

Interface evolution via virtual extension methods

Brian Goetz

Fourth draft, June 2011

1. Problem statement

Once published, it is impossible to add methods to an interface without breaking existing implementations. (Specifically, adding a method to an interface is not a *source-compatible* change.) 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 feature¹.

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 extension², and they are not reflectively discoverable.

2. Virtual extension methods³

In this document, we outline a mechanism for adding new methods to existing interfaces, which we call *virtual extension methods*. Existing interfaces can be augmented without compromising backward compatibility⁴ 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 a method body. A key characteristic of extension methods is that they are *virtual methods* just like other interface methods, but provide a default implementation in the event that the implementing class does not provide a method body. 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();
```

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.

³ In previous drafts these were also called *defender methods*.

⁴ The exact details are both binary and source compatibility are already complicated today, and the compatibility considerations for extension methods are no exception. In general we aim to make strong binary compatibility guarantees and slightly weaker source compatibility guarantees.

```

// The rest of the existing Set methods

public void forEach(Block<T> b)
    default Collections.<T>setForEach;
}

class Collections {
    ...
    public static<T> void setForEach(Set<T> set,
                                    Block<T> block) {...}
}

```

Listing 1. Example virtual extension method.

The declaration of `forEach()` tells us that this is an extension method, because it has a “default” clause⁵. The default clause names the default implementation, which is a static method. The signature of the default implementation must be compatible with that of the extension method, with the type of the enclosing interface inserted as the first argument⁶. It is a compile-time error if the default implementation cannot be found, or the signature is incompatible with the interface method.

Implementations of `Set` are free to provide their own implementation of `forEach()`, since it is a virtual method just like any other interface method. If they do not, the default implementation will be used instead. An interface that has one or more extension methods is called an *extended interface*.

From the perspective of a client of the interface, extension methods are no different from ordinary interface methods. They are invoked via `invokeinterface`, they can be discovered and invoked reflectively, and the client does not care where the implementation comes from.

2.1. Compatibility and adaptation of default implementation

The signature of the default implementation 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 $m(I, A_1, A_2, \dots, A_n)$ must resolve according to the method resolution rules against the default.

⁵ It is a deliberate choice to not simply allow the programmer to include the code of the default inline within the interface. Among other reasons, it would violate the long-held rule that “interfaces don’t have code”.

⁶ The syntax of specifying the default implementation should match that of specifying method references, if method references are to be added to the language.

- 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 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.

2.2. Accessibility

The static method referenced by the default clause must be accessible to the extended interface, but need not be public.

3. Method dispatch

The addition of extension methods adds an extra phase to the runtime method dispatch algorithm. In Java SE 7, for a method invocation on a receiver of type D, first D is searched for a matching non-abstract method, then its superclass, and so on until we reach Object, and if an implementation is not found, a linkage exception is thrown. To support extension methods, we add another step before we throw the linkage exception: we search D and its superclasses for a matching method (abstract or not), and if we do not find an implementation of the method in the implementation hierarchy, we then search for a unique mostspecific interface providing a default for the method among D's interfaces. If we cannot find a unique most specific default, we throw a linkage exception (which may indicate no default found, or may indicate multiple competing defaults found.)

Method resolution supporting extension methods is as follows:

1. First search for a matching method in the superclass hierarchy, from receiver class proceeding upward through superclasses to Object. If a non-abstract method is found, the search is resolved successfully. If an abstract method is found, the search fails and extension methods are *not* considered in the resolution.
2. Construct the list of interfaces implemented by D, directly or indirectly, which contain a matching extension method.
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. If the resulting list of interfaces contains a single interface, the search is resolved successfully; the default implementation named in that interface is the resolution. 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.

The description here is only intended to be a rough description of how method resolution is performed. The exact resolution algorithm is modeled formally in the paper *Featherweight Defenders*.

3.1. Resolution examples

In the following example, we have a diamond-shaped hierarchy:

```
interface A { void m() default X.a; }
interface B extends A { }
interface C extends A { }
class D implements B, C { }
```

In this example, even though D inherits m() from both B and C, when we construct the list of interfaces that provide a default for m(), we see there is only one – A. So m() in D resolves to X.a.

One of B or C (but not both) could still override m() without causing resolution to fail:

```
interface A { void m() default X.a; }
interface B extends A { void m() default X.b; }
interface C extends A { }
class D implements B, C { }
```

Here, when we construct the list of interfaces providing a default (step 2), we get A and B, but then in step 3 we remove A because the default provided by B is more specific. We then get a unique most-specific provider of default – B – and D.m() resolves to X.b.

If both B and C were to override m():

```
interface A { void m() default X.a; }
interface B extends A { void m() default X.b; }
interface C extends A { void m() default X.b; }
class D implements B, C { }
```

In this case, the inheritance of m() in D would be ambiguous (because the set of most-specific default-providing interfaces contains both B and C). At compile time D would be rejected unless D implements a body for m().

3.2. Abstract methods

In searching the superclass hierarchy, we prefer a declaration in a superclass to a default in an interface. This preference includes abstract methods in superclasses as well; the defaults are only considered when the entire implementation hierarchy is silent on the status of the method in question. (This provides us with a simple rule: “the superclass always wins.”)

In the following example, m() is considered abstract in D even though a default is available from A, because the abstract declaration in C takes precedence over the default:

```
interface A { void m() default X.a; }
class C { abstract void m(); }
class D extends C implements A { }
```

3.3. Conflicting defaults

Step (3) in the resolution procedure above addresses the case of ambiguous default implementations. It is allowable to inherit a default through multiple paths (such as through a diamond-shaped interface hierarchy), but the resolution procedure is looking for a *unique, most specific default-providing interface*. If one default “shadows” another (where a subinterface provides a different default for an extension method declared in a superinterface), then the less specific interface is pruned from consideration – no matter where it appears in the inheritance hierarchy. If two or more extended interfaces provide default implementations, and one is not a superinterface of the other, then neither is used and a linkage exception is thrown indicating conflicting default implementations.⁷

3.4. Covariant overrides

Interface methods can be overridden with covariant return types (where the more specific interface provides a more specific return type). If the interface method being overridden has a default, the subinterface *must* either provide a new default or remove the default (the old default is not guaranteed to be type-safe with respect to the more specific return type.)

3.5. “Pass-through” of defaults

In the following case:

```
interface A { void m() default X.a; }  
interface B extends A { void m(); }
```

The declaration of B does *not* have the effect of removing the default from A. From the perspective of resolving the default for m(), this declaration of B is identical to one that does not declare m() at all.

3.6. Removal of defaults

There may be situations in which an interface method overrides a method with a default, and it is more desirable to remove the default than to provide a new default (in that case the subinterface could specify “default none”.) For purposes of identifying the most specific default-providing interface, an interface specifying “default none” is still considered to provide a default (just as we treat both method bodies and abstract methods in classes identically for purposes of determining whether to search for a default.) If the compiler determines that the unique most-specific default-providing interface specifies no default implementation, it will require implementing classes to provide a method body just as if the method were an ordinary abstract interface method.

For purposes of extension method resolution, extension methods with their defaults removed are *not* treated the same as abstract interface methods. Effectively, there are three states an interface method can be in with respect to defaults: an ordinary abstract interface method, an extension method with a default, and an extension method with its default removed. Once a method has been given a default, that default can be removed, but there is no way to return it to being an ordinary abstract interface method. (Once an

⁷Note that conflicting defaults can only arise from inconsistent separate compilation; the compiler will reject conflicting defaults if they are present at compile time.

extension method, always an extension method.) In the following example, A declares a default for m(), and B overrides m() and removes the default. In both D and E, m() is considered abstract and must provide an implementation; class F fails to compile because there is not a unique most-specific default-providing interface (both B and C are competing to provide the default in F).

```
interface A { void m() default X.a; }
interface B extends A { void m() default none; }
interface C extends A { void m() default X.c; }
class D implements B { }
class E implements A,B { }
class F implements B,C { }
```

It is a compile-time error if an interface method specifies “default none” and none of its superinterfaces declare a default for that method.

3.7. Super-calls

A concrete class method or interface extension method may wish to override an extension method, but still refer to the inherited default from one of its superinterfaces, say to choose between conflicting defaults methods, or to decorate the call to the default from the interface.

```
interface A { void m() default X.a; }
interface B { void m() default X.b; }
class C implements A, B {
    public void m() { A.super.m(); }
}
class D implements A {
    public void m() { println("Calling m"); A.super.m(); }
}
```

An implementation of an extension method in a class can refer to the default implementation from a superinterface 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()”. The use of this mechanism is restricted to the contexts where one can use the super.m() construct in Java SE 7. The interfaces through which you can call up to superinterface defaults is restricted to the immediate superinterfaces of the class.

Further, because it is a compile time error for an interface to have conflicting defaults, the default clause in an extension method declaration also may need to refer to the default in one of its superinterfaces:

```
interface A { void m() default X.a; }
interface B { void m() default X.b; }
interface Q extends A, B {
    public void m() default B.super.m;
}
```

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 will likely choose the latter route, and 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.) We could also include setting an additional accessibility bit (`ACC_DEFENDER`) in the `access_flags` field of the `method_info` structure, but will likely not for reasons of flag real estate management.

```
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 implementation exists in another class than the specifying interface, the `getDefaultImplementation()` method may trigger loading of the class hosting the default implementation.

6. Compile-time vs runtime

The compile-time type checking rules are strict, with the goal that a program that compiles globally (i.e., compilation succeeds when all classes are compiled together) will always link properly. However, it is possible, through separate compilation, to create programs that cannot link (such as recompiling an interface to remove a default, or adding a default that would make default resolution ambiguous.) The *Featherweight Defenders* paper separately specifies the compile-time vs run-time semantics.

We may choose to relax the runtime semantics to make a program more robust against conflicts introduced by inconsistent separate compilation, since this hazard can come about quite innocently when a class implements interfaces from two separately maintained libraries. There are a variety of schemes that might increase the set of changes to supertypes that could be considered binary or source compatible (though they also increase complexity).

7. Invocation of extension methods

Extension methods should be invoked like any other virtual method, with `invokevirtual` or `invokeinterface`⁸. 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`⁹. 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.

8. Implementation strategies

There are several possible implementation strategies, with varying impact on the VM and tools ecosystem. The two most credible strategies are:

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 synthetic method is woven in that invokes the appropriate default implementation. In the general case this is done by the class loader at execution time, but can also be done before execution when a module is loaded into a module repository.
2. Modify the behavior of `invokevirtual` / `invokeinterface` in the JVM to implement the invocation semantics described here. (This approach is more invasive, but is more in keeping with the spirit of dynamic linking and lazy resolution.)

⁸ One translation strategy that was initially considered but rejected was to have calls to extension methods translated as `invokedynamic`. However, this would make adding or removing a default a binary-incompatible change in all cases, and would have placed a burden on classfile consumers (such as language compilers) to keep track of yet another form of method invocation.

⁹ 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.

8.1. Implementation recommendation: lazy vtable building

The implementation approach we recommend is to add in default resolution to the standard method resolution in `invokevirtual` / `invokedynamic`, deferring resolution to the first time a given method is invoked on an instance of a given class.

Resolution requires computing the set of superinterfaces that provide (or remove) a default for the method being resolved. This can be done on each resolution request, or (as a time-space tradeoff) we can associate (perhaps via a `ClassValue`) summarized default resolution data with each interface so that we need only examine the immediate superinterfaces of a class when resolving a method in that class.

8.2. Bridge methods

A significant complication for resolution of extension methods is *bridge methods*. Bridge methods are synthetic methods inserted by the compiler to make up for differences between the compile-time (generic) type system and the run-time (erased) type system. There are two significant cases where bridge methods are relevant to extension methods: covariant overrides and generic substitution.

The following example shows a covariant override:

```
interface A { public Object make(); }
interface B extends A { public String make(); }
```

At the language level, a class that implements B has a single method `make()`, which returns `String`. But at the VM level, it must respond to invocations for both of the following method signatures, which from the VM's perspective are two completely different methods:

```
make()Ljava/lang/Object;
make()Ljava/lang/String;
```

The compiler helps us out by ensuring that any class implementing B and providing a body for the `String`-bearing version of `make()` also gets a synthetic implementation (bridge method) for the `Object`-bearing version of `make()`, which just turns around and calls the `String`-bearing version.

This is necessary because if someone casts the B to an A, and invokes `make()` on that, the signature from A will be used to resolve the method, and the object had better respond to that invocation.

The other condition where bridge methods arise is generic substitution. Consider:

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

Here, the developer will provide a method body for `foo(String)`. But, the signature of the (erasure of the) method in B is `foo(Ljava/lang/Object;)`. Again, the compiler has to provide a bridge method.

The compiler only generates bridge methods where a concrete method implementation is present. This means that when we resolve extension methods in a class C, the VM must be prepared to resolve bridges for those extension methods too (and figure out that the `Object`-bearing `make()` is a bridge for the `String`-bearing `make()`). This introduces

additional complexity as the determination of which methods require bridging to which other methods may require analysis of generic type signatures. While there are cases where the static compiler can embed additional information in the classfile to accelerate this process, these cases do not account for 100% of the time where bridge methods are required.¹⁰

8.3. Implementation strategy for super-calls

Calls to “super” defaults will be translated using `invokedynamic`, where the method name and argument list encodes the desired interface, method name, and method signature. The bootstrap method will be a language runtime method in `java.lang.invoke`, which determines which default is being called, and permanently links the call site using a `ConstantCallSite`.

9. Compatibility considerations

There are two primary types of compatibility issues we care about: binary compatibility and source compatibility. To say a change is *binary compatible* means that a classfile in a program could be replaced and all call sites within the program that linked before the change will continue to link after the change. To say a change is *source compatible* means that if the entire program compiled before the change, then the entire program should compile after the change.

As the Java language as it now stands, a number of operations on supertypes could introduce compatibility issues:

- Add concrete method to superclass. This is binary-compatible, but (in rare cases) source-incompatible (there could be a return-type conflict with another method of the same name in a subclass.)
- Add method to interface. Binary compatible, but source-incompatible.
- Change concrete method implementation. Binary and source compatible.
- Remove method from superclass. Binary and source incompatible.
- Remove method from interface. Binary and source incompatible.

With extension methods in the language, we must consider some new operations:

- Add method to interface with default (add-defmeth).
- Add default to existing interface method (add-def).
- Change default on existing extension method (mod-def).
- Remove a default but leave the method (rem-def).
- Remove a method and its default (rem-defmeth.)

¹⁰ One might argue that it would have been better to add bridge method handling as a VM feature rather than a compiler feature. In 2004 having the static compiler generate bridges seemed like a pretty darn clever idea. Today’s problems come from yesterday’s solutions.

9.1. Source compatibility

Under the semantics currently defined, only mod-def is strictly source-compatible, though *most* instances of add-defmeth and add-def are likely to not cause source compatibility issues in practice. The situations where add-defmeth and add-def could cause problems are those where the addition of the method or the default would cause an extension method which properly resolved in a subclass prior to the change to become ambiguous, or those which introduce a signature incompatibility with another method of the same name. These are roughly analogous to the cases where adding a concrete superclass method would be incompatible. (We are investigating mechanisms of mitigating this problem.)

9.2. Binary compatibility

Under the semantics currently defined, only mod-def is strictly binary-compatible, though *most* instances of add-defmeth and add-def are likely to not cause binary compatibility issues in practice. The situations where add-defmeth and add-def could cause problems are those where the addition of the method or the default would cause an extension method which properly resolved in a subclass prior to the change to become ambiguous.

Note that any program which fails to link after such a change would also fail to compile, which means that the changes which are binary incompatible are also source incompatible. Conversely, if a program can be globally compiled successfully, then all invocations of extension methods are guaranteed to link correctly. Therefore, binary compatibilities only happen in the case of programs that have been “broken” by separate compilation.

There are three categories of mechanisms we might use to increase binary compatibility for broken programs (at the cost of increased complexity):

- Structural approaches, such as imposing a linearization ordering on interfaces that is used to break ambiguities;
- Historical approaches, such as remembering things in the classfile about default resolution (such as the identity of the default, or the interface from which it is inherited, or the inheritance path to that interface) that could be used to disambiguate the default;
- Call-site-based approaches, such as using the interface name at an invokeinterface site to identify a “preferred” inheritance path for an ambiguous default.

None of these are perfect solutions; each reduces the set of binary-incompatible changes in a different way. The only value these approaches have is to render implementation classes more robust to separately compiled changes to superinterfaces (such as changes to interfaces from libraries that an application class implements.)

10. 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> {
    public int size()
        default AbstractSetMethods.size;

    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.

11. 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.

12. Restrictions

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

13. 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`. However, the packager can perform a closed-world calculation of all extension methods and their bridges, and directly weave in stubs to classes implementing extended interfaces, eliminating the need for runtime resolution. Similarly, for super-calls, the packager can recognize the `invokedynamic` calls (by their known bootstrap) and replace these with static invocations of the appropriate default implementation.