Interface evolution via “public defender” methods

Brian Goetz, 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; to add closures without extending the Collections classes to take better advantage of closures (e.g., methods like forEach, map, filter, reduce, etc) would be seen as anticlimactic by the community.

Static extension methods have been proposed 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. In general, static-ness is a source of all sorts of problems in Java, so adding more static mechanisms seems like a step in the wrong direction.

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.

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 that of the extension method, with the type of the enclosing interface inserted as the first argument. The syntax of specifying the default method should match that of specifying method references, if method references are to be added to the language.

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. Implementation – compilation of extended interfaces

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 bit (ACC_EXTENDED_INTERFACE) or 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 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;
}
```

4. Implementation – compilation of clients

When the static compiler encounters a non-abstract class that implements an extended interface, and the class does not implement all of the defender methods, for each missing defender method the compiler inserts bridge methods that simply forward the arguments to the default implementation with an invokestatic. The class file emitted by the compiler implements all methods in the interface, including the defender methods, so no further work is needed by the VM to deal with classes that implement extended interfaces and have been compiled by the JDK 7+ compiler.

4.1. Referring to the default implementation

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 allow the simpler syntax "super.methodName()".

4.2. Resolving ambiguity

If the class implements multiple extended interfaces, both of which provide a default implementation for the same method name and signature, the implementing class MUST provide an implementation. (This is expected to be a quite unusual case.) The existence of the InterfaceName.super.methodName() syntax makes it easy for the class to choose one of the defaults. (We might choose to relax this restriction if both interfaces provide the same default.)

4.3. Brittleness considerations

Since the name of the default implementation is injected into the implementing class at compilation time, if the interface is modified to provide a different default and later recompiled, previously compiled implementing classes will continue to use the old default. (This is not unlike the behavior of inlining static final fields into classes that reference them.)

This sort of "brittle client" problem already exists in the language, and may be acceptable. If it is not acceptable, we may want to have the generated forwarding methods invoke the default through an invokedynamic call instead of an invokestatic, and provide a bootstrap method to dynamically link the call site the first time to the default actually specified in the loaded interface class file.

5. Implementation – VM

Classes compiled by the JDK 7 compiler will not need any special treatment, since they will have implementations of all the interface methods (some of which may be compiler-generated bridge methods.) However, the VM may wish to load class files compiled by older compilers, which under the current VM behavior would fail to load because they do not implement all the methods of the interface they claim to implement.

The VM behavior would be extended to mimic the behavior of the static compiler at class load time – if the VM attempts to load a legacy class that implements an extended interface and not all defender methods have implementations, the VM would inject bridge methods into that class that invoke the default implementation. In this way, existing implementations of extended interfaces would continue to work as if the default implementation had been provided in the source code.

5.1. Resolving ambiguity

In the unlikely case that the VM attempts to load a class that implements multiple extended interfaces that have a common defender method but with different defaults, the VM should not attempt to resolve the ambiguity automatically. Instead, it should generate a bridge method that throws UOE or similar exception. We expect this situation to be rare.

5.2. VM implementation strategy

It is desirable that this class modification behavior be implemented as far to the periphery of the VM 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.

6. 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.

7. 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.

8. 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.

9. Restrictions

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

10. 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, 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 us with a vehicle for solving other similar migration problems in the future.