CHAPTER 13

Binary Compatibility

Despite all of its promise, software reuse in object-oriented programming has yet to reach its full potential. A major impediment to reuse is the inability to evolve a compiled class library without abandoning the support for already compiled applications. . . . [A]n object-oriented model must be carefully designed so that class-library transformations that should not break already compiled applications, indeed, do not break such applications.
—Ira Forman, Michael Conner, Scott Danforth, and Larry Raper, Release-to-Release Binary Compatibility in SOM (1995)

Development tools for the Java programming language should support automatic recompilation as necessary whenever source code is available. Particular implementations may also store the source and binary of types in a versioning database and implement a `ClassLoader` that uses integrity mechanisms of the database to prevent linkage errors by providing binary-compatible versions of types to clients.

Developers of packages and classes that are to be widely distributed face a different set of problems. In the Internet, which is our favorite example of a widely distributed system, it is often impractical or impossible to automatically recompile the pre-existing binaries that directly or indirectly depend on a type that is to be changed. Instead, this specification defines a set of changes that developers are permitted to make to a package or to a class or interface type while preserving (not breaking) compatibility with existing binaries.

The paper quoted above appears in *Proceedings of OOPSLA ’95*, published as *ACM SIGPLAN Notices*, Volume 30, Number 10, October 1995, pages 426–438. Within the framework of that paper, Java programming language binaries are binary compatible under all relevant transformations that the authors identify (with some caveats with respect to the addition of instance variables). Using their scheme, here is a list of some important binary compatible changes that the Java programming language supports:

- Reimplementing existing methods, constructors, and initializers to improve performance.
- Changing methods or constructors to return values on inputs for which they previously either threw exceptions that normally should not occur or failed by going into an infinite loop or causing a deadlock.
- Adding new fields, methods, or constructors to an existing class or interface.
- Deleting `private` fields, methods, or constructors of a class.
- When an entire package is updated, deleting default (package-only) access fields, methods, or constructors of classes and interfaces in the package.
- Reordering the fields, methods, or constructors in an existing type declaration.
- Moving a method upward in the class hierarchy.
- Reordering the list of direct superinterfaces of a class or interface.
- Inserting new class or interface types in the type hierarchy.

This chapter specifies minimum standards for binary compatibility guaranteed by all implementations. The Java programming language guarantees compatibility when binaries of classes and interfaces are mixed that are not known to be from compatible sources, but whose sources have been modified in the compatible ways described here. Note that we are discussing compatibility between releases of an application. A discussion of compatibility among releases of the Java platform is beyond the scope of this chapter.

We encourage development systems to provide facilities that alert developers to the impact of changes on pre-existing binaries that cannot be recompiled.

This chapter first specifies some properties that any binary format for the Java programming language must have (§13.1). It next defines binary compatibility, explaining what it is and what it is not (§13.2). It finally enumerates a large set of possible changes to packages (§13.3), classes (§13.4) and interfaces (§13.5), specifying which of these changes are guaranteed to preserve binary compatibility and which are not.

13.1 The Form of a Binary

Programs must be compiled either into the `class` file format specified by the *The Java™ Virtual Machine Specification, Second Edition*, or into a representation that can be mapped into that format by a class loader written in the Java programming language. Furthermore, the resulting `class` file must have certain properties. A number of these properties are specifically chosen to support source code transformations that preserve binary compatibility.

The required properties are:

- The class or interface must be named by its *binary name*, which must meet the following constraints:
  - The binary name of a top-level type is its canonical name (§6.7).
  - The binary name of a member type consists of the binary name of its immediately enclosing type, followed by $, followed by the simple name of the member.
  - The binary name of a local class (§14.3) consists of the binary name of its immediately enclosing type, followed by $, followed by a non-empty sequence of digits, followed by the simple name of the local class.
- The binary name of an anonymous class (§15.9.5) consists of the binary name of its immediately enclosing type, followed by $, followed by a non-empty sequence of digits. A reference to another class or interface type must be symbolic, using the binary name of the type.
- Given a legal expression denoting a field access in a class *C*, referencing a non-constant (§13.4.9) field named *f* declared in a (possibly distinct) class or interface *D*, we define the *qualifying type of the field reference* as follows:
  - If the expression is of the form *Primary.f* then:
    - If the compile-time type of *Primary* is an intersection type (§4.9) *V1 & ... & Vn*, then the qualifying type of the reference is *V1*.
    - Otherwise, the compile-time type of *Primary* is the qualifying type of the reference.
  - If the expression is of the form `super.`*f* then the superclass of *C* is the qualifying type of the reference.
  - If the expression is of the form *X*`.super.`*f* then the superclass of *X* is the qualifying type of the reference.
  - If the reference is of the form *X.f*, where *X* denotes a class or interface, then the class or interface denoted by *X* is the qualifying type of the reference
  - If the expression is referenced by a simple name, then if *f* is a member of the current class or interface, *C*, then let *T* be *C*. Otherwise, let *T* be the

innermost lexically enclosing class of which `f` is a member. `T` is the qualifying type of the reference.

The reference to `f` must be compiled into a symbolic reference to the erasure of the qualifying type of the reference, plus the simple name of the field, `f`. The reference must also include a symbolic reference to the declared type of the field so that the verifier can check that the type is as expected.

- References to fields that are constant variables (§4.12.4) are resolved at compile time to the constant value that is denoted. No reference to such a constant field should be present in the code in a binary file (except in the class or interface containing the constant field, which will have code to initialize it), and such constant fields must always appear to have been initialized; the default initial value for the type of such a field must never be observed. See §13.4.8 for a discussion.
- Given a method invocation expression in a class or interface `C` referencing a method named `m` declared in a (possibly distinct) class or interface `D`, we define the *qualifying type of the method invocation* as follows:

  If `D` is `Object` then the qualifying type of the expression is `Object`. Otherwise:

  - If the expression is of the form *Primary.m* then:
    - If the compile-time type of *Primary* is an intersection type (§4.9) *V1 & ... & Vn*, then the qualifying type of the method invocation is *V1*.
    - Otherwise, the compile-time type of *Primary* is the qualifying type of the method invocation.
  - If the expression is of the form `super.m` then the superclass of `C` is the qualifying type of the method invocation.
  - If the expression is of the form `X.super.m` then the superclass of `X` is the qualifying type of the method invocation.
  - If the reference is of the form `X.m`, where `X` denotes a class or interface, then the class or interface denoted by `X` is the qualifying type of the method invocation
  - If the method is referenced by a simple name, then if `m` is a member of the current class or interface, `C`, let `T` be `C`. Otherwise, let `T` be the innermost lexically enclosing class of which `m` is a member. `T` is the qualifying type of the method invocation.

A reference to a method must be resolved at compile time to a symbolic reference to the erasure of the qualifying type of the invocation, plus the erasure of the signature of the method (§8.4.2). A reference to a method must also include either a symbolic reference to the return type of the denoted method or an indication that the denoted method is declared `void` and does not return a value. The signature of a method must include all of the following:

- The simple name of the method
- The number of parameters to the method
- A symbolic reference to the type of each parameter

- Given a class instance creation expression (§15.9) or a constructor invocation statement (§8.8.5.1) in a class or interface *C* referencing a constructor *m* declared in a (possibly distinct) class or interface *D*, we define the *qualifying type of the constructor invocation* as follows:
  - If the expression is of the form `new` *D*(...) or *X*.`new` *D*(...), then the qualifying type of the invocation is *D*.
  - If the expression is of the form `new` *D*(..){...} or *X*.`new` *D*(...){...}, then the qualifying type of the expression is the compile-time type of the expression.
  - If the expression is of the form `super`(...) or *Primary*.`super`(...) then the qualifying type of the expression is the direct superclass of *C*.
  - If the expression is of the form `this`(...), then the qualifying type of the expression is *C*.

  A reference to a constructor must be resolved at compile time to a symbolic reference to the qualifying type of the invocation, plus the signature of the constructor (§8.8.2). The signature of a constructor must include both:

  - The number of parameters to the constructor
  - A symbolic reference to the type of each parameter

  In addition the constructor of a non-private inner member class must be compiled such that it has as its first parameter, an additional implicit parameter representing the immediately enclosing instance (§8.1.2).

- Any constructs introduced by the compiler that do not have a corresponding construct in the source code must be marked as synthetic, except for default constructors and the class initialization method.

A binary representation for a class or interface must also contain all of the following:

- If it is a class and is not class `Object`, then a symbolic reference to the direct superclass of this class
- A symbolic reference to each direct superinterface, if any
- A specification of each field declared in the class or interface, given as the simple name of the field and a symbolic reference to the type of the field
- If it is a class, then the signature of each constructor, as described above
- For each method declared in the class or interface, its signature and return type, as described above
- The code needed to implement the class or interface:
  - For an interface, code for the field initializers
  - For a class, code for the field initializers, the instance and static initializers, and the implementation of each method or constructor
- Every type must contain sufficient information to recover its canonical name (§6.7).
- Every member type must have sufficient information to recover its source level access modifier.
- Every nested class must have a symbolic reference to its immediately enclosing class.
- Every class that contains a nested class must contain symbolic references to all of its member classes, and to all local and anonymous classes that appear in its methods, constructors and static or instance initializers.

The following sections discuss changes that may be made to class and interface type declarations without breaking compatibility with pre-existing binaries. Under the translation requirements given above, the Java virtual machine and its `class` file format support these changes. Any other valid binary format, such as a compressed or encrypted representation that is mapped back into class files by a class loader under the above requirements will necessarily support these changes as well.

13.2 What Binary Compatibility Is and Is Not

A change to a type is *binary compatible with* (equivalently, does not *break binary compatibility* with) preexisting binaries if preexisting binaries that previously linked without error will continue to link without error.

Binaries are compiled to rely on the accessible members and constructors of other classes and interfaces. To preserve binary compatibility, a class or interface should treat its accessible members and constructors, their existence and behavior, as a *contract* with its users.

The Java programming language is designed to prevent additions to contracts and accidental name collisions from breaking binary compatibility; specifically:

- Addition of more methods overloading a particular method name does not break compatibility with preexisting binaries. The method signature that the preexisting binary will use for method lookup is chosen by the method overload resolution algorithm at compile time (§15.12.2). (If the language had been designed so that the particular method to be executed was chosen at run time, then such an ambiguity might be detected at run time. Such a rule would imply that adding an additional overloaded method so as to make ambiguity possible at a call site could break compatibility with an unknown number of preexisting binaries. See §13.4.23 for more discussion.)

Binary compatibility is not the same as source compatibility. In particular, the example in §13.4.6 shows that a set of compatible binaries can be produced from sources that will not compile all together. This example is typical: a new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the preexisting binary for that unchanged part of the source code retains the fully-qualified, previous meaning of the name. Producing a consistent set of source code requires providing a qualified name or field access expression corresponding to the previous meaning.

13.3 Evolution of Packages

A new top-level class or interface type may be added to a package without breaking compatibility with pre-existing binaries, provided the new type does not reuse a name previously given to an unrelated type. If a new type reuses a name previously given to an unrelated type, then a conflict may result, since binaries for both types could not be loaded by the same class loader.

Changes in top-level class and interface types that are not `public` and that are not a superclass or superinterface, respectively, of a `public` type, affect only types within the package in which they are declared. Such types may be deleted or otherwise changed, even if incompatibilities are otherwise described here, provided that the affected binaries of that package are updated together.

13.4 Evolution of Classes

This section describes the effects of changes to the declaration of a class and its members and constructors on pre-existing binaries.

13.4.1 `abstract` Classes

If a class that was not `abstract` is changed to be declared `abstract`, then pre-existing binaries that attempt to create new instances of that class will throw either an `InstantiationError` at link time, or (if a reflective method is used) an `InstantiationException` at run time; such a change is therefore not recommended for widely distributed classes.

Changing a class that was declared `abstract` to no longer be declared `abstract` does not break compatibility with pre-existing binaries.

13.4.2 `final` Classes

If a class that was not declared `final` is changed to be declared `final`, then a `VerifyError` is thrown if a binary of a pre-existing subclass of this class is loaded, because `final` classes can have no subclasses; such a change is not recommended for widely distributed classes.

Changing a class that was declared `final` to no longer be declared `final` does not break compatibility with pre-existing binaries.

13.4.3 `public` Classes

Changing a class that was not declared `public` to be declared `public` does not break compatibility with pre-existing binaries.

If a class that was declared `public` is changed to not be declared `public`, then an `IllegalAccessError` is thrown if a pre-existing binary is linked that needs but no longer has access to the class type; such a change is not recommended for widely distributed classes.

13.4.4 Superclasses and Superinterfaces

A `ClassCircularityError` is thrown at load time if a class would be a superclass of itself. Changes to the class hierarchy that could result in such a circularity when newly compiled binaries are loaded with pre-existing binaries are not recommended for widely distributed classes.

Changing the direct superclass or the set of direct superinterfaces of a class type will not break compatibility with pre-existing binaries, provided that the total set of superclasses or superinterfaces, respectively, of the class type loses no members.

If a change to the direct superclass or the set of direct superinterfaces results in any class or interface no longer being a superclass or superinterface, respectively, then link-time errors may result if pre-existing binaries are loaded with the binary of the modified class. Such changes are not recommended for widely distributed classes.

For example, suppose that the following test program:

```
class Hyper { char h = 'h'; }
class Super extends Hyper { char s = 's'; }
class Test extends Super {
    public static void printH(Hyper h) {
        System.out.println(h.h);
        }
     public static void main(String[] args) {
    printH(new Super());
     }
}
```

is compiled and executed, producing the output:

```
h
```

Suppose that a new version of class `Super` is then compiled:

```
class Super { char s = 's'; }
```

This version of class `Super` is not a subclass of `Hyper`. If we then run the existing binaries of `Hyper` and `Test` with the new version of `Super`, then a `VerifyError` is thrown at link time. The verifier objects because the result of `new Super()` cannot be passed as an argument in place of a formal parameter of type `Hyper`, because `Super` is not a subclass of `Hyper`.

It is instructive to consider what might happen without the verification step: the program might run and print:

```
s
```

This demonstrates that without the verifier the type system could be defeated by linking inconsistent binary files, even though each was produced by a correct Java compiler.

The lesson is that an implementation that lacks a verifier or fails to use it will not maintain type safety and is, therefore, not a valid implementation.

13.4.5 Class Formal Type Parameters

Renaming a type variable (§4.4) declared as a formal type parameter of a class has no effect with respect to pre-existing binaries. Adding or removing a type parameter does not, in itself, have any implications for binary compatibility.

DISCUSSION

Note that if such type variables are used in the type of a field or method, that may have the normal implications of changing the aforementioned type.

Changing the first bound of a type parameter will change the erasure (§4.6) of any member that uses that type variable in its own type, and this may effect binary compatibility. Changing any other bound has no effect on binary compatibility.

13.4.6 Class Body and Member Declarations

No incompatibility with pre-existing binaries is caused by adding an instance (respectively `static`) member that has the same name, accessibility, (for fields) or same name, accessibility, signature, and return type (for methods) as an instance (respectively `static`) member of a superclass or subclass. No error occurs even if the set of classes being linked would encounter a compile-time error.

Deleting a class member or constructor that is not declared `private` may cause a linkage error if the member or constructor is used by a pre-existing binary.

If the program:

```
class Hyper {
   void hello() { System.out.println("hello from Hyper"); }
}

class Super extends Hyper {
   void hello() { System.out.println("hello from Super"); }
}

class Test {
   public static void main(String[] args) {
      new Super().hello();
```

```
        }
    }
```

is compiled and executed, it produces the output:

```
    hello from Super
```

Suppose that a new version of class `Super` is produced:

```
    class Super extends Hyper { }
```

then recompiling `Super` and executing this new binary with the original binaries for `Test` and `Hyper` produces the output:

```
    hello from Hyper
```

as expected.

The `super` keyword can be used to access a method declared in a superclass, bypassing any methods declared in the current class. The expression:

`super.`*Identifier*

is resolved, at compile time, to a method *`M`* in the superclass *`S`*. If the method *`M`* is an instance method, then the method *`MR`* invoked at run time is the method with the same signature as *`M`* that is a member of the direct superclass of the class containing the expression involving `super`. Thus, if the program:

```
    class Hyper {
        void hello() { System.out.println("hello from Hyper"); }
    }

    class Super extends Hyper { }
    class Test extends Super {
        public static void main(String[] args) {
            new Test().hello();
        }

        void hello() {
            super.hello();
        }

    }
```

is compiled and executed, it produces the output:

```
    hello from Hyper
```

Suppose that a new version of class `Super` is produced:

```
    class Super extends Hyper {
        void hello() { System.out.println("hello from Super"); }
    }
```

If `Super` and `Hyper` are recompiled but not `Test`, then running the new binaries with the existing binary of `Test` produces the output:

```
    hello from Super
```

as you might expect. (A flaw in some early implementations caused them to print:

```
    hello from Hyper
```

incorrectly.)

13.4.7 Access to Members and Constructors

Changing the declared access of a member or constructor to permit less access may break compatibility with pre-existing binaries, causing a linkage error to be thrown when these binaries are resolved. Less access is permitted if the access modifier is changed from default access to `private` access; from `protected` access to default or `private` access; or from `public` access to `protected`, default, or `private` access. Changing a member or constructor to permit less access is therefore not recommended for widely distributed classes.

Perhaps surprisingly, the binary format is defined so that changing a member or constructor to be more accessible does not cause a linkage error when a subclass (already) defines a method to have less access.

So, for example, if the package `points` defines the class `Point`:

```
package points;
public class Point {
    public int x, y;
    protected void print() {
        System.out.println("(" + x + "," + y + ")");
    }
}
```

used by the `Test` program:

```
class Test extends points.Point {
            protected      void      print()      {      Sys-
tem.out.println("Test"); }
    public static void main(String[] args) {
        Test t = new Test();
        t.print();
    }
}
```

then these classes compile and `Test` executes to produce the output:

```
Test
```

If the method `print` in class `Point` is changed to be `public`, and then only the `Point` class is recompiled, and then executed with the previously existing binary for `Test` then no linkage error occurs, even though it is improper, at compile time, for a `public` method to be overridden by a `protected` method (as shown by the fact that the class `Test` could not be recompiled using this new `Point` class unless print were changed to be `public`.)

Allowing superclasses to change `protected` methods to be `public` without breaking binaries of preexisting subclasses helps make binaries less fragile. The alternative, where such a change would cause a linkage error, would create additional binary incompatibilities.

13.4.8 Field Declarations

Widely distributed programs should not expose any fields to their clients. Apart from the binary compatibility issues discussed below, this is generally good software engineering practice. Adding a field to a class may break compatibility with pre-existing binaries that are not recompiled.

Assume a reference to a field *f* with qualifying type *T*. Assume further that *f* is in fact an instance (respectively `static`) field declared in a superclass of *T*, *S*, and that the type of *f* is *X*. If a new field of type *X* with the same name as *f* is added to a subclass of *S* that is a superclass of *T* or *T* itself, then a linkage error may occur. Such a linkage error will occur only if, in addition to the above, either one of the following conditions hold:

- The new field is less accessible than the old one.
- The new field is a `static` (respectively instance) field.

In particular, no linkage error will occur in the case where a class could no longer be recompiled because a field access previously referenced a field of a superclass with an incompatible type. The previously compiled class with such a reference will continue to reference the field declared in a superclass.

Thus compiling and executing the code:

```
class Hyper { String h = "hyper"; }
class Super extends Hyper { String s = "super"; }
class Test {
    public static void main(String[] args) {
        System.out.println(new Super().h);
    }
}
```

produces the output:

```
hyper
```

Changing `Super` to be defined as:

```
class Super extends Hyper {
    String s = "super";
    int h = 0;
}
```

recompiling `Hyper` and `Super`, and executing the resulting new binaries with the old binary of `Test` produces the output:

```
hyper
```

The field `h` of `Hyper` is output by the original binary of `main`. While this may seem surprising at first, it serves to reduce the number of incompatibilities that occur at run time. (In an ideal world, all source files that needed recompilation would be recompiled whenever any one of them changed, eliminating such surprises. But such a mass recompilation is often impractical or impossible, especially in the Internet. And, as was previously noted, such recompilation would sometimes require further changes to the source code.)

As an example, if the program:

```
class Hyper { String h = "Hyper"; }
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

is compiled and executed, it produces the output:

```
Hyper
```

Suppose that a new version of class `Super` is then compiled:

```
class Super extends Hyper { char h = 'h'; }
```

If the resulting binary is used with the existing binaries for `Hyper` and `Test`, then the output is still:

```
Hyper
```

even though compiling the source for these binaries:

```
class Hyper { String h = "Hyper"; }
class Super extends Hyper { char h = 'h'; }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

would result in a compile-time error, because the `h` in the source code for `main` would now be construed as referring to the `char` field declared in `Super`, and a `char` value can't be assigned to a `String`.

Deleting a field from a class will break compatibility with any pre-existing binaries that reference this field, and a `NoSuchFieldError` will be thrown when such a reference from a pre-existing binary is linked. Only `private` fields may be safely deleted from a widely distributed class.

For purposes of binary compatibility, adding or removing a field *f* whose type involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of a field of the same name whose type is the erasure (§4.6) of the type of *f*.

13.4.9 `final` Fields and Constants

If a field that was not `final` is changed to be `final`, then it can break compatibility with pre-existing binaries that attempt to assign new values to the field.

For example, if the program:

```
class Super { static char s; }
class Test extends Super {
    public static void main(String[] args) {
        s = 'a';
        System.out.println(s);
    }
}
```

is compiled and executed, it produces the output:

```
a
```

Suppose that a new version of class `Super` is produced:

```
class Super { final static char s = ’b’; }
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in a `IllegalAccessError`.

Deleting the keyword `final` or changing the value to which a field is initialized does not break compatibility with existing binaries.

If a field is a constant variable (§4.12.4), then deleting the keyword `final` or changing its value will not break compatibility with pre-existing binaries by causing them not to run, but they will not see any new value for the usage of the field unless they are recompiled. This is true even if the usage itself is not a compile-time constant expression (§15.28)

If the example:

```
class Flags { final static boolean debug = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.debug)
            System.out.println("debug is true");
    }
}
```

is compiled and executed, it produces the output:

```
debug is true
```

Suppose that a new version of class `Flags` is produced:

```
class Flags { final static boolean debug = false; }
```

If `Flags` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` produces the output:

```
debug is true
```

because the value of `debug` was a compile-time constant, and could have been used in compiling `Test` without making a reference to the class `Flags`.

This result is a side-effect of the decision to support conditional compilation, as discussed at the end of §14.20.

This behavior would not change if `Flags` were changed to be an interface, as in the modified example:

```
interface Flags { boolean debug = true; }
class Test {
   public static void main(String[] args) {
      if (Flags.debug)
         System.out.println("debug is true");
   }
}
```

(One reason for requiring inlining of constants is that `switch` statements require constants on each `case`, and no two such constant values may be the same. The compiler checks for duplicate constant values in a `switch` statement at compile time; the `class` file format does not do symbolic linkage of `case` values.)

The best way to avoid problems with “inconstant constants” in widely-distributed code is to declare as compile time constants only values which truly are unlikely ever to change. Many compile time constants in interfaces are small integer values replacing enumerated types, which the language does not support; these small values can be chosen arbitrarily, and should not need to be changed. Other than for true mathematical constants, we recommend that source code make very sparing use of class variables that are declared `static` and `final`. If the read-only nature of `final` is required, a better choice is to declare a `private static` variable and a suitable accessor method to get its value. Thus we recommend:

```
private static int N;
public static int getN() { return N; }
```

rather than:

```
public static final int N = ...;
```

There is no problem with:

```
public static int N = ...;
```

if `N` need not be read-only. We also recommend, as a general rule, that only truly constant values be declared in interfaces. We note, but do not recommend, that if a field of primitive type of an interface may change, its value may be expressed idiomatically as in:

```
interface Flags {
   boolean debug = new Boolean(true).booleanValue();
}
```

insuring that this value is not a constant. Similar idioms exist for the other primitive types.

One other thing to note is that `static final` fields that have constant values (whether of primitive or `String` type) must never appear to have the default initial value for their type (§4.5.5). This means that all such fields appear to be initialized first during class initialization (§8.3.2.1, §9.3.1, §12.4.2).

13.4.10 `static` Fields

If a field that is not declared `private` was not declared `static` and is changed to be declared `static`, or vice versa, then a linkage time error, specifically an `IncompatibleClassChangeError`, will result if the field is used by a preexisting binary which expected a field of the other kind. Such changes are not recommended in code that has been widely distributed.

13.4.11 `transient` Fields

Adding or deleting a `transient` modifier of a field does not break compatibility with pre-existing binaries.

13.4.12 Method and Constructor Declarations

Adding a method or constructor declaration to a class will not break compatibility with any pre-existing binaries, in the case where a type could no longer be recompiled because an invocation previously referenced a method or constructor of a superclass with an incompatible type. The previously compiled class with such a reference will continue to reference the method or constructor declared in a superclass.

Assume a reference to a method *m* with qualifying type *T*. Assume further that *m* is in fact an instance (respectively `static`) method declared in a superclass of *T*, *S*. If a new method of type *X* with the same signature and return type as *m* is added to a subclass of *S* that is a superclass of *T* or *T* itself, then a linkage error may occur. Such a linkage error will occur only if, in addition to the above, either one of the following conditions hold:

- The new method is less accessible than the old one.
- The new method is a `static` (respectively instance) method.

Deleting a method or constructor from a class may break compatibility with any pre-existing binary that referenced this method or constructor; a `NoSuchMethodError` may be thrown when such a reference from a pre-existing binary is linked. Such an error will occur only if no method with a matching signature and return type is declared in a superclass.

If the source code for a class contains no declared constructors, the Java compiler automatically supplies a constructor with no parameters. Adding one or more constructor declarations to the source code of such a class will prevent this default constructor from being supplied automatically, effectively deleting a constructor, unless one of the new constructors also has no parameters, thus replacing the default constructor. The automatically supplied constructor with no parameters is given the same access modifier as the class of its declaration, so any replacement should have as much or more access if compatibility with pre-existing binaries is to be preserved.

13.4.13 Method and Constructor Formal Type Parameters

Renaming a type variable (§4.4) declared as a formal type parameter of a method or constructor has no effect with respect to pre-existing binaries. Adding or removing a type parameter does not, in itself, have any implications for binary compatibility.

DISCUSSION

Note that if such type variables are used in the type of the method or constructor, that may have the normal implications of changing the aforementioned type.

Changing the first bound of a type parameter will change the erasure (§4.6) of any member that uses that type variable in its own type, and this may effect binary compatibility. Specifically,:

- If the type parameter is used as the type of a field, the effect is as if the field was removed and a field with the same name, whose type is the new erasure of the type variable, was added.
- If the type variable is used as the type of any formal parameter of a method, but not as the return type, the effect is as if that method were removed, and replaced with a new method that is identical except for the types of the aforementioned formal parameters, which now have the new erasure of the type variable as their type.
- If the type variable is used as a return type of a method, but not as the type of any formal parameter of the method, the effect is as if that method were

removed, and replaced with a new method that is identical except for the return type, which is now the new erasure of the type variable.

- If the type variable is used as a return type of a method and as the type of some formal paramters of the method, the effect is as if that method were removed, and replaced with a new method that is identical except for the return type, which is now the new erasure of the type variable, and except for the types of the aforementioned formal parameters, which now have the new erasure of the type variable as their type.

Changing any other bound has no effect on binary compatibility.

13.4.14 Method and Constructor Parameters

Changing the name of a formal parameter of a method or constructor does not impact pre-existing binaries. Changing the name of a method, the type of a formal parameter to a method or constructor, or adding a parameter to or deleting a parameter from a method or constructor declaration creates a method or constructor with a new signature, and has the combined effect of deleting the method or constructor with the old signature and adding a method or constructor with the new signature (see §13.4.12).

For purposes of binary compatibility, adding or removing a method or constructor *m* whose signature involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of the a method whose signature is the erasure (§4.6) of the signature of *m*.

13.4.15 Method Result Type

Changing the result type of a method, replacing a result type with `void`, or replacing `void` with a result type has the combined effect of deleting the old method and adding a new method with the new result type or newly `void` result (see §13.4.12).

For purposes of binary compatibility, adding or removing a method or constructor *m* whose return type involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of the a method whose return type is the erasure (§4.6) of the return type of *m*.

13.4.16 `abstract` Methods

Changing a method that is declared `abstract` to no longer be declared `abstract` does not break compatibility with pre-existing binaries.

Changing a method that is not declared `abstract` to be declared `abstract` will break compatibility with pre-existing binaries that previously invoked the method, causing an `AbstractMethodError`.

If the example program:

```
class Super { void out() { System.out.println("Out"); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println("Way ");
        t.out();
    }
}
```

is compiled and executed, it produces the output:

```
Way
Out
```

Suppose that a new version of class `Super` is produced:

```
abstract class Super {
    abstract void out();
}
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in a `AbstractMethodError`, because class `Test` has no implementation of the method `out`, and is therefore is (or should be) abstract.

13.4.17 `final` Methods

Changing an instance method that is not `final` to be `final` may break compatibility with existing binaries that depend on the ability to override the method.

If the test program:

```
class Super { void out() { System.out.println("out"); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        t.out();
    }
            void out() { super.out(); }
}
```

is compiled and executed, it produces the output:

```
    out
```

Suppose that a new version of class `Super` is produced:

```
    class Super { final void out() { System.out.println("!"); } }
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in a `VerifyError` because the class `Test` improperly tries to override the instance method `out`.

Changing a class (`static`) method that is not `final` to be `final` does not break compatibility with existing binaries, because the method could not have been overridden.

Removing the `final` modifier from a method does not break compatibility with pre-existing binaries.

13.4.18 `native` Methods

Adding or deleting a `native` modifier of a method does not break compatibility with pre-existing binaries.

The impact of changes to types on preexisting `native` methods that are not recompiled is beyond the scope of this specification and should be provided with the description of an implementation. Implementations are encouraged, but not required, to implement `native` methods in a way that limits such impact.

13.4.19 `static` Methods

If a method that is not declared `private` was declared `static` (that is, a class method) and is changed to not be declared `static` (that is, to an instance method), or vice versa, then compatibility with pre-existing binaries may be broken, resulting in a linkage time error, namely an `IncompatibleClassChangeError`, if these methods are used by the pre-existing binaries. Such changes are not recommended in code that has been widely distributed.

13.4.20 `synchronized` Methods

Adding or deleting a `synchronized` modifier of a method does not break compatibility with existing binaries.

13.4.21 Method and Constructor Throws

Changes to the `throws` clause of methods or constructors do not break compatibility with existing binaries; these clauses are checked only at compile time.

13.4.22 Method and Constructor Body

Changes to the body of a method or constructor do not break compatibility with pre-existing binaries.

We note that a compiler cannot expand a method inline at compile time. The keyword `final` on a method does not mean that the method can be safely inlined; it means only that the method cannot be overridden. It is still possible that a new version of that method will be provided at link time. Furthermore, the structure of the original program must be preserved for purposes of reflection.

In general we suggest that implementations use late-bound (run-time) code generation and optimization.

13.4.23 Method and Constructor Overloading

Adding new methods or constructors that overload existing methods or constructors does not break compatibility with pre-existing binaries. The signature to be used for each invocation was determined when these existing binaries were compiled; therefore newly added methods or constructors will not be used, even if their signatures are both applicable and more specific than the signature originally chosen.

While adding a new overloaded method or constructor may cause a compile-time error the next time a class or interface is compiled because there is no method or constructor that is most specific (§15.12.2.2), no such error occurs when a program is executed, because no overload resolution is done at execution time.

If the example program:

```
class Super {
    static void out(float f) { System.out.println("float"); }
}

class Test {
    public static void main(String[] args) {
        Super.out(2);
    }
}
```

is compiled and executed, it produces the output:

```
float
```

Suppose that a new version of class `Super` is produced:

```
class Super {
    static void out(float f) { System.out.println("float"); }
```

```
        static void out(int i) { System.out.println("int"); }
    }
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` still produces the output:

```
    float
```

However, if `Test` is then recompiled, using this new `Super`, the output is then:

```
    int
```

as might have been naively expected in the previous case.

13.4.24 Method Overriding

If an instance method is added to a subclass and it overrides a method in a superclass, then the subclass method will be found by method invocations in pre-existing binaries, and these binaries are not impacted. If a class method is added to a class, then this method will not be found unless the qualifying type of the reference is the subclass type.

13.4.25 Static Initializers

13.4.26 Adding, deleting, or changing a static initializer (§8.7) of a class does not impact pre-existing binaries.**Evolution of Enums**

Adding or reordering constants from an enum type will not break compatibility with pre-existing binaries.

If a precompiled binary attempts to access an enum constant that no longer exists, the client will fail at runtime with a `NoSuchFieldError`. Therefore such a change is not recommended for widely distributed enums.

In all other respects, the binary compatibility rules for enums are identical to those for classes.

13.5 Evolution of Interfaces

This section describes the impact of changes to the declaration of an interface and its members on pre-existing binaries.

13.5.1 `public` Interfaces

Changing an interface that is not declared `public` to be declared `public` does not break compatibility with pre-existing binaries.

If an interface that is declared `public` is changed to not be declared `public`, then an `IllegalAccessError` is thrown if a pre-existing binary is linked that needs but no longer has access to the interface type, so such a change is not recommended for widely distributed interfaces.

13.5.2 Superinterfaces

Changes to the interface hierarchy cause errors in the same way that changes to the class hierarchy do, as described in §13.4.4. In particular, changes that result in any previous superinterface of a class no longer being a superinterface can break compatibility with pre-existing binaries, resulting in a `VerifyError`.

13.5.3 The Interface Members

Adding a method to an interface does not break compatibility with pre-existing binaries. A field added to a superinterface of *C* may hide a field inherited from a superclass of *C*. If the original reference was to an instance field, an `IncompatibleClassChangeError` will result. If the original reference was an assignment, an `IllegalAccessError` will result.

Deleting a member from an interface may cause linkage errors in pre-existing binaries.

If the example program:

```
interface I { void hello(); }
class Test implements I {
   public static void main(String[] args) {
      I anI = new Test();
      anI.hello();
   }
           public      void      hello()      {      Sys-
tem.out.println("hello"); }
}
```

is compiled and executed, it produces the output:

```
hello
```

Suppose that a new version of interface `I` is compiled:

```
interface I { }
```

If `I` is recompiled but not `Test`, then running the new binary with the existing binary for `Test` will result in a `NoSuchMethodError`. (In some early implementations this program still executed; the fact that the method `hello` no longer exists in interface `I` was not correctly detected.)

13.5.4 Interface Formal Type Parameters

The effects of changes to the formal parameters of an interface are the same as those of analogous changes to the formal parameters of a class.

13.5.5 Field Declarations

The considerations for changing field declarations in interfaces are the same as those for `static final` fields in classes, as described in §13.4.8 and §13.4.9.

13.5.6 Abstract Method Declarations

The considerations for changing abstract method declarations in interfaces are the same as those for `abstract` methods in classes, as described in §13.4.14, §13.4.15, §13.4.21, and §13.4.23.

13.5.7 Evolution of Annotation Types

Annotation types behave exactly like any other interface. Adding or removing an element from an annotation type is analogous to adding or removing a method. There are important considerations governing other changes to annotation types, but these have no effect on the linkage of binaries by the Java virtual machine. Rather, such changes effect the behavior of reflective APIs that manipulate annotations. The documentation of these APIs specifes their behavior when various changes are made to the underlying annotation types.

Adding or removing annotations has no effect on the correct linkage of the binary representations of programs in the Java programming language.

Lo! keen-eyed, towering Science! . . .
Yet again, lo! the Soul—above all science . . .
For it, the partial to the permanent flowing,
For it, the Real to the Ideal tends.
For it, the mystic evolution . . .
—Walt Whitman, *Song of the Universal* (1874)