Kinds of Compatibility

When evolving the JDK, compatibility concerns are taken very seriously. However, different standards are applied to evolving various aspects of the platform. From a certain point of view, it is true that any observable difference could *potentially* cause some unknown application to break. Indeed, just changing the reported version number is incompatible in this sense because, for example, a JNLP file can refuse to run an application on later versions of the platform. Therefore, since making no changes at all is clearly not a viable policy for evolving the platform, changes need to be evaluated against and managed according to a variety of compatibility contracts.

For Java programs, there are three main categories of compatibility:

1. **Source:** Source compatibility concerns translating Java source code into class files.
2. **Binary:** Binary compatibility is defined in *The Java Language Specification* as preserving the ability to link without error.
3. **Behavioral:** Behavioral compatibility includes the semantics of the code that is executed at runtime.

Note that non-source compatibility is sometimes colloquially referred to as "binary compatibility." Such usage is incorrect since the Java Language Specification (JLS) spends an entire chapter precisely defining the term binary compatibility; often behavioral compatibility is the intended notion instead.

Other kinds of compatibility include serial compatibility for Serializable types and *migration compatibility*. Migration compatibility was a constraint on how generics were added to the platform; libraries and their clients had to be able to be generified independently while preserving the ability of code to be compiled and run.

The basic challenge of compatibility is judging if the benefits of a change outweigh the possibly negative consequences (if any) to existing software and systems impacted by the change. In a closed-world scenario where all the clients of an API are known and can in principle be simultaneously changed, introducing "incompatible" changes is just the small matter of being able to coordinate the engineering necessary to make the change. In contrast, for APIs that are used as widely as the JDK, rigorously finding all the possible programs impacted by an incompatible change is impractical. So evolving such APIs in this environment is quite constrained by comparison.

Generally, we will consider whether a program P is compatible in some fashion (or not) with respect to two versions of a library L_1 and L_2 that differ in some way. (We will not consider the compatibility impact of such changes to independent implementers of L.) Sometimes only a particular program is of interest; is the change from L_1 to L_2 compatible with *this* program? When evaluating how the platform should evolve, a broader consideration of the programs of concern is used. For example, does the change from L_1 to L_2 cause a problem for any program that *currently* exists? If so, what fraction of existing programs is affected? Finally, the broadest consideration is does the change affect any program that *could* exist? Often once a platform version is released, the latter two notions are similar because imperfect knowledge about the set of actual programs means it can be more tractable to consider the worst possible outcome for any potential program rather than estimate the impact over actual programs. Stated more formally, depending on the change being considered, judging the change based on the worst possible outcome for any program is more appropriate than judging based on some other kind of norm of the disruption over the space of known programs.

Generally each kind of compatibility has both positive and negative aspects; that is, the positive aspect of keeping things that "work" working and the negative aspect of keeping things that "don't work" *not* working. For example, the TCK tests for Java compilers include both positive tests of programs that must be accepted and negative tests of programs that must be rejected. In many circumstances, preserving or expanding the positive behavior is more acceptable and important than maintaining the negative behavior and this guide will focus on positive compatibility.

In terms of relative severity, source compatibility problems are usually the mildest since there are often straightforward workarounds, such as adjusting import statements or switching to fully qualified names. Gradations of source compatibility are identified and discussed below. Behavioral compatibility problems can have a range of impacts while true binary compatibility issues are problematic since linking is prevented.

Source Compatibility

> *The basic job of any linker or loader is simple: It binds more abstract names to more concrete names, which permits programmers to write code using the more abstract names. (*Linkers and Loaders*)*

A Java compiler's job also includes mapping more abstract names to more concrete ones, specifically mapping simple and qualified names appearing in source code into binary names in class files. Source compatibility concerns this mapping of source code into class files, not only whether or not such a mapping is possible, but also whether or not the resulting class files are suitable. Source compatibility is influenced by changing the set of types available during compilation, such as adding a new class, as well as changes within existing types themselves, such as adding an overloaded method. There is a large set of possible changes to classes and interfaces examined for their binary compatibility impact. All these changes could also be classified according to their source compatibility repercussions, but only a few kinds of changes will be analyzed below.

The most rudimentary kind of positive source compatibility is whether code that compiles against L_1 will continue to compile against L_2; however, that is not the entirety of the space of concerns since the class file resulting from compilation might *not* be equivalent. Java source code often uses simple names for types; using information about imports, the compiler will interpret these simple names and transform them into binary names for use in the resulting class file(s). In a class file, the binary name of an entity (along with its signature in the case of methods and constructors) serves as the unique, universal identifier to allow the entity to be referenced. So different degrees of source compatibility can be identified:

- Does the client code still compile (or not compile)?
- If the client code still compiles, do all the names resolve to the same binary names in the class file?
- If the client code still compiles and the names do *not* all resolve to the same binary names, does a *behaviorally equivalent* class file result?

Whether or not a program is valid can also be affected by language changes. Usually previously invalid program are made valid, as when generics were added, but sometimes existing programs are rendered invalid, as when keywords were added (`strictfp`, `assert`, and `enum`). The version number of the resulting class file is also an external compatibility issue of sorts since it restricts which platform versions the code can be run on. Also, the compilation strategies and environment can vary when different platform versions are used to produce different class file versions, meaning compiler bug fixes and differences in compiler-internal contracts can affect the contents of the resulting class files.

Full source compatibility with *any* existing program is usually not achievable because of `*` imports. For example, consider L_1 with packages `foo` and `bar` where `foo` includes the class `Quux`. Then L_2 adds class `bar.Quux`. Consider the following program:

```
import foo.*;
import bar.*;

public class HelloQuux {
    public static void main(String... args) {
        Object o = Quux.class;
        System.out.println("Hello " + o.toString());
    }
}
```

The `HelloQuux` class will compile under L_1 but *not* under L_2 since the name "`Quux`" is now ambiguous as reported by `javac`:

```
HelloQuux.java:6: reference to Quux is ambiguous, both class bar.Quux in bar and
 class foo.Quux in foo match
        Object o = Quux.class;
                   ^
1 error
```

An adversarial program could almost always include `*` imports that conflict with a given library. Therefore, judging source compatibility by requiring *all* possible programs to compile is an overly restrictive criterion. However, when naming their types, API designers should *not* reuse "`String`", "`Object`", and other names of core classes from packages like `java.lang` and `java.util` to avoid this kind of annoying name conflict.

Due to the `*` import wrinkle, a more reasonable definition of source compatibility considers programs transformed to only use fully qualified names. Let *FQN* *P*, *L*) be program *P* where each name is replaced by its fully qualified form in the context of libraries *L*. Call such a library transformation from L_1 to L_2 *binar y-preserving source compatible* with source program *P* if *FQN*(*P*, L_1) equals *FQN*(*P*, L_2). This is a strict form of source compatibility that will usually result in class files for *P* using the same binary names when compiled against both versions of the library. Class files with the same binary names will result when each type has a distinct fully qualified name. Multiple types can have the same fully qualified name but differing binary names; those cases do not arise when the standard naming conventions are being followed.

To illustrate the differing degrees of source compatibility, consider class `Lib` below.

```
// Original version
public final class Lib {
  public double foo(double d) {
    return d * 2.0;
  }
}
```

A change that could break compilation of existing clients is removing method `foo`:

```
// Change that breaks compilation
public final class Lib {
  // Where oh where has foo gone?
}
```

Removing a method is also binary incompatible. The remaining changes to `Lib` discussed below will all preserve binary compatibility.

Adding a method with a name distinct from any existing method is binary-preserving source compatible; the class files produced by recompiling existing clients of the library will be semantically equivalent since the source methods will be resolved the same way.

```
// Binary-preserving source compatible change
public final class Lib {
  public double foo(double d) {
    return d * 2.0;
  }

  // Method with new name added
  public int bar() {
    return 42;
  }
}
```

However, adding overloaded methods has the potential to change method resolution and thus change the signatures of the method call sites in the resulting class file. Whether or not such a change is problematic with respect to source compatibility depends on what semantics are required and how the different overloaded methods operate on the same inputs, which interacts with behavioral equivalence notions. For example, consider adding to `Lib` an overloading of `foo` which takes an `int`:

```
// Behaviorally equivalent source compatible change
public final class Lib {
  public double foo(double d) {
    return d * 2.0;
  }

  // New overloading
  public double foo(int i) {
    return i * 2.0;
  }
}
```

In the original version of `Lib`, a call to `foo` with an integer argument will resolve to `foo(double)` and under the rules for method invocation conversion the value of the `int` argument will be converted to a `double` through a primitive widening conversion. So given client code

```
public class Client {
  public static void main(String... args) {
    int i = 42;
    double d = (new Lib()).foo(i);
  }
}
```

The call to `foo` gets compiled as byte code into a series of instructions like, in `javap` style output:

```
   ...
   10:        iload_1
   11:        i2d
   12:        invokevirtual        #4; //Method Lib.foo:(D)D
   ...
```

The `i2d` instructions converts an `int` to `double`. The string "`C.foo:(D)D`" indicates this is a call to method `foo` in class `Lib` where `foo` takes one `doubl` argument and returns a `double` result.

In contrast, when the same client code is compiled against the behaviorally equivalent version of the code with an overloading for `foo`, the newly added overloading of `foo` is selected instead and the argument does not need any conversion:

```
   ...
   10:        iload_1
   11:        invokevirtual        #4; //Method Lib.foo:(I)D
   ...
```

The string "`Lib.foo:(I)D`" indicates the `foo` method taking one `int` argument is selected and there is no intermediary conversion instruction from `int` to `double` to convert the argument before the `invokevirtual` instruction to call the method.

In terms of operations on the arguments and computation of the results, these two overloadings of `foo` are operationally equivalent. For `int` call sites, both methods start by converting the `int` argument to `double`. In the original method, this conversion is done before the method call; in the new method, the conversion is done inside the method call before the multiply by 2.0. Next, the `int`-converted-to-`double` is multiplied by 2.0 and the product returned.

Not all overloaded methods are behaviorally equivalent; some are just *compilation preserving*. For example, consider adding a third `foo` method that takes a `long` argument:

```
// Compilation preserving source compatible change
public final class Lib {
  public double foo(double d) {
    return d * 2.0;
  }

  public double foo(int i) {
    return i * 2.0;
  }

  // New overloading, not behaviorally equivalent
  public double foo(long el) {
    return (double) (el * 2L;)
  }
}
```

In the previous versions of `Lib`, a call to `foo` with a `long` argument would resolve to calling `foo(double)`, in which case the value of the `long` argument would be converted to `double` before the method was called. Then inside the body of `foo`, the value would be multiplied by 2.0 and returned. However, with the presence of the `foo(long)` overloading, call sites to `foo` with a `long` argument will resolve to calling `foo(long)` instead of `foo(double)`. The `foo(long)` method *first* multiplies by 2 and *then* converts to `double`, the opposite order of operations compared to calling `foo(double)`. Whether or not the argument value is converted to `double` before or after the multiply by two matters since the two sequences of operations can yield different results. For example, a large positive `long` value multiplied by two can overflow to a *negative* value, but a large positive `double` value when multiplied by two will retain a positive sign. This kind of subtle change in overloading behavior occurred with the addition of a `BigDecimal` constructor taking a `long` argument as part of JSR 13.

When adding an overloaded method or constructor to an existing library, if the newly added method could be applicable to the same call sites as the original method, such as if the new method takes the same number of arguments as an original method and has more specific types, call sites in existing clients may now resolve to the new method when recompiled. Well-written programs will follow the Liskov substitution principle and perform "the same" operation on the argument no matter which overloaded method is called. Less than well-written programs may fail to follow this principle.

blocked URL

If a new method or constructor cannot change resolution in existing clients, then the change is a binary-preserving source transformation. In binary-preserving source compatibility, existing clients will yield equivalent class files if recompiled. The difference between behaviorally equivalent and compilation preserving source compatibility that is *not* behaviorally equivalent depends on the implementation of the methods in question. If a new method changes resolution, if the different class file that results has similar enough behavior, the change may still be acceptable, while changing resolution in such a way that does *not* preserve semantics is likely problematic. Changing a library in such a way that current clients no longer compile is seldom appropriate.

Binary Compatibility

> ***JLS §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.*

The JLS defines binary compatibility strictly according to linkage; if *P* links with L_1 and continues to link with L_2, the change made in L_2 is binary compatible. The runtime behavior after linking is *not* included in binary compatibility:

> ***JLS §13.4.22 – Method and Constructor Body***
> *Changes to the body of a method or constructor do not break [binary] compatibility with pre-existing binaries.*

As an extreme example, if the body of a method is changed to throw an error instead of compute a useful result, while the change is certainly a compatibility issue, it is *not* a binary compatibility issue since client classes would continue to link. Also, it is *not* a binary compatibility issue to add methods to an interface. Class files compiled against the old version of the interface will still link against the new interface *despite* the class not having an implementation of the new method. If the new method is called at runtime, an `AbstractMethodError` is thrown; if the new method is not called, the existing methods can be used without incident. (Adding a method to an interface is a source incompatibility that can break compilation though.)

Binary compatibility across releases has long been a policy of JDK evolution. Long-term portable binaries are judged to be a great asset to the Java SE ecosystem. For this reason, even old deprecated methods have to date been kept in the platform so that legacy class files with references to them will continue to link.

Behavioral Compatibility

Intuitively, behavioral compatibility should mean that with the same inputs program *P* does "the same" or an "equivalent" operation under different versions of libraries or the platform. Defining equivalence can be a bit involved; for example, even just defining a proper `equals` method in a class can be nontrivial. In this case, to formalize this concept would require an *operational semantics* for the JVM for the aspects of the system a program was interested in. For example, there is a fundamental difference in visible changes between programs that introspect on the system and those that do not. Examples of introspection include calling core reflection, relying on stack trace output, using timing measurements to influence code execution, and so on. For programs that do not use, say, core reflection, changes to the structure of libraries, such as adding new `public` methods, is entirely transparent. In contrast, a (poorly behaved) program could use reflection to look up the set of `public` methods on a library class and throw an exception if any unexpected methods were present. A tricky program could even make decisions based on information like a timing side channel. For example, two threads could repeatedly run different operations and make some indication of progress, for example, incrementing an atomic counter, and the relative rates of progress could be compared. If the ratio is over a certain threshold, some unrelated action could be taken, or not. This allows a program to create a dependence on the optimization capabilities of a particular JVM implementation, which is generally outside a reasonable behavioral compatibility contract.

The evolution of a library is constrained by the library's contract included in its specification; for `final` classes this contract doesn't usually include a prohibition of adding new public methods! While an end-user may not care why a program does not work with a newer version of a library, what contracts are being followed or broken should determine which party has the onus for fixing the problem. That said, there are times in evolving the JDK when differences are found between the specified behavior and the actual behavior (for example JDK-4707389, JDK-6365176). The two basic approaches to fixing these bugs are to change the implementation to match the specified behavior or to change the specification (in a platform release) to match the implementation's (perhaps long-standing) behavior; often the latter option is chosen since it has a lower de facto impact on behavioral compatibility.

While many classes and methods in the platform describe the exact input-output relationship between arguments and returned values, a few methods eschew this approach and are specified to have unspecified behavior. One such example is `HashSet`:

> *[`HashSet`] makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.*
> *...*
>
> *[The `iterator` method] Returns an iterator over the elements in this set. The elements are returned in no particular order.*

The iterator algorithm can and has varied over the years. These variations are fully source and binary compatible. While such behavioral differences are fine in a platform release, because of behavioral compatibility they are marginally acceptable for a maintenance release, and questionable for an update release.

Other Kinds of Compatibility

Besides Java SE APIs, as a platform the JDK exposes many other kinds of exported *interfaces*. These interfaces should generally be evolved analogously to behavioral compatibility in Java SE APIs, avoiding gratuitously breaking clients of the interface.

Managing Compatibility

> ***Original Preface to JLS***
> *Except for timing dependencies or other non-determinisms and given sufficient time and sufficient memory space, a program written in the Java programming language should compute the same result on all machines and in all implementations.*

The above statement from the original JLS could be regarded as vacuously true about any platform: except for the non-determinisms, a program is deterministic. The difference was that in Java, with programmer discipline, the set of deterministic programs was nontrivial and the set of *predictable* programs was quite large. In other words, the platform provider and the programmer both have responsibilities in making programs portable in practice; the platform should abide by the specification and conversely programs should tolerate any valid implementation of the specification.

Evolving the platform is a balance between maintaining stability to enjoy various kinds of compatibility and making changes to enjoy various kinds of progress.