JEP 457: Class-File API (Preview)

Author	Brian Goetz
Owner	Adam Sotona
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	22
Component	core-libs / java.lang.classfile
Discussion	classfile dash api dash dev at openjdk dot org
Effort	M
Duration	M
Relates to	JEP 466: Class-File API (Second Preview)
Reviewed by	Alex Buckley, Paul Sandoz
Endorsed by	Paul Sandoz
Created	2022/01/20 14:51
Updated	2024/02/05 15:31
Issue	8280389

Summary

Provide a standard API for parsing, generating, and transforming Java class files. This is a preview API.

Goals

- Provide an API for processing class files that tracks the `class` file format defined by the Java Virtual Machine Specification.
- Enable JDK components to migrate to the standard API, and eventually remove the JDK's internal copy of the third-party ASM library.

Non-Goals

- It is not a goal to obsolete existing libraries that process class files, nor to be the world's fastest class-file library.
- It is not a goal to extend the Core Reflection API to give access to the bytecode of loaded classes.
- It is not a goal to provide code analysis functionality; that can be layered atop the Class-File API via third-party libraries.

Motivation

Class files are the lingua franca of the Java ecosystem. Parsing, generating, and transforming class files is ubiquitous because it allows independent tools and libraries to examine and extend programs without jeopardizing the maintainability of source code. For example, frameworks use on-the-fly bytecode transformation to transparently add functionality that would be impractical, if not impossible, for application developers to include in source code.

The Java ecosystem has many libraries for parsing and generating class files, each with different design goals, strengths, and weaknesses. Frameworks that process class files generally bundle a class-file library such as ASM, BCEL, or Javassist. However, a significant problem for class-file libraries is that the class-file format is evolving more quickly than in the past, due to the six-month release cadence of the JDK. In recent years, the class-file format has evolved to support Java language features such as sealed classes and to expose JVM features such as dynamic constants and nestmates. This trend will continue with forthcoming features such as value classes and generic method specialization.

Because the class-file format can evolve every six months, frameworks are more frequently encountering class files that are newer than the class-file library that they bundle. This version skew results in errors visible to application developers or, worse, in framework developers trying to write code to parse class files from the future and engaging in leaps of faith that nothing too serious will change. Framework developers need a class-file library that they can trust is up-to-date with the running JDK.

The JDK has its own class-file library inside the `javac` compiler. It also bundles ASM to implement tools such as `jar` and `jlink`, and to support the implementation of lambda expressions at run time. Unfortunately, the JDK's use of a third-party library causes a tiresome delay in the uptake of new class-file features across the ecosystem. The ASM version for JDK N cannot finalize until after JDK N finalizes, so tools in JDK N cannot handle class-file features that are new in JDK N, which means `javac` cannot safely emit class-file features which are new in JDK N until JDK N+1. This is especially problematic when JDK N is a highly anticipated release such as JDK 21, and developers are eager to write programs that entail the use of new class-file features.

The Java Platform should define and implement a standard class-file API that evolves together with the class-file format. Components of the Platform would be able to rely solely on this API, rather than rely perpetually on the willingness of third-party developers to update and test their class-file libraries. Frameworks and tools that use the standard API would support class files from the latest JDK automatically, so that new language and VM features with representation in class files could be adopted quickly and easily.

Description

We have adopted the following design goals and principles for the Class-File API.

- *Class-file entities are represented by immutable objects* — All class-file entities, such as fields, methods, attributes, bytecode instructions, annotations, etc., are represented by immutable objects. This facilitates reliable sharing when a class file is being transformed.
- *Tree-structured representation* — A class file has a tree structure. A class has some metadata (name, superclass, etc.), and a variable number of fields, methods, and attributes. Fields and methods themselves have metadata and further contain attributes, including the `Code` attribute. The `Code` attribute further contains instructions, exception handlers, and so forth. The API for navigating and building class files should reflect this structure.
- *User-driven navigation* — The path we take through the class-file tree is driven by user choices. If the user cares only about annotations on fields then we should only have to parse as far down as the annotation attributes inside the `field_info` structure; we should not have to look into any of the class attributes or the bodies of methods, or at other attributes of the field. Users should be able to deal with compound entities, such as methods, either as single units or as streams of their constituent parts, as desired.
- *Laziness* — User-driven navigation enables significant efficiencies, such as not parsing any more of the class file than is required to satisfy the user's needs. If the user is not going to dive into the contents of a method then we need not parse any more of the `method_info` structure than is needed to figure out where the next class-file element starts. We can lazily inflate, and cache, the full representation when the user asks for it.
- *Unified streaming and materialized views* — Like ASM, we want to support both a streaming and a materialized view of a class file. The streaming view is suitable for the majority of use cases, while the materialized view is more general since it enables random access. We can provide a materialized view far less expensively than ASM through laziness, as enabled by immutability. We can, further, align the streaming and materialized views so that they use a common vocabulary and can be used in coordination, as is convenient for each use case.
- *Emergent transformation* — If the class-file parsing and generation APIs are sufficiently aligned then transformation can be an emergent property that does not require its own special mode or significant new API surface. (ASM achieves this by using a common visitor structure for readers and writers.) If classes, fields, methods, and code bodies are readable and writable as streams of elements then a transformation can be viewed as a flat-map operation on this stream, defined by lambdas.
- *Detail hiding* — Many parts of a class file (constant pool, bootstrap method table, stack maps, etc.) are derived from other parts of the class file. It makes no sense to ask the user to construct these directly; this is extra work for the user and increases the chance of error. The API will automatically generate entities that are tightly coupled to other entities based on the fields, methods, and instructions added to the class file.
- *Lean into the language* — In 2002, the visitor approach used by ASM seemed clever, and was surely more pleasant to use than what came before. However, the Java programming language has improved tremendously since then — with the introduction of lambdas, records, sealed classes, and pattern matching — and the Java Platform now has a standard API for describing class-file constants (`java.lang.constant`). We can use these features to design an API that is more flexible and pleasant to use, less verbose, and less error-prone.

Elements, builders, and transforms

The Class-File API resides in the `java.lang.classfile` package and subpackages. It defines three main abstractions:

- An *element* is an immutable description of some part of a class file; it may be an instruction, attribute, field, method, or an entire class file. Some elements, such as methods, are *compound elements*; in addition to being elements they also contain elements of their own, and can be dealt with whole or else further decomposed.
- Each kind of compound element has a corresponding *builder* which has specific building methods (e.g., `ClassBuilder::withMethod`) and is also a `Consumer` of the appropriate element type.
- Finally, a *transform* represents a function that takes an element and a builder and mediates how, if at all, that element is transformed into other elements.

We introduce the API by showing how it can be used to parse class files, generate class files, and combine parsing and generation into transformation.

This is preview API, disabled by default

To try the examples below in JDK 22 you must enable preview features as follows:

- Compile the program with `javac --release 22 --enable-preview Main.java` and run it with `java --enable-preview Main`; or,
- When using the source code launcher, run the program with `java --source 22 --enable-preview Main.java`

Parsing class files with patterns

ASM's streaming view of class files is visitor-based. Visitors are bulky and inflexible; the visitor pattern is often characterized as a library workaround for the lack of pattern matching in a language. Now that the Java language has pattern matching we can express things more directly and concisely. For example, if we want to traverse a `Code` attribute and collect dependencies for a class dependency graph then we can simply iterate through the instructions and match on the ones we find interesting. A `CodeModel` describes a `Code` attribute; we can iterate over its `CodeElements` and handle those that include symbolic references to other types:

```
CodeModel code = ...
Set<ClassDesc> deps = new HashSet<>();
for (CodeElement e : code) {
    switch (e) {
        case FieldInstruction f  -> deps.add(f.owner());
        case InvokeInstruction i -> deps.add(i.owner());
        ... and so on for instanceof, cast, etc ...
    }
}
```

Generating class files with builders

Suppose we wish to generate the following method in a class file:

```
void fooBar(boolean z, int x) {
    if (z)
        foo(x);
    else
        bar(x);
}
```

With ASM we could generate the method as follows:

```
ClassWriter classWriter = ...;
MethodVisitor mv = classWriter.visitMethod(0, "fooBar", "(ZI)V", null, null);
mv.visitCode();
mv.visitVarInsn(ILOAD, 1);
Label label1 = new Label();
mv.visitJumpInsn(IFEQ, label1);
mv.visitVarInsn(ALOAD, 0);
mv.visitVarInsn(ILOAD, 2);
mv.visitMethodInsn(INVOKEVIRTUAL, "Foo", "foo", "(I)V", false);
Label label2 = new Label();
mv.visitJumpInsn(GOTO, label2);
mv.visitLabel(label1);
mv.visitVarInsn(ALOAD, 0);
mv.visitVarInsn(ILOAD, 2);
mv.visitMethodInsn(INVOKEVIRTUAL, "Foo", "bar", "(I)V", false);
mv.visitLabel(label2);
mv.visitInsn(RETURN);
mv.visitEnd();
```

The `MethodVisitor` in ASM doubles as both a visitor and a builder. Clients can create a `ClassWriter` directly and then can ask the `ClassWriter` for a `MethodVisitor`. The Class-File API inverts this idiom: Instead of the client creating a builder with a constructor or factory, the client provides a lambda which accepts a builder:

```
ClassBuilder classBuilder = ...;
classBuilder.withMethod("fooBar", MethodTypeDesc.of(CD_void, CD_boolean, CD_int), flags,
                        methodBuilder -> methodBuilder.withCode(codeBuilder -> {
    Label label1 = codeBuilder.newLabel();
    Label label2 = codeBuilder.newLabel();
    codeBuilder.iload(1)
        .ifeq(label1)
        .aload(0)
        .iload(2)
        .invokevirtual(ClassDesc.of("Foo"), "foo", MethodTypeDesc.of(CD_void, CD_int))
        .goto_(label2)
        .labelBinding(label1)
        .aload(0)
        .iload(2)
        .invokevirtual(ClassDesc.of("Foo"), "bar", MethodTypeDesc.of(CD_void, CD_int))
        .labelBinding(label2);
        .return_();
});
```

This is more specific and transparent — the builder has lots of convenience methods such as `aload(n)` — but not yet any more concise or higher-level. Yet there is already a powerful hidden benefit: By capturing the sequence of operations in a lambda we get the possibility of *replay*, which enables the library to do work that previously the client had to do. For example, branch offsets can be either short or long. If clients generate instructions imperatively then they have to compute the size of each branch's offset when generating the branch, which is complex and error prone. But if the client provides a lambda that takes a builder then the library can optimistically try to generate the method with short offsets and, if that fails, discard the generated state and re-invoke the lambda with different code generation parameters.

Decoupling builders from visitation also lets us provide higher-level conveniences to manage block scoping and local-variable index calculation, and allows us to eliminate manual label management and branching:

```
CodeBuilder classBuilder = ...;
classBuilder.withMethod("fooBar", MethodTypeDesc.of(CD_void, CD_boolean, CD_int), flags,
                        methodBuilder -> methodBuilder.withCode(codeBuilder -> {
    codeBuilder.iload(codeBuilder.parameterSlot(0))
               .ifThenElse(
                   b1 -> b1.aload(codeBuilder.receiverSlot())
                           .iload(codeBuilder.parameterSlot(1))
                           .invokevirtual(ClassDesc.of("Foo"), "foo",
                                          MethodTypeDesc.of(CD_void, CD_int)),
                   b2 -> b2.aload(codeBuilder.receiverSlot())
                           .iload(codeBuilder.parameterSlot(1))
                           .invokevirtual(ClassDesc.of("Foo"), "bar",
                                          MethodTypeDesc.of(CD_void, CD_int))
               .return_();
});
```

Because block scoping is managed by the Class-File API, we did not have to generate labels or branch instructions — they are inserted for us. Similarly, the Class-File API can optionally manage block-scoped allocation of local variables, freeing clients of the bookkeeping of local-variable slots as well.

Transforming class files

The parsing and generation methods in the Class-File API line up so that transformation is seamless. The parsing example above traversed a sequence of `CodeElements`, letting the client match against the individual elements. The builder accepts `CodeElements` so that typical transformation idioms fall out naturally.

Suppose we want to process a class file and keep everything unchanged except for removing methods whose names start with "debug". We would get a `ClassModel`, create a `ClassBuilder`, iterate the elements of the original `ClassModel`, and pass all of them through to the builder except for the methods we want to drop:

```
ClassFile cf = ClassFile.of();
ClassModel classModel = cf.parse(bytes);
byte[] newBytes = cf.build(classModel.thisClass().asSymbol(),
        classBuilder -> {
            for (ClassElement ce : classModel) {
                if (!(ce instanceof MethodModel mm
                        && mm.methodName().stringValue().startsWith("debug"))) {
                    classBuilder.with(ce);
                }
            }
        });
```

Transforming method bodies is slightly more complicated since we have to explode classes into their parts (fields, methods, and attributes), select the method elements, explode the method elements into their parts (including the code attribute), and then explode the code attribute into its elements (i.e., instructions). The following transformation swaps invocations of methods on class `Foo` to invocations of methods on class `Bar`:

```
ClassFile cf = ClassFile.of();
ClassModel classModel = cf.parse(bytes);
byte[] newBytes = cf.build(classModel.thisClass().asSymbol(),
        classBuilder -> {
            for (ClassElement ce : classModel) {
                if (ce instanceof MethodModel mm) {
                    classBuilder.withMethod(mm.methodName(), mm.methodType(),
                            mm.flags().flagsMask(), methodBuilder -> {
                                for (MethodElement me : mm) {
                                    if (me instanceof CodeModel codeModel) {
                                        methodBuilder.withCode(codeBuilder -> {
                                            for (CodeElement e : codeModel) {
                                                switch (e) {
                                                    case InvokeInstruction i
                                                            when i.owner().asInternalName().equals("Foo")) ->
                                                        codeBuilder.invokeInstruction(i.opcode(),
                                                                                      ClassDesc.of("Bar"),
                                                                                      i.name(), i.type());
                                                        default -> codeBuilder.with(e);
                                                }
                                            }
                                        });
                                    }
                                    else
                                        methodBuilder.with(me);
                                }
                            });
                }
                else
                    classBuilder.with(ce);
            }
        });
```

Navigating the class-file tree by exploding entities into elements and examining each element involves some boilerplate which is repeated at multiple levels. This idiom is common to all traversals, so it is something the library should help with. The common pattern of taking a class-file entity, obtaining a corresponding builder, examining each element of the entity and possibly replacing it with other elements can be expressed by *transforms*, which are applied by *transformation methods*.

A transform accepts a builder and an element. It either replaces the element with other elements, drops the element, or passes the element through to the builder. Transforms are functional interfaces, so transformation logic can be captured in lambdas.

A transformation method copies the relevant metadata (names, flags, etc.) from a composite element to a builder and then processes the composite's elements by applying a transform, handling the repetitive exploding and iteration.

Using transformation we can rewrite the previous example as:

```
ClassFile cf = ClassFile.of();
ClassModel classModel = cf.parse(bytes);
byte[] newBytes = cf.transform(classModel, (classBuilder, ce) -> {
    if (ce instanceof MethodModel mm) {
        classBuilder.transformMethod(mm, (methodBuilder, me)-> {
            if (me instanceof CodeModel cm) {
                methodBuilder.transformCode(cm, (codeBuilder, e) -> {
                    switch (e) {
                        case InvokeInstruction i
                                when i.owner().asInternalName().equals("Foo") ->
                            codeBuilder.invokeInstruction(i.opcode(), ClassDesc.of("Bar"),
                                                          i.name().stringValue(),
                                                          i.typeSymbol(), i.isInterface());
                            default -> codeBuilder.with(e);
                    }
                });
            }
            else
                methodBuilder.with(me);
        });
    }
    else
        classBuilder.with(ce);
});
```

The iteration boilerplate is gone, but the deep nesting of lambdas to access the instructions is still intimidating. We can simplify this by factoring out the instruction-specific activity into a `CodeTransform`:

```
CodeTransform codeTransform = (codeBuilder, e) -> {
    switch (e) {
        case InvokeInstruction i when i.owner().asInternalName().equals("Foo") ->
            codeBuilder.invokeInstruction(i.opcode(), ClassDesc.of("Bar"),
                                          i.name().stringValue(),
                                          i.typeSymbol(), i.isInterface());
        default -> codeBuilder.accept(e);
    }
};
```

We can then *lift* this transform on code elements into a transform on method elements. When the lifted transform sees a `Code` attribute, it transforms it with the code transform, passing all other method elements through unchanged:

```
MethodTransform methodTransform = MethodTransform.transformingCode(codeTransform);
```

We can do the same again to lift the resulting transform on method elements into a transform on class elements:

```
ClassTransform classTransform = ClassTransform.transformingMethods(methodTransform);
```

Now our example becomes simply:

```
ClassFile cf = ClassFile.of();
byte[] newBytes = cf.transform(cf.parse(bytes), classTransform);
```

Testing

The Class-File API has a large surface area and must generate classes in conformance with the Java Virtual Machine Specification, so significant quality and conformance testing will be required. Further, to the degree that we replace uses of ASM in the JDK with uses of the Class-File API, we will compare the results of using both libraries to detect regressions, and do extensive performance testing to detect and avoid performance regressions.

Alternatives

An obvious idea is to "just" merge ASM into the JDK and take on responsibility for its ongoing maintenance, but this is not the right choice. ASM is an old code base with lots of legacy baggage. It is difficult to evolve, and the design priorities that informed its architecture are likely not what we would choose today. Moreover, the Java language has improved substantially since ASM was created, so what might have been the best API idioms in 2002 may not be ideal two decades later.