Module java.base

Package java.lang.classfile

package java.lang.classfile

`java.lang.classfile` is a preview API of the Java platform.
Programs can only use `java.lang.classfile` when preview features are enabled.
Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.

Provides classfile parsing, generation, and transformation library.

The `java.lang.classfile` package contains classes for reading, writing, and modifying Java class files, as specified in Chapter 4 of the *Java Java Virtual Machine Specification*.

Reading classfiles

The main class for reading classfiles is `ClassModel`[PREVIEW]; we convert bytes into a `ClassModel`[PREVIEW] with `ClassFile.parse(byte[])`[PREVIEW]:

```
ClassModel cm = ClassFile.of().parse(bytes);
```

There are several additional overloads of `parse` that let you specify various processing options.

A `ClassModel`[PREVIEW] is an immutable description of a class file. It provides accessor methods to get at class metadata (e.g., `ClassModel.thisClass()`[PREVIEW], `ClassModel.flags()`[PREVIEW]), as well as subordinate classfile entities (`ClassModel.fields()`[PREVIEW], `AttributedElement.attributes()`[PREVIEW]). A `ClassModel`[PREVIEW] is inflated lazily; most parts of the classfile are not parsed until they are actually needed.

We can enumerate the names of the fields and methods in a class by:

```
ClassModel cm = ClassFile.of().parse(bytes);
for (FieldModel fm : cm.fields())
    System.out.printf("Field %s%n", fm.fieldName().stringValue());
```

```
for (MethodModel mm : cm.methods())
    System.out.printf("Method %s%n", mm.methodName().stringValue());
```

When we enumerate the methods, we get a `MethodModel`[PREVIEW] for each method; like a `ClassModel`, it gives us access to method metadata and the ability to descend into subordinate entities such as the bytecodes of the method body. In this way, a `ClassModel` is the root of a tree, with children for fields, methods, and attributes, and `MethodModel` in turn has its own children (attributes, `CodeModel`, etc.)

Methods like `ClassModel.methods()`[PREVIEW] allows us to traverse the class structure explicitly, going straight to the parts we are interested in. This is useful for certain kinds of analysis, but if we wanted to process the whole classfile, we may want something more organized. A `ClassModel`[PREVIEW] also provides us with a view of the classfile as a series of class *elements*, which may include methods, fields, attributes, and more, and which can be distinguished with pattern matching. We could rewrite the above example as:

```
ClassModel cm = ClassFile.of().parse(bytes);
for (ClassElement ce : cm) {
    switch (ce) {
        case MethodModel mm -> System.out.printf("Method %s%n", mm.methodName().stringValue());
        case FieldModel fm -> System.out.printf("Field %s%n", fm.fieldName().stringValue());
        default -> { }
    }
}
```

The models returned as elements from traversing `ClassModel` can in turn be sources of elements. If we wanted to traverse a classfile and enumerate all the classes for which we access fields and methods, we can pick out the class elements that describe methods, then in turn pick out the method elements that describe the code attribute, and finally pick out the code elements that describe field access and invocation instructions:

```
ClassModel cm = ClassFile.of().parse(bytes);
Set<ClassDesc> dependencies = new HashSet<>();

for (ClassElement ce : cm) {
    if (ce instanceof MethodModel mm) {
        for (MethodElement me : mm) {
            if (me instanceof CodeModel xm) {
                for (CodeElement e : xm) {
                    switch (e) {
                        case InvokeInstruction i -> dependencies.add(i.owner().asSymbol());
```

```
                        case FieldInstruction i -> dependencies.add(i.owner().asSymbol());
                        default -> { }
                    }
                }
            }
        }
    }
}
```

This same query could alternately be processed as a stream pipeline over class elements:

```
ClassModel cm = ClassFile.of().parse(bytes);
Set<ClassDesc> dependencies =
      cm.elementStream()
        .flatMap(ce -> ce instanceof MethodModel mm ? mm.elementStream() : Stream.empty())
        .flatMap(me -> me instanceof CodeModel com ? com.elementStream() : Stream.empty())
        .<ClassDesc>mapMulti((xe, c) -> {
            switch (xe) {
                case InvokeInstruction i -> c.accept(i.owner().asSymbol());
                case FieldInstruction i -> c.accept(i.owner().asSymbol());
                default -> { }
            }
        })
        .collect(toSet());
```

Models and elements

The view of classfiles presented by this API is framed in terms of *models* and *elements*. Models represent complex structures, such as classes, methods, fields, record elements, or the code body of a method. Models can be explored either via random-access navigation (such as the `ClassModel.methods()`[PREVIEW] accessor) or as a linear sequence of *elements*. (Elements can in turn also be models; a `FieldModel`[PREVIEW] is also an element of a class.) For each model type (e.g., `MethodModel`[PREVIEW]), there is a corresponding element type (`MethodElement`[PREVIEW]). Models and elements are immutable and are inflated lazily so creating a model does not necessarily require processing its entire content.

The constant pool

Much of the interesting content in a classfile lives in the *constant pool*. `ClassModel`[PREVIEW] provides a lazily-inflated, read-only view of the constant pool via `ClassModel.constantPool()`[PREVIEW]. Descriptions of classfile content is often exposed in the form of various subtypes of `PoolEntry`[PREVIEW], such as `ClassEntry`[PREVIEW] or `Utf8Entry`[PREVIEW].

Constant pool entries are also exposed through models and elements; in the above traversal example, the `InvokeInstruction`[PREVIEW] element exposed a method for `owner` that corresponds to a `Constant_Class_info` entry in the constant pool.

Attributes

Much of the contents of a classfile is stored in attributes; attributes are found on classes, methods, fields, record components, and on the `Code` attribute. Most attributes are surfaced as elements; for example, `SignatureAttribute`[PREVIEW] is a `ClassElement`[PREVIEW], `MethodElement`[PREVIEW], and `FieldElement`[PREVIEW] since it can appear in all of those places, and is included when iterating the elements of the corresponding model.

Some attributes are not surfaced as elements; these are attributes that are tightly coupled to -- and logically part of -- other parts of the class file. These include the `BootstrapMethods`, `LineNumberTable`, `StackMapTable`, `LocalVariableTable`, and `LocalVariableTypeTable` attributes. These are processed by the library and treated as part of the structure they are coupled to (the entries of the `BootstrapMethods` attribute are treated as part of the constant pool; line numbers and local variable metadata are modeled as elements of `CodeModel`[PREVIEW].)

The `Code` attribute, in addition to being modeled as a `MethodElement`[PREVIEW], is also a model in its own right (`CodeModel`[PREVIEW]) due to its complex structure.

Each standard attribute has an interface (in `java.lang.classfile.attribute`) which exposes the contents of the attribute and provides factories to construct the attribute. For example, the `Signature` attribute is defined by the `SignatureAttribute`[PREVIEW] class, and provides accessors for `SignatureAttribute.signature()`[PREVIEW] as well as factories taking `Utf8Entry`[PREVIEW] or `String`.

Custom attributes

Attributes are converted between their classfile form and their corresponding object form via an `AttributeMapper`[PREVIEW]. An `AttributeMapper` provides the `AttributeMapper.readAttribute(AttributedElement, ClassReader, int)`[PREVIEW] method for mapping from the classfile format to an attribute instance, and the `AttributeMapper.writeAttribute(java.lang.classfile.BufWriter, java.lang.Object)`[PREVIEW] method for mapping back to the classfile format. It also contains metadata including the attribute name, the set of classfile entities where the attribute is applicable, and whether multiple attributes of the same kind are allowed on a single entity.

There are built-in attribute mappers (in `Attributes`[PREVIEW]) for each of the attribute types defined in section 4.7 of *The Java Virtual Machine Specification*, as well as several common nonstandard attributes used by the JDK such as `CharacterRangeTable`.

Unrecognized attributes are delivered as elements of type `UnknownAttribute`[PREVIEW], which provide access only to the `byte[]` contents of the attribute.

For nonstandard attributes, user-provided attribute mappers can be specified through the use of the `ClassFile.AttributeMapperOption.of(java.util.function.Function)`[PREVIEW]} classfile option. Implementations of custom attributes should extend `CustomAttribute`[PREVIEW].

Options

`ClassFile.of(java.lang.classfile.ClassFile.Option[])`[PREVIEW] accepts a list of options. `ClassFile.Option`[PREVIEW] is a base interface for some statically enumerated options, as well as factories for more complex options, including:

- `ClassFile.StackMapsOption`[PREVIEW] -- generate stackmaps (default is `STACK_MAPS_WHEN_REQUIRED`)
- `ClassFile.DebugElementsOption`[PREVIEW] -- processing of debug information, such as local variable metadata (default is `PASS_DEBUG`)
- `ClassFile.LineNumbersOption`[PREVIEW] -- processing of line numbers (default is `PASS_LINE_NUMBERS`)
- `ClassFile.AttributesProcessingOption`[PREVIEW] -- unrecognized or problematic original attributes (default is `PASS_ALL_ATTRIBUTES`)
- `ClassFile.ConstantPoolSharingOption`[PREVIEW]} -- share constant pool when transforming (default is `SHARED_POOL`)
- `ClassFile.ClassHierarchyResolverOption.of(java.lang.classfile.ClassHierarchyResolver)`[PREVIEW] -- specify a custom class hierarchy resolver used by stack map generation
- `ClassFile.AttributeMapperOption.of(java.util.function.Function)`[PREVIEW] -- specify format of custom attributes

Most options allow you to request that certain parts of the classfile be skipped during traversal, such as debug information or unrecognized attributes. Some options allow you to suppress generation of portions of the classfile, such as stack maps. Many of these options are to access performance tradeoffs; processing debug information and line numbers has a cost (both in writing and reading.) If you don't need this information, you can suppress it with options to gain some performance.

Writing classfiles

ClassFile generation is accomplished through *builders*. For each entity type that has a model, there is also a corresponding builder type; classes are built through `ClassBuilder`[PREVIEW], methods through `MethodBuilder`[PREVIEW], etc.

Rather than creating builders directly, builders are provided as an argument to a user-provided lambda. To generate the familiar "hello world" program, we ask for a class builder, and use that class builder to create method builders for the constructor and `main` method, and in turn use the method builders to create a `Code` attribute and use the code builders to generate the instructions:

```
byte[] bytes = ClassFile.of().build(CD_Hello,
        clb -> clb.withFlags(ClassFile.ACC_PUBLIC)
                  .withMethod(ConstantDescs.INIT_NAME, ConstantDescs.MTD_void,
                              ClassFile.ACC_PUBLIC,
                              mb -> mb.withCode(
                                      cob -> cob.aload(0)
                                                .invokespecial(ConstantDescs.CD_Object,
```

```
                                                        ConstantDescs.INIT_NAME, ConstantDescs.MTD_void)
                                        .return_()))
                .withMethod("main", MTD_void_StringArray, ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
                            mb -> mb.withCode(
                                    cob -> cob.getstatic(CD_System, "out", CD_PrintStream)
                                              .ldc("Hello World")
                                              .invokevirtual(CD_PrintStream, "println", MTD_void_String)
                                              .return_())));
```

The convenience methods `ClassBuilder.buildMethodBody` allows us to ask `ClassBuilder`[PREVIEW] to create code builders to build method bodies directly, skipping the method builder custom lambda:

```
byte[] bytes = ClassFile.of().build(CD_Hello,
        clb -> clb.withFlags(ClassFile.ACC_PUBLIC)
                  .withMethodBody(ConstantDescs.INIT_NAME, ConstantDescs.MTD_void,
                                  ClassFile.ACC_PUBLIC,
                                  cob -> cob.aload(0)
                                            .invokespecial(ConstantDescs.CD_Object,
                                                           ConstantDescs.INIT_NAME, ConstantDescs.MTD_void)
                                            .return_())
                  .withMethodBody("main", MTD_void_StringArray, ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
                                  cob -> cob.getstatic(CD_System, "out", CD_PrintStream)
                                            .ldc("Hello World")
                                            .invokevirtual(CD_PrintStream, "println", MTD_void_String)
                                            .return_()));
```

Builders often support multiple ways of expressing the same entity at different levels of abstraction. For example, the `invokevirtual` instruction invoking `println` could have been generated with `CodeBuilder.invokevirtual`[PREVIEW], `CodeBuilder.invokeInstruction`[PREVIEW], or `CodeBuilder.with`[PREVIEW].

The convenience method `CodeBuilder.invokevirtual` behaves as if it calls the convenience method `CodeBuilder.invokeInstruction`, which in turn behaves as if it calls method `CodeBuilder.with`. This composing of method calls on the builder enables the composing of transforms (as described later).

Symbolic information

To describe symbolic information for classes and types, the API uses the nominal descriptor abstractions from `java.lang.constant` such as `ClassDesc` and `MethodTypeDesc`, which is less error-prone than using raw strings.

If a constant pool entry has a nominal representation then it provides a method returning the corresponding nominal descriptor type e.g. method `ClassEntry.asSymbol()`[PREVIEW] returns `ClassDesc`.

Where appropriate builders provide two methods for building an element with symbolic information, one accepting nominal descriptors, and the other accepting constant pool entries.

Consistency checks, syntax checks and verification

No consistency checks are performed while building or transforming classfiles (except for null arguments checks). All builders and classfile elements factory methods accepts the provided information without implicit validation. However, fatal inconsistencies (like for example invalid code sequence or unresolved labels) affects internal tools and may cause exceptions later in the classfile building process.

Using nominal descriptors assures the right serial form is applied by the ClassFile API library based on the actual context. Also these nominal descriptors are validated during their construction, so it is not possible to create them with invalid content by mistake. Following example pass class name to the `ClassDesc.of(java.lang.String)` method for validation and the library performs automatic conversion to the right internal form of the class name when serialized in the constant pool as a class entry.

```
var validClassEntry = constantPoolBuilder.classEntry(ClassDesc.of("mypackage.MyClass"));
```

On the other hand it is possible to use builders methods and factories accepting constant pool entries directly. Constant pool entries can be constructed also directly from raw values, with no additional conversions or validations. Following example uses intentionally wrong class name form and it is applied without any validation or conversion.

```
var invalidClassEntry = constantPoolBuilder.classEntry(
                            constantPoolBuilder.utf8Entry("mypackage.MyClass"));
```

More complex verification of a classfile can be achieved by invocation of `ClassFile.verify(java.lang.classfile.ClassModel)`[PREVIEW].

Transforming classfiles

ClassFile Processing APIs are most frequently used to combine reading and writing into transformation, where a classfile is read, localized changes are made, but much of the classfile is passed through unchanged. For each kind of builder, `XxxBuilder` has a method `with(XxxElement)` so that elements that we wish to pass through unchanged can be handed directly back to the builder.

If we wanted to strip out methods whose names starts with "debug", we could get an existing `ClassModel`[PREVIEW], build a new classfile that provides a `ClassBuilder`[PREVIEW], iterate the elements of the original `ClassModel`[PREVIEW], and pass through all of them to the builder except the methods we want to drop:

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

This hands every class element, except for those corresponding to methods whose names start with `debug`, back to the builder. Transformations can of course be more complicated, diving into method bodies and instructions and transforming those as well, but the same structure is repeated at every level, since every entity has corresponding model, builder, and element abstractions.

Transformation can be viewed as a "flatMap" operation on the sequence of elements; for every element, we could pass it through unchanged, drop it, or replace it with one or more elements. Because transformation is such a common operation on classfiles, each model type has a corresponding `XxxTransform` type (which describes a transform on a sequence of `XxxElement`) and each builder type has `transformYyy` methods for transforming its child models. A transform is simply a functional interface that takes a builder and an element, and an implementation "flatMap"s elements into the builder. We could express the above as:

```
ClassTransform ct = (builder, element) -> {
    if (!(element instanceof MethodModel mm && mm.methodName().stringValue().startsWith("debug")))
        builder.with(element);
};
var cc = ClassFile.of();
byte[] newBytes = cc.transform(cc.parse(bytes), ct);
```

`ClassTransform.dropping` convenience method allow us to simplify the same transformation construction and express the above as:

```
ClassTransform ct = ClassTransform.dropping(
                           element -> element instanceof MethodModel mm
                                   && mm.methodName().stringValue().startsWith("debug"));
```

Lifting transforms

While the example using transformations are only slightly shorter, the advantage of expressing transformation in this way is that the transform operations can be more easily combined. Suppose we want to redirect invocations of static methods on `Foo` to the corresponding method on `Bar` instead. We could express this as a transformation on `CodeElement`[PREVIEW]:

```
CodeTransform fooToBar = (b, e) -> {
    if (e instanceof InvokeInstruction i
            && i.owner().asInternalName().equals("Foo")
            && i.opcode() == Opcode.INVOKESTATIC)
                b.invokeInstruction(i.opcode(), CD_Bar, i.name().stringValue(), i.typeSymbol(), i.isInterface());
    else b.with(e);
};
```

We can then *lift* this transformation on code elements into a transformation on method elements. This intercepts method elements that correspond to a `Code` attribute, dives into its code elements, and applies the code transform to them, and passes other method elements through unchanged:

```
MethodTransform mt = MethodTransform.transformingCode(fooToBar);
```

and further lift the transform on method elements into one on class elements:

```
ClassTransform ct = ClassTransform.transformingMethods(mt);
```

or lift the code transform into the class transform directly:

```
ClassTransform ct = ClassTransform.transformingMethodBodiess(fooToBar);
```

and then transform the classfile:

```
var cc = ClassFile.of();
byte[] newBytes = cc.transform(cc.parse(bytes), ct);
```

This is much more concise (and less error-prone) than the equivalent expressed by traversing the classfile structure directly:

```
byte[] newBytes = ClassFile.of().build(classModel.thisClass().asSymbol(),
    classBuilder -> {
```

```
      for (ClassElement ce : classModel) {
          if (ce instanceof MethodModel mm) {
              classBuilder.withMethod(mm.methodName().stringValue(), mm.methodTypeSymbol(),
                                      mm.flags().flagsMask(),
                                      methodBuilder -> {
                          for (MethodElement me : mm) {
                              if (me instanceof CodeModel xm) {
                                  methodBuilder.withCode(codeBuilder -> {
                                      for (CodeElement e : xm) {
                                          if (e instanceof InvokeInstruction i && i.owner().asInternalName().equals("I
                                                                              && i.opcode() == Opcode.INVOKESTATIC)
                                                      codeBuilder.invokeInstruction(i.opcode(), CD_Bar,
                                                                                    i.name().stringValue(), i.typeSymb
                                          else codeBuilder.with(e);
                                      }});
                                  }
                                  else
                                  methodBuilder.with(me);
                              }
                          });
                      }
              else
              classBuilder.with(ce);
          }
```

Composing transforms

Transforms on the same type of element can be composed in sequence, where the output of the first is fed to the input of the second. Suppose we want to instrument all method calls, where we print the name of a method before calling it:

```
CodeTransform instrumentCalls = (b, e) -> {
    if (e instanceof InvokeInstruction i) {
        b.getstatic(CD_System, "out", CD_PrintStream)
         .ldc(i.name().stringValue())
         .invokevirtual(CD_PrintStream, "println", MTD_void_String);
```

```
    }
    b.with(e);
};
```

Then we can compose `fooToBar` and `instrumentCalls` with `CodeTransform.andThen(java.lang.classfile.CodeTransform)`[PREVIEW]:

```
var cc = ClassFile.of();
byte[] newBytes = cc.transform(cc.parse(bytes),
                               ClassTransform.transformingMethods(
                                   MethodTransform.transformingCode(
                                       fooToBar.andThen(instrumentCalls))));
```

Transform `instrumentCalls` will receive all code elements produced by transform `forToBar`, either those code elements from the original classfile or replacements (replacing static invocations to `Foo` with those to `Bar`).

Constant pool sharing

Transformation doesn't merely handle the logistics of reading, transforming elements, and writing. Most of the time when we are transforming a classfile, we are making relatively minor changes. To optimize such cases, transformation seeds the new classfile with a copy of the constant pool from the original classfile; this enables significant optimizations (methods and attributes that are not transformed can be processed by bulk-copying their bytes, rather than parsing them and regenerating their contents.) If constant pool sharing is not desired it can be suppressed with the `ClassFile.ConstantPoolSharingOption`[PREVIEW] option. Such suppression may be beneficial when transformation removes many elements, resulting in many unreferenced constant pool entries.

Transformation handling of unknown classfile elements

Custom classfile transformations might be unaware of classfile elements introduced by future JDK releases. To achieve deterministic stability, classfile transforms interested in consuming all classfile elements should be implemented strictly to throw exceptions if running on a newer JDK, if the transformed class file is a newer version, or if a new and unknown classfile element appears. As for example in the following strict compatibility-checking transformation snippets:

```
CodeTransform fooToBar = (b, e) -> {
    if (ClassFile.latestMajorVersion() > ClassFile.JAVA_22_VERSION) {
        throw new IllegalArgumentException("Cannot run on JDK > 22");
    }
    switch (e) {
        case ArrayLoadInstruction i -> doSomething(b, i);
        case ArrayStoreInstruction i -> doSomething(b, i);
```

```
        default ->  b.with(e);
    }
};
```

```
ClassTransform fooToBar = (b, e) -> {
    switch (e) {
        case ClassFileVersion v when v.majorVersion() > ClassFile.JAVA_22_VERSION ->
            throw new IllegalArgumentException("Cannot transform class file version " + v.majorVersion());
        default ->  doSomething(b, e);
    }
};
```

```
CodeTransform fooToBar = (b, e) -> {
    switch (e) {
        case ArrayLoadInstruction i -> doSomething(b, i);
        case ArrayStoreInstruction i -> doSomething(b, i);
        case BranchInstruction i -> doSomething(b, i);
        case ConstantInstruction i -> doSomething(b, i);
        case ConvertInstruction i -> doSomething(b, i);
        case DiscontinuedInstruction i -> doSomething(b, i);
        case FieldInstruction i -> doSomething(b, i);
        case InvokeDynamicInstruction i -> doSomething(b, i);
        case InvokeInstruction i -> doSomething(b, i);
        case LoadInstruction i -> doSomething(b, i);
        case StoreInstruction i -> doSomething(b, i);
        case IncrementInstruction i -> doSomething(b, i);
        case LookupSwitchInstruction i -> doSomething(b, i);
        case MonitorInstruction i -> doSomething(b, i);
        case NewMultiArrayInstruction i -> doSomething(b, i);
        case NewObjectInstruction i -> doSomething(b, i);
        case NewPrimitiveArrayInstruction i -> doSomething(b, i);
        case NewReferenceArrayInstruction i -> doSomething(b, i);
        case NopInstruction i -> doSomething(b, i);
        case OperatorInstruction i -> doSomething(b, i);
        case ReturnInstruction i -> doSomething(b, i);
```

```
        case StackInstruction i -> doSomething(b, i);
        case TableSwitchInstruction i -> doSomething(b, i);
        case ThrowInstruction i -> doSomething(b, i);
        case TypeCheckInstruction i -> doSomething(b, i);
        case PseudoInstruction i ->  doSomething(b, i);
        default ->
            throw new IllegalArgumentException("An unknown instruction could not be handled by this transformation");
    }
```

Conversely, classfile transforms that are only interested in consuming a portion of classfile elements do not need to concern with new and unknown classfile elements and may pass them through. Following example shows such future-proof code transformation:

```
CodeTransform fooToBar = (b, e) -> {
    switch (e) {
        case ArrayLoadInstruction i -> doSomething(b, i);
        case ArrayStoreInstruction i -> doSomething(b, i);
        default ->  b.with(e);
    }
};
```

API conventions

The API is largely derived from a *data model* for the classfile format, which defines each element kind (which includes models and attributes) and its properties. For each element kind, there is a corresponding interface to describe that element, and factory methods to create that element. Some element kinds also have convenience methods on the corresponding builder (e.g., `CodeBuilder.invokevirtual(java.lang.constant.ClassDesc, java.lang.String, java.lang.constant.MethodTypeDesc)`[PREVIEW]).

Most symbolic information in elements is represented by constant pool entries (for example, the owner of a field is represented by a `ClassEntry`[PREVIEW].) Factories and builders also accept nominal descriptors from `java.lang.constant` (e.g., `ClassDesc`.)

Data model

```
ClassElement =
    FieldModel*(UtfEntry name, Utf8Entry descriptor)
```

```
    | MethodModel*(UtfEntry name, Utf8Entry descriptor)
    | ModuleAttribute?(int flags, ModuleEntry moduleName, UtfEntry moduleVersion,
                       List<ModuleRequireInfo> requires, List<ModuleOpenInfo> opens,
                       List<ModuleExportInfo> exports, List<ModuleProvidesInfo> provides,
                       List<ClassEntry> uses)
    | ModulePackagesAttribute?(List<PackageEntry> packages)
    | ModuleTargetAttribute?(Utf8Entry targetPlatform)
    | ModuleHashesAttribute?(Utf8Entry algorithm, List<HashInfo> hashes)
    | ModuleResolutionAttribute?(int resolutionFlags)
    | SourceFileAttribute?(Utf8Entry sourceFile)
    | SourceDebugExtensionsAttribute?(byte[] contents)
    | CompilationIDAttribute?(Utf8Entry compilationId)
    | SourceIDAttribute?(Utf8Entry sourceId)
    | NestHostAttribute?(ClassEntry nestHost)
    | NestMembersAttribute?(List<ClassEntry> nestMembers)
    | RecordAttribute?(List<RecordComponent> components)
    | EnclosingMethodAttribute?(ClassEntry className, NameAndTypeEntry method)
    | InnerClassesAttribute?(List<InnerClassInfo> classes)
    | PermittedSubclassesAttribute?(List<ClassEntry> permittedSubclasses)
    | DeclarationElement*
```

where `DeclarationElement` are the elements that are common to all declarations (classes, methods, fields) and so are factored out:

```
DeclarationElement =
    SignatureAttribute?(Utf8Entry signature)
    | SyntheticAttribute?()
    | DeprecatedAttribute?()
    | RuntimeInvisibleAnnotationsAttribute?(List<Annotation> annotations)
    | RuntimeVisibleAnnotationsAttribute?(List<Annotation> annotations)
    | CustomAttribute*
    | UnknownAttribute*
```

Fields and methods are models with their own elements. The elements of fields and methods are fairly simple; most of the complexity of methods lives in the `CodeModel`[PREVIEW] (which models the `Code` attribute along with the code-related attributes: stack map table, local variable table, line number table, etc.)

```
FieldElement =
    DeclarationElement
    | ConstantValueAttribute?(ConstantValueEntry constant)

MethodElement =
    DeclarationElement
    | CodeModel?()
    | AnnotationDefaultAttribute?(ElementValue defaultValue)
    | MethodParametersAttribute?(List<MethodParameterInfo> parameters)
    | ExceptionsAttribute?(List<ClassEntry> exceptions)
```

`CodeModel`[PREVIEW] is unique in that its elements are *ordered*. Elements of `Code` include ordinary bytecodes, as well as a number of pseudo-instructions representing branch targets, line number metadata, local variable metadata, and catch blocks.

```
CodeElement = Instruction | PseudoInstruction

Instruction =
    LoadInstruction(TypeKind type, int slot)
    | StoreInstruction(TypeKind type, int slot)
    | IncrementInstruction(int slot, int constant)
    | BranchInstruction(Opcode opcode, Label target)
    | LookupSwitchInstruction(Label defaultTarget, List<SwitchCase> cases)
    | TableSwitchInstruction(Label defaultTarget, int low, int high,
                             List<SwitchCase> cases)
    | ReturnInstruction(TypeKind kind)
    | ThrowInstruction()
    | FieldInstruction(Opcode opcode, FieldRefEntry field)
    | InvokeInstruction(Opcode opcode, MemberRefEntry method, boolean isInterface)
    | InvokeDynamicInstruction(InvokeDynamicEntry invokedynamic)
    | NewObjectInstruction(ClassEntry className)
    | NewReferenceArrayInstruction(ClassEntry componentType)
    | NewPrimitiveArrayInstruction(TypeKind typeKind)
    | NewMultiArrayInstruction(ClassEntry componentType, int dims)
    | ArrayLoadInstruction(Opcode opcode)
    | ArrayStoreInstruction(Opcode opcode)
    | TypeCheckInstruction(Opcode opcode, ClassEntry className)
```

```
    | ConvertInstruction(TypeKind from, TypeKind to)
    | OperatorInstruction(Opcode opcode)
    | ConstantInstruction(ConstantDesc constant)
    | StackInstruction(Opcode opcode)
    | MonitorInstruction(Opcode opcode)
    | NopInstruction()

PseudoInstruction =
    | LabelTarget(Label label)
    | LineNumber(int line)
    | ExceptionCatch(Label tryStart, Label tryEnd, Label handler, ClassEntry exception)
    | LocalVariable(int slot, UtfEntry name, Utf8Entry type, Label startScope, Label endScope)
    | LocalVariableType(int slot, Utf8Entry name, Utf8Entry type, Label startScope, Label endScope)
    | CharacterRange(int rangeStart, int rangeEnd, int flags, Label startScope, Label endScope)
```

Since:
22

Related Packages

Package	Description
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.classfile.attribute[PREVIEW]	**Preview.** Provides interfaces describing classfile attributes for the `java.lang.classfile`[PREVIEW] library.
java.lang.classfile.components[PREVIEW]	**Preview.** Provides specific components, transformations, and tools built on top of the `java.lang.classfile`[PREVIEW] library.
java.lang.classfile.constantpool[PREVIEW]	**Preview.** Provides interfaces describing classfile constant pool entries for the `java.lang.classfile`[PREVIEW] library.
java.lang.classfile.instruction[PREVIEW]	**Preview.** Provides interfaces describing code instructions for the `java.lang.classfile`[PREVIEW] library.

All Classes and Interfaces | Interfaces | Classes | Enum Classes

Class	Description
AccessFlags[PREVIEW]	**Preview.** Models the access flags for a class, method, or field.
Annotation[PREVIEW]	**Preview.** Models an annotation on a declaration.
AnnotationElement[PREVIEW]	**Preview.** Models a key-value pair of an annotation.
AnnotationValue[PREVIEW]	**Preview.** Models the value of a key-value pair of an annotation.
AnnotationValue.OfAnnotation[PREVIEW]	**Preview.** Models an annotation-valued element
AnnotationValue.OfArray[PREVIEW]	**Preview.** Models an array-valued element
AnnotationValue.OfBoolean[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfByte[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfCharacter[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfClass[PREVIEW]	**Preview.** Models a class-valued element

AnnotationValue.OfConstant[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfDouble[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfEnum[PREVIEW]	**Preview.** Models an enum-valued element
AnnotationValue.OfFloat[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfInteger[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfLong[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfShort[PREVIEW]	**Preview.** Models a constant-valued element
AnnotationValue.OfString[PREVIEW]	**Preview.** Models a constant-valued element
Attribute[PREVIEW]<A extends **Attribute**[PREVIEW]<A>>	**Preview.** Models a classfile attribute 4.7.
AttributedElement[PREVIEW]	**Preview.** A `ClassFileElement`[PREVIEW] describing an entity that has attributes, such as a class, field, method, code attribute, or record component.
AttributeMapper[PREVIEW]<A>	**Preview.** Bidirectional mapper between the classfile representation of an attribute and how that attribute is modeled in the API.

AttributeMapper.AttributeStability[PREVIEW]	**Preview.** Attribute stability indicator
Attributes[PREVIEW]	**Preview.** Attribute mappers for standard classfile attributes.
BootstrapMethodEntry[PREVIEW]	**Preview.** Models an entry in the bootstrap method table.
BufWriter[PREVIEW]	**Preview.** Supports writing portions of a classfile to a growable buffer.
ClassBuilder[PREVIEW]	**Preview.** A builder for classfiles.
ClassElement[PREVIEW]	**Preview.** A marker interface for elements that can appear when traversing a `ClassModel`[PREVIEW] or be presented to a `ClassBuilder`[PREVIEW].
ClassFile[PREVIEW]	**Preview.** Represents a context for parsing, transforming, and generating classfiles.
ClassFile.AttributeMapperOption[PREVIEW]	**Preview.** Option describing attribute mappers for custom attributes.
ClassFile.AttributesProcessingOption[PREVIEW]	**Preview.** Option describing whether to process or discard unrecognized or problematic original attributes when a class, record component, field, method or code is transformed in its exploded form.
ClassFile.ClassHierarchyResolverOption[PREVIEW]	**Preview.** Option describing the class hierarchy resolver to use when generating stack maps.

ClassFile.ConstantPoolSharingOption[PREVIEW]	**Preview.** Option describing whether to preserve the original constant pool when transforming a classfile.
ClassFile.DeadCodeOption[PREVIEW]	**Preview.** Option describing whether to patch out unreachable code.
ClassFile.DeadLabelsOption[PREVIEW]	**Preview.** Option describing whether to filter unresolved labels.
ClassFile.DebugElementsOption[PREVIEW]	**Preview.** Option describing whether to process or discard debug elements.
ClassFile.LineNumbersOption[PREVIEW]	**Preview.** Option describing whether to process or discard line numbers.
ClassFile.Option[PREVIEW]	**Preview.** An option that affects the parsing and writing of classfiles.
ClassFile.ShortJumpsOption[PREVIEW]	**Preview.** Option describing whether to automatically rewrite short jumps to long when necessary.
ClassFile.StackMapsOption[PREVIEW]	**Preview.** Option describing whether to generate stackmaps.
ClassFileBuilder[PREVIEW]<E extends **ClassFileElement**[PREVIEW],B extends **ClassFileBuilder**[PREVIEW]<E,B>>	**Preview.** A builder for a classfile or portion of a classfile.
ClassFileElement[PREVIEW]	**Preview.** Immutable model for a portion of (or the entirety of) a classfile.
ClassFileTransform[PREVIEW]<C extends **ClassFileTransform**[PREVIEW]<C,E,B>,E extends **ClassFileElement**[PREVIEW],B extends **ClassFileBuilder**[PREVIEW]<E,B>>	**Preview.** A transformation on streams of elements.

ClassFileTransform.ResolvedTransform[PREVIEW]<E extends **ClassFileElement**[PREVIEW]>	**Preview.** The result of binding a transform to a builder.
ClassFileVersion[PREVIEW]	**Preview.** Models the classfile version information for a class.
ClassHierarchyResolver[PREVIEW]	**Preview.** Provides class hierarchy information for generating correct stack maps during code building.
ClassHierarchyResolver.ClassHierarchyInfo[PREVIEW]	**Preview.** Information about a resolved class.
ClassModel[PREVIEW]	**Preview.** Models a classfile.
ClassReader[PREVIEW]	**Preview.** Supports reading from a classfile.
ClassSignature[PREVIEW]	**Preview.** Models the generic signature of a class file, as defined by 4.7.9.
ClassTransform[PREVIEW]	**Preview.** A transformation on streams of `ClassElement`[PREVIEW].
CodeBuilder[PREVIEW]	**Preview.** A builder for code attributes (method bodies).
CodeBuilder.BlockCodeBuilder[PREVIEW]	**Preview.** A builder for blocks of code.
CodeBuilder.CatchBuilder[PREVIEW]	**Preview.** A builder to add catch blocks.
CodeElement[PREVIEW]	**Preview.**

	A marker interface for elements that can appear when traversing a `CodeModel`[PREVIEW] or be presented to a `CodeBuilder`[PREVIEW].
CodeModel[PREVIEW]	**Preview.** Models the body of a method (the `Code` attribute).
CodeTransform[PREVIEW]	**Preview.** A transformation on streams of `CodeElement`[PREVIEW].
CompoundElement[PREVIEW]<E extends **ClassFileElement**[PREVIEW]>	**Preview.** A `ClassFileElement`[PREVIEW] that has complex structure defined in terms of other classfile elements, such as a method, field, method body, or entire class.
CustomAttribute[PREVIEW]<T extends **CustomAttribute**[PREVIEW]<T>>	**Preview.** Models a non-standard attribute of a classfile.
FieldBuilder[PREVIEW]	**Preview.** A builder for fields.
FieldElement[PREVIEW]	**Preview.** A marker interface for elements that can appear when traversing a `FieldModel`[PREVIEW] or be presented to a `FieldBuilder`[PREVIEW].
FieldModel[PREVIEW]	**Preview.** Models a field.
FieldTransform[PREVIEW]	**Preview.** A transformation on streams of `FieldElement`[PREVIEW].
Instruction[PREVIEW]	**Preview.** Models an executable instruction in a method body.
Interfaces[PREVIEW]	**Preview.** Models the interfaces of a class.

Label[PREVIEW]	**Preview.** A marker for a position within the instructions of a method body.
MethodBuilder[PREVIEW]	**Preview.** A builder for methods.
MethodElement[PREVIEW]	**Preview.** A marker interface for elements that can appear when traversing a `MethodModel`[PREVIEW] or be presented to a `MethodBuilder`[PREVIEW].
MethodModel[PREVIEW]	**Preview.** Models a method.
MethodSignature[PREVIEW]	**Preview.** Models the generic signature of a method, as defined by 4.7.9.
MethodTransform[PREVIEW]	**Preview.** A transformation on streams of `MethodElement`[PREVIEW].
Opcode[PREVIEW]	**Preview.** Describes the opcodes of the JVM instruction set, as described in 6.5.
Opcode.Kind[PREVIEW]	**Preview.** Kinds of opcodes.
PseudoInstruction[PREVIEW]	**Preview.** Models metadata about a `CodeAttribute`[PREVIEW], such as entries in the exception table, line number table, local variable table, or the mapping between instructions and labels.
Signature[PREVIEW]	**Preview.** Models generic Java type signatures, as defined in 4.7.9.1.
Signature.ArrayTypeSig[PREVIEW]	**Preview.**

	Models the signature of an array type.
Signature.BaseTypeSig[PREVIEW]	**Preview.** Models the signature of a primitive type or void
Signature.ClassTypeSig[PREVIEW]	**Preview.** Models the signature of a possibly-parameterized class or interface type.
Signature.RefTypeSig[PREVIEW]	**Preview.** Models the signature of a reference type, which may be a class, interface, type variable, or array type.
Signature.ThrowableSig[PREVIEW]	**Preview.** Models a signature for a throwable type.
Signature.TypeArg[PREVIEW]	**Preview.** Models the type argument.
Signature.TypeArg.WildcardIndicator[PREVIEW]	**Preview.** Indicator for whether a wildcard has default bound, no bound, an upper bound, or a lower bound
Signature.TypeParam[PREVIEW]	**Preview.** Models a signature for a type parameter of a generic class or method.
Signature.TypeVarSig[PREVIEW]	**Preview.** Models the signature of a type variable.
Superclass[PREVIEW]	**Preview.** Models the superclass of a class.
TypeAnnotation[PREVIEW]	**Preview.**

	Models an annotation on a type use, as defined in 4.7.19 and 4.7.20.
TypeAnnotation.CatchTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on the i'th type in an exception parameter declaration.
TypeAnnotation.EmptyTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on either the type in a field declaration, the return type of a method, the type of a newly constructed object, or the receiver type of a method or constructor.
TypeAnnotation.FormalParameterTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on the type in a formal parameter declaration of a method, constructor, or lambda expression.
TypeAnnotation.LocalVarTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on the type in a local variable declaration, including a variable declared as a resource in a try-with-resources statement.
TypeAnnotation.LocalVarTargetInfo[PREVIEW]	**Preview.** Indicates a range of code array offsets within which a local variable has a value, and the index into the local variable array of the current frame at which that local variable can be found.
TypeAnnotation.OffsetTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on either the type in an instanceof expression or a new expression, or the type before the :: in a method reference expression.
TypeAnnotation.SupertypeTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on a type in the extends or implements clause of a class or interface declaration.

TypeAnnotation.TargetInfo[PREVIEW]	**Preview.** Specifies which type in a declaration or expression is being annotated.
TypeAnnotation.TargetType[PREVIEW]	**Preview.** The kind of target on which the annotation appears, as defined in 4.7.20.1.
TypeAnnotation.ThrowsTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on the i'th type in the throws clause of a method or constructor declaration.
TypeAnnotation.TypeArgumentTarget[PREVIEW]	**Preview.** Indicates that an annotation appears either on the i'th type in a cast expression, or on the i'th type argument in the explicit type argument list for any of the following: a new expression, an explicit constructor invocation statement, a method invocation expression, or a method reference expression.
TypeAnnotation.TypeParameterBoundTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on the i'th bound of the j'th type parameter declaration of a generic class, interface, method, or constructor.
TypeAnnotation.TypeParameterTarget[PREVIEW]	**Preview.** Indicates that an annotation appears on the declaration of the i'th type parameter of a generic class, generic interface, generic method, or generic constructor.
TypeAnnotation.TypePathComponent[PREVIEW]	**Preview.** JVMS: Type_path structure identifies which part of the type is annotated, as defined in 4.7.20.2
TypeAnnotation.TypePathComponent.Kind[PREVIEW]	**Preview.** Type path kind, as defined in 4.7.20.2

TypeKind[PREVIEW]	**Preview.** Describes the types that can be part of a field or method descriptor.
WritableElement[PREVIEW]<T>	**Preview.** A classfile element that can encode itself as a stream of bytes in the encoding expected by the classfile format.