interface B could extend A, override M, and add a default.

12. Unintended consequences for 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 extension 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 3. Possible use of extension methods.

The prevailing wisdom in API design (see Effective Java) is to define types with interfaces and skeletal implementations with companion abstract classes. Extension 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.

13. 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.

14. Restrictions

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

15. Java ME considerations

Java ME applications differ from Java SE applications because they undergo a linking and packaging step that is not performed in Java SE applications, and may not support invokedynamic. In this environment, the packager can directly insert the appropriate stubs into all classes which implement extended interfaces, and there is no need for deferred linking via invokedynamic as a closed-world analysis can determine the correct linkage for each stub.

16. 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.