Repeating Annotations and Method Parameter Reflection

Alex Buckley
Joe Darcy

2013-09-30

Table of Contents

CHAPTER 1

Repeating Annotations

JAVA SE 5.0 added annotations to the Java programming language, but allowed at most one annotation of a given annotation type to be written on a declaration. In Java SE 8, Oracle proposes to change the Java programming language to allow multiple annotations of a given annotation type to be written on a declaration:

```
@Foo(1) @Foo(2) @Bar
class A {}
```

To respect a pre-existing idiom for representing multiple annotations of a given annotation type, the Java programming language and Java SE platform API jointly assume that multiple such annotations are stored in an array-valued element of a "container annotation". A little setup is required to associate a "repeatable" annotation type with its "containing" annotation type:

```
@Repeatable(FooContainer.class)
@interface Foo { int value(); }

@interface FooContainer { Foo[] value(); }
```

As a result of `Foo` "opting in" to being repeatable, the Java programming language in Java SE 8 accepts multiple `@Foo` annotations on class `A` above. At compile-time, `A` is considered to be annotated by `@FooContainer(value = {@Foo(1),@Foo(2)})` and `@Bar`.

At runtime, the `java.lang.reflect.AnnotatedElement` object which represents class `A` offers new methods that automatically "look through" `@FooContainer` and return the two `@Foo` annotations directly. The methods of `java.lang.reflect.AnnotatedElement` from Java SE 7 are unchanged, so they continue to return the `@FooContainer` annotation which is physically present in the `class` file.

The meta-annotation `@Repeatable` serves two purposes:

- It enables *cardinality control* for annotation types whose authors desire it. Whereas in Java SE 5.0 an annotation could appear on a declaration either zero times or once (given careful use of `@Target` on the annotation type's declaration to limit where it may appear), in Java SE 8 an annotation may appear zero times, once, or more than once if `@Repeatable` is used.
- It ensures *behavioral compatibility* for legacy reflection methods. By storing multiple annotations in a container annotation, the legacy methods - which expect at most one annotation of a given type - never see multiple annotations of a given type in the `class` file. At the same time, new reflection methods which take a repeatable annotation type as a parameter can identify container annotations in the `class` file and "look through" them.

A note on terminology: *The Java Language Specification, Java SE 8 Edition* speaks of annotations being present on a *declaration*, while the Java SE platform API speaks of annotations being present on an *element* (that is, a program element, not an element of an element-value pair in an annotation).

1.1 *The Java Language Specification, Java SE 8 Edition*

9.6 Annotation Types

An annotation type *`T`* is *repeatable* if its declaration is (meta-)annotated with an `@Repeatable` annotation whose `value` element indicates a *containing annotation type of `T`*.

An annotation type *`TC`* is a *containing annotation type of `T`* if all of the following are true:

1. *`TC`* declares a `value()` method whose return type is *`T`*`[]`.
2. Any methods declared by *`TC`* other than `value()` have a default value (§9.6.2).
3. *`TC`* is retained for at least as long as *`T`*, where retention is expressed explicitly or implicitly with the `@Retention` annotation (§9.6.3.2). Specifically:
   - If the retention of *`TC`* is `java.lang.annotation.RetentionPolicy.SOURCE`, then the retention of *`T`* is `java.lang.annotation.RetentionPolicy.SOURCE`.
   - If the retention of *`TC`* is `java.lang.annotation.RetentionPolicy.CLASS`, then the retention of *`T`* is either `java.lang.annotation.RetentionPolicy.CLASS` or `java.lang.annotation.RetentionPolicy.SOURCE`.

- If the retention of *`TC`* is `java.lang.annotation.RetentionPolicy.RUNTIME`, then the retention of *`T`* is `java.lang.annotation.RetentionPolicy.SOURCE`, `java.lang.annotation.RetentionPolicy.CLASS`, or `java.lang.annotation.RetentionPolicy.RUNTIME`.

4. *`T`* is applicable to at least the same kinds of program element as *`TC`* (§9.6.3.1). Specifically, if the kinds of program element where *`T`* is applicable are denoted by the set m_1, and the kinds of program element where *`TC`* is applicable are denoted by the set m_2, then each kind in m_2 must occur in m_1 unless the kind in m_2 is `java.lang.annotation.ElementType.ANNOTATION_TYPE`, in which case at least one of `java.lang.annotation.ElementType.ANNOTATION_TYPE` or `java.lang.annotation.ElementType.TYPE` must occur in m_1.

5. If the declaration of *`T`* has a (meta-)annotation that corresponds to `java.lang.annotation.Documented`, then the declaration of *`TC`* must have a (meta-)annotation that corresponds to `java.lang.annotation.Documented`.

   Note that it is permissible for *`TC`* to be `@Documented` while *`T`* is not `@Documented`.

6. If the declaration of *`T`* has a (meta-)annotation that corresponds to `java.lang.annotation.Inherited`, then the declaration of *`TC`* must have a (meta)-annotation that corresponds to `java.lang.annotation.Inherited`.

   Note that it is permissible for *`TC`* to be `@Inherited` while *`T`* is not `@Inherited`.

The fourth clause implements the policy that an annotation type may be *repeatable* on only some of the kinds of program element where it is *applicable*. Assume `Foo` is a repeatable annotation type and `FooContainer` is its containing annotation type; then:

- If `Foo` has no `@Target` meta-annotation and `FooContainer` has no `@Target` meta-annotation, then `@Foo` may be repeated on any program element which supports annotations.
- If `Foo` has no `@Target` meta-annotation but `FooContainer` has an `@Target` meta-annotation, then `@Foo` may only be repeated on program elements where `@FooContainer` may appear.
- If `Foo` has an `@Target` meta-annotation, then in the judgment of the designers of the Java programming language, `FooContainer` must be declared with knowledge of the `Foo`'s applicability. Specifically, the kind of program element where `FooContainer` may appear must be the same as, or a subset of, `Foo`'s kinds.

  For example, if `Foo` is applicable to field and method declarations, then `FooContainer` may legitimately serve as `Foo`'s containing annotation type if `FooContainer` is applicable to just field declarations (preventing `@Foo` from being repeated on method declarations). But if `FooContainer` is applicable only to formal parameter declarations, then `FooContainer` was a poor choice of containing annotation type by `Foo` because `@FooContainer` cannot be implicitly declared on some program elements where `@Foo` is repeated.

Similarly, if `Foo` is applicable to field and method declarations, then `FooContainer` cannot legitimately serve as `Foo`'s containing annotation type if `FooContainer` is applicable to field and parameter declarations. While it would be possible to take the intersection of the program elements and make `Foo` repeatable on field declarations only, the presence of additional program elements for `FooContainer` indicates that `FooContainer` was not designed as a containing annotation type for `Foo`. It would therefore be dangerous for `Foo` to rely on it.

An annotation whose type declaration indicates a target of `java.lang.annotation.ElementType.TYPE` can appear in at least as many locations as an annotation whose type declaration indicates a target of `java.lang.annotation.ElementType.ANNOTATION_TYPE`. For example, given the following declarations of repeatable and containing annotation types:

```
@Target(ElementType.TYPE)
@Repeatable(FooContainer.class)
@interface Foo {}

@Target(ElementType.ANNOTATION_TYPE)
@Interface FooContainer {
    Foo[] value();
}
```

`@Foo` can appear on any type declaration while `@FooContainer` can appear on only annotation type declarations. Therefore, the following annotation type declaration is legal:

```
@Foo @Foo
@interface X {}
```

while the following interface declaration is illegal:

```
@Foo @Foo
interface X {}
```

It is a compile-time error if an annotation type *`T`* is (meta-)annotated with an `@Repeatable` annotation whose `value` element indicates a type which is not a containing annotation type of *`T`*.

Consider the following declarations:

```
@Repeatable(FooContainer.class)
@interface Foo {}

@interface FooContainer { Object[] value(); }
```

Compiling the `Foo` declaration produces a compile-time error because `Foo` uses `@Repeatable` to *nominate* `FooContainer` as its containing annotation type, but

`FooContainer` is not in fact a containing annotation type of `Foo`. (The return type of `FooContainer.value()` is not `Foo[]`.)

An annotation type can use `@Repeatable` to nominate at most one containing annotation type because nominating more than one would cause an undesirable choice at compile time, when multiple annotations of the repeatable annotation type are logically replaced with a container annotation (§9.7.5).

An annotation type can be the containing annotation type of at most one annotation type.

This is implied by the requirement that if the declaration of an annotation type *`T`* specifies a containing annotation type of *`TC`*, then the `value()` method of *`TC`* has a return type involving *`T`*, specifically *`T`*`[]`.

An annotation type cannot specify itself as its containing annotation type.

This is implied by the requirement on the `value()` method of the containing annotation type. Specifically, if an annotation type *`A`* specified itself (via `@Repeatable`) as its containing annotation type, then the return type of *`A`*'s `value()` method would have to be *`A`*`[]`; but this would cause a compile-time error since an annotation type cannot refer to itself in its elements (§9.6.1).

More generally, two annotation types cannot specify each other to be their containing annotation types, because cyclic annotation type declarations are illegal.

For example, the following program causes a compile-time error:

```
@interface M {
    O[] value() default {};
}

@interface O {
    M[] value() default {};
}

@M({@O, @O})
@O({@M, @M})
public class Foo {}
```

with the message:

```
Foo.java:2: error: cyclic annotation element type
    O[] value() default {};
        ^
1 error
```

An annotation type *`TC`* may be the containing annotation type of some annotation type *`T`* while also having its own containing annotation type *`TC`* '. That is, a containing annotation type may itself be a repeatable annotation type.

The following are legal declarations:

```
// Foo: Repeatable annotation type
@Repeatable(FooContainer.class)
@interface Foo { int value(); }

// FooContainer: Containing annotation type of Foo
//               and repeatable annotation type
@Repeatable(FooContainerContainer.class)
@interface FooContainer { Foo[] value(); }

// FooContainerContainer: Containing annotation type of FooContainer
@interface FooContainerContainer { FooContainer[] value(); }
```

Thus an annotation of a containing annotation type may be repeated:

```
@FooContainer({@Foo(1)}) @FooContainer({@Foo(2)}) class A {}
```

An annotation type which is both repeatable and containing is subject to the rules on mixing annotations of repeatable annotation type with annotations of containing annotation type (§9.7.5). For example, it is not possible to write multiple `@Foo` annotations alongside multiple `@FooContainer` annotations, nor is it possible to write multiple `@FooContainer` annotations alongside multiple `@FooContainerContainer` annotations. However, if the `FooContainerContainer` annotation type was itself repeatable, then it would be possible to write multiple `@Foo` annotations alongside multiple `@FooContainerContainer` annotations.

9.6.3.6 `@Deprecated`

A Java compiler must produce a warning when a deprecated type, method, field, or constructor is used (overridden, invoked, or referenced by name) *in a construct which is explicitly or implicitly declared*, unless: ...

The only implicitly declared construct that can cause a deprecation warning is a container annotation (§9.7.5). Namely, if *`T`* is a repeatable annotation type and *`TC`* is its containing annotation type, and *`TC`* is deprecated, then repeating the `@T` annotation will cause a deprecation warning. The warning is due to the implicit `@TC` container annotation. It is strongly discouraged to deprecate a containing annotation type without deprecating the corresponding repeatable annotation type.

9.6.3.8 `@Repeatable`

The annotation type `java.lang.annotation.Repeatable` is used on the declaration of a *repeatable annotation type* to indicate its containing annotation type (§9.6).

> Note that an `@Repeatable` meta-annotation on the declaration of *`T`*, indicating *`TC`*, is *not* sufficient to make *`TC`* the containing annotation type of *`T`*. There are numerous well-formedness rules for *`TC`* to be considered the containing annotation type of *`T`*.

9.7.5 Repeating Annotations

It is a compile-time error if more than one annotation of the same type *`T`* appears as a modifier for a given declaration, *unless `T` is repeatable (§9.6), and both `T` and the containing annotation type of `T` are applicable to the declaration (§9.6.3.1).*

If and only if a declaration has multiple annotations of a repeatable annotation type *`T`*, then it is as if the declaration has zero explicitly declared annotations of type *`T`* and one implicitly declared annotation of the containing annotation type of *`T`*.

The implicitly declared annotation is called a *container annotation*. The annotations of repeatable annotation type *`T`* are called *base annotations*.

The elements of the (array-typed) `value` element of the container annotation are all the base annotations in the left-to-right order in which they appear on the declaration.

> It is conventional to write base annotations contiguously on a declaration, but this is not required.

> Note the "if and only if" above. If a declaration only has one annotation of a given repeatable annotation type, then container annotations are not relevant.

It is a compile-time error if a declaration is annotated with more than one annotation of a repeatable annotation type *`T`* and any annotations of the containing annotation type of *`T`*.

> One might expect to be able to repeat an annotation in the presence of its own container:

```
    @Foo(0) @Foo(1) @FooContainer({@Foo(2)}) class A {}
```

> However, it is perverse to use a container annotation unnecessarily, and furthermore the idiom is hard to compile:

- The `@Foo` annotation repeats, so will be wrapped by an `@FooContainer` annotation. Then, the `@FooContainer` annotation repeats. Either the `@FooContainer` annotations are wrapped by an `@FooContainerContainer` annotation, or they are stored directly in the `ClassFile` structure. The first option leads to multiple levels of wrapping and unwrapping, which is undesirable in the judgment of the designers of the Java programming language. The second option is at odds with the "containerization" approach which causes the reflection libraries of the Java SE platform to prohibit duplicate annotations of the same type in a `ClassFile` attribute, even though the Java Virtual Machine permits it.

- Alternatively, compiling the `@Foo` annotations into the `value` element of the `@FooContainer` annotation is undesirable because it changes the semantic content of the handwritten `@FooContainer` annotation.

Ultimately, the presence of a container annotation prevents multiple annotations of its own repeatable annotation type.

It is a compile-time error if a declaration is annotated with any annotations of a repeatable annotation type *`T`* and more than one annotation of the containing annotation type of *`T`*.

Assuming `FooContainer` is itself a repeatable annotation type with a containing annotation type of `FooContainerContainer`, one might expect the following code to be legal:

```
@Foo(1) @FooContainer({@Foo(2)}) @FooContainer({@Foo(3)}) class A {}
```

on the grounds that the `@FooContainer` annotations could be wrapped in a single `@FooContainerContainer`. However, it is perverse to repeat annotations which are themselves containers when an annotation of their underlying repeatable type is present.

The two rules above obviously combine to prohibit multiple annotations of a repeatable annotation type and multiple annotations of its containing annotation type:

```
@Foo(0) @Foo(1) @FooContainer({@Foo(2)}) @FooContainer({@Foo(3)}) class A {}
```

However, they do allow the following simple case which was legal prior to Java SE 8:

```
@Foo(1) @FooContainer({@Foo(2)}) class A {}
```

With only one annotation of the repeatable annotation type `Foo`, no container annotation is implicitly declared, even if `FooContainer` is the containing annotation type of `Foo`. The compiled form of this code is therefore the same in Java SE 8 as in Java SE 7.

9.7.x Assorted corrections to Modifier rules

In Java SE 7, it is generally a compile-time error if the same modifier appears more than once on a declaration. But since an annotation is a modifier, and may appear more than once on a declaration in Java SE 8, the rules must be sharpened to require a compile-time error if the same *keyword* appears more than once as a modifier.

8.1.1 "Class Modifiers": It is a compile-time error if the same keyword appears more than once as a modifier for a class declaration.

8.5 "Member Type Declarations": It is a compile-time error if the same keyword appears more than once as a modifier for a member type declaration.

8.9 "Enums": It is a compile-time error if the same keyword appears more than once as a modifier for an enum declaration.

9.1.1 "Interface Modifiers": It is a compile-time error if the same keyword appears more than once as a modifier for an interface declaration.

9.5 "Member Type Declarations": It is a compile-time error if the same keyword appears more than once as a modifier for a member type declaration in an interface.

8.4.3 "Method Modifiers": It is a compile-time error if the same keyword appears more than once as a modifier for a method declaration.

8.8.3 "Constructor Modifiers": It is a compile-time error if the same keyword appears more than once as a modifier for a constructor declaration."

8.3.1 "Field Modifiers": It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration.

8.4.1 "Formal Parameters": It is a compile-time error if `final` appears more than once as a modifier for a formal parameter declaration.

9.3 "Field Declarations": It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration in an interface.

9.4 "Abstract Method Declarations": It is a compile-time error if the same keyword appears more than once as a modifier for a method declaration in an interface.

14.4 "Local Variable Declaration Statements": It is a compile-time error if `final` appears more than once as a modifier for a local variable declaration.

14.20 "The `try` Statement": It is a compile-time error if `final` appears more than once as a modifier for an exception parameter declaration.

14.20.3 "`try`-with-resources": It is a compile-time error if `final` appears more than once as a modifier for each variable declared in a *ResourceSpecification*.

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, *such as making an annotation type repeatable (§9.6)*, but these have no effect on the linkage of binaries by the Java Virtual Machine. Rather, such changes affect the behavior of reflective APIs that manipulate annotations. The documentation of these APIs specifies their behavior when various changes are made to the underlying annotation types.

1.2 Core Reflection API

In Java SE 7, the annotation retrieval methods of `java.lang.reflect.AnnotatedElement` are as follows:

```
+-------------------------+------------------------+
| Directly present        | Present                |
+-------------------------+------------------------+
| N/A                     | getAnnotation(Class<T>) |
| getDeclaredAnnotations() | getAnnotations()       |
+-------------------------+------------------------+
```

For completeness and consistency, Oracle proposes to:

- Add `getDeclaredAnnotation(Class<T>)`. The behavior is that of `getAnnotation(Class<T>)` but ignoring inherited annotations on classes. *For simplicity, the added method is treated as an "SE 7 method" in the remainder of this document.*
- Refine the specification of `isAnnotationPresent(X)` to be equivalent to:

  ```
  getAnnotation(X) != null
  ```

To expose multiple annotations of a repeatable annotation type on an element in Java SE 8, Oracle proposes to:

- Add `get[Declared]AnnotationsByType(Class<T>)`, the multiple-annotation-aware versions of `get[Declared]Annotation(Class<T>)`. The new methods return an array of annotations of the supplied `Class` which appear on an element, "looking through" a container annotation (if present) when the supplied `Class` represents a repeatable annotation type (§9.6).

  The SE 7-era methods will not be modified to "look through" container annotations.

Here are the annotation retrieval methods of `java.lang.reflect.AnnotatedElement` in Java SE 8:

```
+-------------------------------------+-------------------------------+
| Directly present                    | Present                       |
+-------------------------------------+-------------------------------+
| getDeclaredAnnotation(Class<T>)     | getAnnotation(Class<T>)       |
| getDeclaredAnnotations()            | getAnnotations()              |
| getDeclaredAnnotationsByType(Class<T>) | getAnnotationsByType(Class<T>) |
+-------------------------------------+-------------------------------+
```

The declaration of a class may inherit annotations from the declaration of its superclass. Assume *T* is an annotation type that is applicable to class declarations (via `@Target`) and is inheritable (via `@Inherited`). The policy in Java SE 7 is:

- If a class declaration does not have a "directly present" annotation of type *T*, the class declaration may have a "present" annotation of type *T* due to inheritance.
- If a class declaration does have a "directly present" annotation of type *T*, the annotation is deemed to "override" an annotation of type *T* which is "directly present" or "present" on the superclass.

When the new `get[Declared]AnnotationsByType(Class<T>)` methods are called for a repeatable annotation type *T*, the question is how to extend the policy to handle multiple annotations of type *T* on the superclass and/or subclass. Oracle proposes the following policy for Java SE 8:

- If a class declaration does not have either a "directly present" annotation of type *T* or an "indirectly present by containment" annotation of type *T*, the class declaration may be "associated" with an annotation of type *T* due to inheritance.
- If a class declaration does have either a "directly present" annotation of type *T* or an "indirectly present by containment" annotation of type *T*, the annotation is deemed to "override" *every* annotation of type *T* which is "associated" with the superclass.

This policy for Java SE 8 is reified in the following definitions:

- An annotation *A* is *directly present* on an element *E* if *E* has a `RuntimeVisibleAnnotations` or `RuntimeVisibleParameterAnnotations` or `RuntimeVisibleTypeAnnotations` attribute, and the attribute contains *A*.
- An annotation *A* is *indirectly present* on an element *E* if *E* has a `RuntimeVisibleAnnotations` or `RuntimeVisibleParameterAnnotations` or `RuntimeVisibleTypeAnnotations` attribute, and *A*'s type is repeatable, and the attribute contains exactly one annotation whose `value` element contains *A* and whose type is the containing annotation type of *A*'s type (§9.6).
- An annotation *A* is *present* on an element *E* if either:
  – *A* is *directly present* on *E*; or

  – No annotation of `A`'s type is *directly present* on `E`, and `E` is a class, and `A`'s type is inheritable (§9.6.3.3), and `A` is *present* on the superclass of `E`.

- An annotation `A` is *associated* with an element `E` if either:
  – `A` is *directly or indirectly present* on `E`; or
  – No annotation of `A`'s type is *directly or indirectly present* on `E`, and `E` is a class, and `A`'s type is inheritable (§9.6.3.3), and `A` is *associated* with the superclass of `E`.

For an invocation of `get[Declared]AnnotationsByType(Class<T>)`, the order of annotations which are *directly or indirectly present* on an element `E` is computed as if *indirectly present* annotations on `E` are *directly present* on `E` in place of their container annotation, in the order in which they appear in the `value` element of the container annotation.

If the reflection libraries of the Java SE platform load an annotation type `T` which is (meta-)annotated with an `@Repeatable` annotation whose `value` element indicates a type `TC`, but `TC` does not declare a `value()` method with a return type of `T[]`, then an exception of type `java.lang.annotation.AnnotationFormatError` is thrown.

This exception indicates an ill-formed relationship between a supposedly repeatable annotation type and a prospective containing annotation type. Note that `TC` is not required to meet all of the conditions necessary at compile time to be a *containing annotation type of* `T` (§9.6). At runtime, it suffices to check that `TC` has a suitably array-typed `value()` method.

Example 1.2-1. Repeating an annotation is behaviorally compatible

Assume the following declarations, where `Foo` is inheritable:

```
@Foo(1) class A {}
        class B extends A {}
```

SE 7 methods in SE 7 and 8:

```
A.class.getAnnotation(Foo.class)         = @Foo(1)
A.class.getDeclaredAnnotation(Foo.class) = @Foo(1)

A.class.getAnnotation(FooContainer.class)         = null
A.class.getDeclaredAnnotation(FooContainer.class) = null

A.class.getAnnotations()         = [ @Foo(1) ]
A.class.getDeclaredAnnotations() = [ @Foo(1) ]

B.class.getAnnotation(Foo.class)         = @Foo(1)
B.class.getDeclaredAnnotation(Foo.class) = null

B.class.getAnnotation(FooContainer.class)         = null
B.class.getDeclaredAnnotation(FooContainer.class) = null

B.class.getAnnotations()         = [ @Foo(1) ]
B.class.getDeclaredAnnotations() = [ ]
```

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ @Foo(1) ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(1) ]

B.class.getAnnotationsByType(Foo.class)         = [ @Foo(1) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. Per §9.6, `FooContainer` must be inheritable because `Foo` is inheritable. Assume the declarations are changed to:

```
@Foo(1) @Foo(2) class A {}
                class B extends A {}
```

The SE 7 methods assume at most one annotation of a given type on any element. The container annotation which is implicitly declared at compile-time has the effect of "hiding" multiple annotations of type `Foo` from these methods.

SE 7 methods in SE 8:

```
A.class.getAnnotation(Foo.class)         = null
A.class.getDeclaredAnnotation(Foo.class) = null

A.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
A.class.getDeclaredAnnotation(FooContainer.class) = @FooContainer(...)

A.class.getAnnotations()         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotations() = [ @FooContainer(...) ]

B.class.getAnnotation(Foo.class)         = null
B.class.getDeclaredAnnotation(Foo.class) = null

B.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
B.class.getDeclaredAnnotation(FooContainer.class) = null

B.class.getAnnotations()         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotations() = [ ]
```

The SE 8 methods "look through" the container annotation to return the multiple `@Foo` annotations. The container annotation is returned if queried for explicitly.

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ @Foo(1), @Foo(2) ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(1), @Foo(2) ]

A.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]

B.class.getAnnotationsByType(Foo.class)           = [ @Foo(1), @Foo(2) ]
B.class.getDeclaredAnnotationsByType(Foo.class)   = [ ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ ]
```

Now suppose an `@Foo` annotation is placed on the subclass:

```
@Foo(1) @Foo(2) class A {}
@Foo(3)         class B extends A {}
```

The SE 7 methods for this class `B` do not change their behavior between SE 7 and SE 8. Namely, `@Foo(3)` on `B` is deemed to "override" every `@Foo` annotation on `A`. This policy is reified by storing `@Foo(1) @Foo(2)` inside a container annotation, so they are effectively hidden when `B` is queried for inherited `@Foo` annotations.

SE 7 methods in SE 7 and 8:

```
B.class.getAnnotation(Foo.class)         = @Foo(3)
B.class.getDeclaredAnnotation(Foo.class) = @Foo(3)

B.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
B.class.getDeclaredAnnotation(FooContainer.class) = null

B.class.getAnnotations()         = [ @Foo(3), @FooContainer(...) ]
B.class.getDeclaredAnnotations() = [ @Foo(3) ]
```

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(3) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(3) ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ ]
```

Let us return to the initial declarations with one `@Foo` annotation on the superclass, but now suppose that `Foo` is *not* inheritable:

```
@Foo(1) class A {}
        class B extends A {}
```

The SE 7 methods in SE 7 and SE 8 are identical to the earlier scenario with these declarations of class `A` and `B`, except that now `B.class.getAnnotation(Foo.class)` returns `null` (rather than `@Foo(1)`) and `B.class.getAnnotations()` returns `[ ]` (rather than `[ @Foo(1) ]`). In the same fashion, the SE 8 methods in SE 8 are identical to that earlier scenario, except that now `B.class.getAnnotationsByType(Foo.class)` returns `[ ]` (rather than `[ @Foo(1) ]`).

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. While `Foo` is not inheritable, suppose that `FooContainer` *is* inheritable. (Per §9.6, it is permissible for only the containing annotation type to be inheritable.) Assume the declarations are changed to:

```
@Foo(1) @Foo(2) class A {}
                class B extends A {}
```

The SE 7 methods in SE 7 and SE 8 are identical to the earlier scenario with these declarations of class `A` and `B`. The SE 8 methods *on class* `A` "look through" the container annotation to return the multiple `@Foo` annotations, also like the previous scenario. However, the SE 8 methods *on class* `B` see the inherited container annotation but not the (non-inherited) base annotations.

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ @Foo(1), @Foo(2) ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(1), @Foo(2) ]

A.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]

B.class.getAnnotationsByType(Foo.class)           = [ ]
B.class.getDeclaredAnnotationsByType(Foo.class)   = [ ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ ]
```

Now suppose an `@Foo` annotation is placed on the subclass:

```
@Foo(1) @Foo(2) class A {}
@Foo(3)         class B extends A {}
```

The SE 7 and SE 8 methods for this class `B` are identical to the earlier scenario with these declarations of class `A` and `B`. For the SE 8 methods, the earlier scenario showed `@Foo(3)` overriding `@Foo(1) @Foo(2)` which would otherwise have been inherited by class `B` (since `Foo` was inheritable). That consideration is moot here, since `Foo` is not inheritable and thus `@Foo(1) @Foo(2)` are never inherited by class `B`. However, the end result is the same: the only `Foo` annotation visible on class `B` is `@Foo(3)`.

Example 1.2-2. Idiomatic container annotation continues to work

Assume a declaration with an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations:

```
@FooContainer({@Foo(1),@Foo(2)}) class A {}
```

SE 7 methods in SE 7 and SE 8:

```
A.class.getAnnotation(Foo.class)         = null
A.class.getDeclaredAnnotation(Foo.class) = null

A.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
A.class.getDeclaredAnnotation(FooContainer.class) = @FooContainer(...)

A.class.getAnnotations()         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotations() = [ @FooContainer(...) ]
```

Note that the SE 7 methods above behave the same as in Example 1.2-1, “Repeating an annotation is behaviorally compatible” when `Foo` was made repeatable by its author and `@Foo` was repeated. Thus, a legacy client which uses SE 7 methods is indifferent to whether an annotation writer uses an idiomatic container or applies multiple annotations directly.

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ ]

A.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. This "opt-in" by the author of the annotation types has no effect on the behavior of the SE 7 methods, but the SE 8 methods will "look through".

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ @Foo(1), @Foo(2) ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(1), @Foo(2) ]

A.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Example 1.2-3. Mix of singular and idiomatic container annotations continues to work

Assume a declaration with one `@Foo` annotation *and* an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations:

```
@Foo(0) @FooContainer({@Foo(1),@Foo(2)}) class A {}
```

SE 7 methods in SE 7 and SE 8:

```
A.class.getAnnotation(Foo.class)         = @Foo(0)
A.class.getDeclaredAnnotation(Foo.class) = @Foo(0)

A.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
A.class.getDeclaredAnnotation(FooContainer.class) = @FooContainer(...)

A.class.getAnnotations()         = [ @Foo(0), @FooContainer(...) ]
A.class.getDeclaredAnnotations() = [ @Foo(0), @FooContainer(...) ]
```

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ @Foo(0) ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(0) ]

A.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. This "opt-in" by the author of the annotation types has no effect on the behavior of the SE 7 methods, but the SE 8 methods will "look through".

SE 8 methods in SE 8:

```
A.class.getAnnotationsByType(Foo.class)         = [ @Foo(0), @Foo(1), @Foo(2) ]
A.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(0), @Foo(1), @Foo(2) ]

A.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Example 1.2-4. Mix of singular and idiomatic container annotations continues to work (with inheritance)

Assume a declaration with one `@Foo` annotation and a subclass with an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations. Assume that `Foo`, while not repeatable, is inheritable:

```
@Foo(0)                           class A {}
@FooContainer({@Foo(1),@Foo(2)}) class B extends A {}
```

SE 7 methods in SE 7 and 8:

```
B.class.getAnnotation(Foo.class)         = @Foo(0)
B.class.getDeclaredAnnotation(Foo.class) = null

B.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
B.class.getDeclaredAnnotation(FooContainer.class) = @FooContainer(...)

B.class.getAnnotations()         = [ @Foo(0), @FooContainer(...) ]
B.class.getDeclaredAnnotations() = [ @FooContainer(...) ]
```

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(0) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. (Per §9.6, `FooContainer` must be inheritable because `Foo` is inheritable, even though `@FooContainer` is not inherited in this example.) This "opt-in" by the author of the annotation types has no effect on the behavior of the SE 7 methods, but the SE 8 methods will "look through". The contained `@Foo` annotations on class `B` override the uncontained `@Foo(0)` annotation on class `A` which would otherwise be inherited by class `B` (since `Foo` is inheritable).

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(1), @Foo(2) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(1), @Foo(2) ]

B.class.getAnnotationsByType(FooContainer.class         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class = [ @FooContainer(...) ]
```

Let us return to the declaration with one `@Foo` annotation and a subclass with an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations. `Foo` is inheritable. Assume there is an additional `@Foo` annotation on the subclass:

```
@Foo(0)                                  class A {}
@Foo(3) @FooContainer({@Foo(1),@Foo(2)}) class B extends A {}
```

SE 7 methods in SE 7 and 8:

```
B.class.getAnnotation(Foo.class)         = @Foo(3)
B.class.getDeclaredAnnotation(Foo.class) = @Foo(3)

B.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
B.class.getDeclaredAnnotation(FooContainer.class) = @FooContainer(...)

B.class.getAnnotations()         = [ @Foo(3), @FooContainer(...) ]
B.class.getDeclaredAnnotations() = [ @Foo(3), @FooContainer(...) ]
```

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(3) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(3) ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. This "opt-in" by the author of the annotation types has no effect on the behavior of the SE 7 methods, but the SE 8 methods will "look through". The contained and uncontained `@Foo` annotations on class `B` collectively override the `@Foo(0)` annotation on class `A` which would otherwise be inherited by class `B` (since `Foo` is inheritable).

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(3), @Foo(1), @Foo(2) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(3), @Foo(1), @Foo(2) ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ @FooContainer(...) ]
```

Example 1.2-5. Mix of singular and idiomatic container annotations continues to work (with inheritance, reversed)

Assume a declaration with one `@Foo` annotation and a *superclass* with an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations. Assume that `FooContainer` is inheritable:

```
@FooContainer({@Foo(1),@Foo(2)}) class A {}
@Foo(0)                          class B extends A {}
```

SE 7 methods in SE 7 and 8:

```
B.class.getAnnotation(Foo.class)         = @Foo(0)
B.class.getDeclaredAnnotation(Foo.class) = @Foo(0)

B.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
B.class.getDeclaredAnnotation(FooContainer.class) = null

B.class.getAnnotations()         = [ @Foo(0), @FooContainer(...) ]
B.class.getDeclaredAnnotations() = [ @Foo(0) ]
```

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(0) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(0) ]

B.class.getAnnotations(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotations(FooContainer.class) = [ ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. (We have not said if `Foo` is inheritable, but per §9.6, it is legal for `FooContainer` to be inheritable even if `Foo` is not inheritable.) This "opt-in" by the author of the annotation types has no effect on the behavior of the SE 7 methods, but the SE 8 methods will "look through". However, the `@Foo(0)` annotation on class `B` overrides the contained `@Foo` annotations on class `A` which, if `Foo` is inheritable, would otherwise be inherited by class `B`.

SE 8 methods in SE 8:

```
B.class.getAnnotationsByType(Foo.class)         = [ @Foo(0) ]
B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(0) ]

B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(..) ]
B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ ]
```

Let us return to the declaration with one `@Foo` annotation and a superclass with an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations. `Foo` and `FooContainer` are inheritable. Assume there is an additional `@Foo` annotation on the superclass:

```
@Foo(3) @FooContainer({@Foo(1),@Foo(2)}) class A {}
```

```
    @Foo(0)                                  class B extends A {}
```

SE 7 methods in SE 7 and 8:

```
    B.class.getAnnotation(Foo.class)         = @Foo(0)
    B.class.getDeclaredAnnotation(Foo.class) = @Foo(0)

    B.class.getAnnotation(FooContainer.class)         = @FooContainer(...)
    B.class.getDeclaredAnnotation(FooContainer.class) = null

    B.class.getAnnotations()         = [ @Foo(0), @FooContainer(...) ]
    B.class.getDeclaredAnnotations() = [ @Foo(0) ]
```

SE 8 methods in SE 8:

```
    B.class.getAnnotationsByType(Foo.class)         = [ @Foo(0) ]
    B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(0) ]

    B.class.getAnnotationsByType(FooContainer.class)         = [ @FooContainer(...) ]
    B.class.getDeclaredAnnotationsByType(FooContainer.class) = [ ]
```

Now suppose `Foo` is made repeatable with `FooContainer` as its containing annotation type. This "opt-in" by the author of the annotation types has no effect on the behavior of the SE 7 methods, but the SE 8 methods will "look through". However, the `@Foo(0)` annotation on class `B` overrides the contained and uncontained `@Foo` annotations on class `A` which, if `Foo` is inheritable, would otherwise be inherited by class `B`

SE 8 methods in SE 8:

```
    B.class.getAnnotationsByType(Foo.class)         = [ @Foo(0) ]
    B.class.getDeclaredAnnotationsByType(Foo.class) = [ @Foo(0) ]

    B.class.getAnnotationsByType(FooContainer.class)        = [ @FooContainer(...) ]
    B.class.getDeclaredAnnotationsByType(FooContainer.class = [ ]
```

1.3 Language Model API

In Java SE 7, the annotation retrieval methods of the language model API (defined by JSR 269) are:

- In `javax.lang.model.element.Element`, `getAnnotation(Class<T>)` for retrieving *present* annotations of the supplied `Class`, inspired by `java.lang.reflect.AnnotatedElement`;
- In `javax.lang.model.element.Element`, `getAnnotationMirrors()` for retrieving mirrors of *directly present* annotations;

- In `javax.lang.model.util.Elements`, `getAllAnnotationMirrors(Element)` for retrieving mirrors of *present* annotations.

To expose multiple annotations of a repeatable annotation type on an element in Java SE 8, Oracle proposes to:

- In `javax.lang.model.element.Element`, add `getAnnotationsByType(Class<T>)` for consistency with `java.lang.reflect.AnnotatedElement`. Similar to §1.2, `getAnnotationsByType(Class<T>)` returns all annotations of the supplied `Class` which are *associated* with the element, "looking through" a container annotation (if present) when the supplied `Class` represents a repeatable annotation type.

The language model API relies on the following definitions, derived from the corresponding definitions in core reflection. The chief difference is in the "directly present" term, since the language model API can expose constructs from source files *and* `class` files.

- An annotation *`A`* is *directly present* on a construct *`C`* if either:
  - *`A`* is explicitly or implicitly declared as applying to the source code representation of *`C`*; or
  - *`A`* appears in the executable output corresponding to *`C`*, such as in the `RuntimeVisibleAnnotations` attribute of a `class` file.

  Typically, if *`A`* is the only annotation of its type applied to *`C`*, then *`A`* is explicitly declared as applying to the source code representation of *`C`*. If there are multiple annotations of the same type applied to *`C`*, then a container annotation is implicitly declared as applying to the source code representation of *`C`* (§9.7.5). Note that the multiple annotations of the same type are *not* directly present on *`C`*; rather, they are *indirectly present*.)

- An annotation *`A`* is *indirectly present* on a construct *`C`* if *`A`*'s type is repeatable, and *`C`* has exactly one annotation whose `value` element contains *`A`* and whose type is the containing annotation type of *`A`*'s type (§9.6).
- An annotation *`A`* is *present* on a construct *`C`* if either:
  - *`A`* is *directly present* on *`C`*; or
  - No annotation of *`A`*'s type is *directly present* on *`C`*, and *`C`* is a class, and *`A`*'s type is inheritable (§9.6.3.3), and *`A`* is *present* on the superclass of *`C`*.
- An annotation *`A`* is *associated* with a construct *`C`* if either:
  - *`A`* is *directly* or *indirectly present* on *`C`*; or

  - No annotation of `A`'s type is *directly* or *indirectly present* on `C`, and `C` is a class, and `A`'s type is inheritable (§9.6.3.3), and `A` is *associated* with the superclass of `C`.

As far as possible, the following correspondences should hold between the language model API and the core reflection API: (where `X` is a class)

- `|Element for X|.getAnnotationMirrors()` should return the same as `X.class.getDeclaredAnnotations()`.
- `Elements.getAllAnnotationMirrors(|Element for X|)` should return the same as `X.class.getAnnotations()`.

CHAPTER 2

Method Parameter Reflection

JAVA programmers traditionally consider the names of formal parameters of methods and constructors to be debugging symbols. Parameter names are stored in `class` files only if debugging flags are passed to the compiler (e.g. `javac -g`) and there is no general API to retrieve parameter names from a `class` file even if present.

Oracle believes that parameter names are an integral part of a Java program because they hold so much value for reflective clients like IDEs and language-interop tools. The purpose of the *Method Parameter Reflection* feature in Java SE 8 is to define first-class `class` file storage and API retrieval for parameter names and related metadata.

Oracle believes the ability to retrieve parameter names at run time loses much of its value if parameters are "opted out" of `class` file storage by default, and instead have to "opt in" by some syntactic means. Unfortunately, the static and dynamic footprint of storing parameter names will be an unwelcome surprise for many `class` file producers and consumers. Also, storing parameter names by default means that new information will be exposed about security-sensitive methods, e.g. parameter names like `secret` or `password`. In light of these concerns, Oracle in Java SE 8 will consider parameter names as "opted out" of `class` file storage by default.

Furthermore, Oracle will not define an "opt in" mechanism in the Java programming language in Java SE 8. Instead, Oracle will seek to ensure that compilers for the Java programming language can be configured to store parameter names in `class` files (e.g. `javac  -parameters`). The new `java.lang.reflect.Parameter` API which retrieves parameter names is indifferent to how a `class` file was generated, so the Java programming language is free to add an "opt in" mechanism after Java SE 8 without affecting reflective clients.

2.1 *The Java Virtual Machine Specification, Java SE 8 Edition*

Recall that a `ClassFile` of version 51.0 (Java SE 7) stores only:

- Parameter types as seen in the Java programming language, in a method type signature referenced by `method_info.attributes['Signature'].signature_index`.
- Parameter types as seen by the Java Virtual Machine, in a method descriptor referenced by `method_info.descriptor_index`.

(We ignore the storage of parameter names in the `LocalVariableTable` attribute because it is generated only when debugging output is generated by a compiler, and it is invisible to the `java.lang.reflect` API.)

The storage of parameter names in a `ClassFile` of version 52.0 (Java SE 8) is informed by three points:

1. Parameter names are not essential to the Java Virtual Machine. They play no part in linkage, so changing a parameter name will never be a binary-incompatible change.
2. `ClassFile` producers will often wish to avoid storing parameter names, and to strip them from the `ClassFile` if present.
3. Additional information about parameters may be stored in future releases of the Java SE platform, such as default values or modifiers other than `final`.

For these reasons, parameter names and flags seen in the Java programming language are not stored directly in the venerable `method_info` structure. Instead, they are stored in a new attribute, `MethodParameters`, as specified below.

4.2.2 Unqualified Names

Names of methods, fields, local variables, *and formal parameters* are stored as unqualified names. An unqualified name must *contain at least one Unicode code point* and must not contain the (Unicode code points corresponding to the) ASCII characters '.', ';', '[', or '/'.

4.7 Attributes

The `MethodParameters` attribute must be recognized and correctly read by a `class` file reader in order to properly implement the Java SE platform class libraries (§2.12), if the `class` file's version number is 52.0 or above and the Java Virtual

Machine implementation recognizes `class` files whose version number is 52.0 or above.

4.7.22 The `MethodParameters` Attribute

The `MethodParameters` attribute is an optional variable-length attribute in the `attributes` table of a `method_info` structure. A `MethodParameters` attribute records information about the formal parameters of a method, such as their names.

The `attributes` table of a `method_info` structure must not contain more than one `MethodParameters` attribute.

```
MethodParameters_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 parameters_count;
    {   u2 name_index;
        u2 access_flags;
    } parameters[parameters_count];
}
```

The items of the `MethodParameters_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "`MethodParameters`".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes.

`parameters_count`

The value of the `parameters_count` item indicates the number of parameter descriptors in the method descriptor referenced by the `descriptor_index` of the attribute's enclosing `method_info` structure.

This is not a constraint which a Java Virtual Machine implementation must enforce during format checking (JVMS 4.9). The task of matching parameter descriptors in a method descriptor against the items in this attribute that indicate parameter metadata is done by the reflection libraries of the Java SE platform.

`parameters_count` is one byte because JVMS 4.3.3 limits a method descriptor to 255 parameters.

`parameters`

Each `parameters` array entry contains the following pair of items:

`name_index`

The value of the `name_index` item must either be zero or a valid index into the `constant_pool` table. If the value of the `name_index` item is nonzero, the `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing a valid unqualified name denoting a formal parameter (§4.2.2). If the value of the `name_index` item is zero, then this `parameters` element indicates a formal parameter with no name.

`access_flags`

The value of the `access_flags` item is as follows:

`0x0010 (ACC_FINAL)`

Indicates that the formal parameter was declared `final`.

`0x1000 (ACC_SYNTHETIC)`

Indicates that the formal parameter was not explicitly or implicitly declared in source code, according to the specification of the language in which the source code was written (§13.1). (The formal parameter is an implementation artifact of the compiler which produced this `class` file.)

`0x8000 (ACC_MANDATED)`

Indicates that the formal parameter was implicitly declared in source code, according to the specification of the language in which the source code was written (§13.1). (The formal parameter is mandated by a language specification, so all compilers for the language must emit it.)

`access_flags` uses the traditional values for `ACC_FINAL` and `ACC_SYNTHETIC`. To allow a compiler to mark a formal parameter as implicitly declared in source code (§13.1), we define 0x8000 as `ACC_MANDATED`. It is possible that this flag will be legal for other `ClassFile` artifacts, such as fields and methods, in future releases of the Java SE platform.

There is no implicit or explicit correspondence between the i'th entry in `parameters` and the i'th type in the signature of the enclosing method (`method_info . attributes['Signature'] . signature_index`).

There is an implicit correspondence between the i'th entry in `parameters` and the i'th type in the descriptor of the enclosing method (`method_info . descriptor_index`).

> This correspondence, and the associated constraint at reflection-time that `parameters_count` matches the arity of the descriptor, is for simplicity. While one could imagine storing information for only a subset of parameters typed in the descriptor, it would unduly complicate the `ClassFile` format given that most compilers will produce a `MethodParameters` attribute denoting *every* parameter typed in the descriptor (even parameters which are not physically present in source).

There is an implicit correspondence between the i'th entry in `parameters` and the i'th annotation in the parameter annotations of the enclosing method (`method_info . attributes['RuntimeVisibleParameterAnnotations'] . parameter_annotations`).

2.2 *The Java Language Specification, Java SE 8 Edition*

8.8.9 Default Constructor

If a class *other than an anonymous class* contains no constructor declarations, then a default constructor is implicitly declared. The form of the default constructor is as follows:

- The default constructor has no formal parameters and no `throws` clauses.
- If the class being declared is the primordial class `Object`, then the default constructor has an empty body.

  Otherwise, the default constructor simply invokes the superclass constructor with no arguments.
- In a class type, if the class is declared `public` ...

If a class is an anonymous class, then an anonymous constructor is implicitly declared (§15.9.5.1).

The constructor of a non-private inner member class implicitly declares, as its first formal parameter, a variable representing the immediately enclosing instance (§8.1.3).

> The rationale for why only this kind of constructor has an implicitly declared parameter is subtle. The following explanation may be helpful:
>
> 1) In a class instance creation expression for a non-private inner member class, §15.9.2 specifies the immediately enclosing instance of the member class. The member class may have been emitted by a compiler which is different than the compiler of the class instance creation expression. Therefore, there must be a standard way for the compiler of the creation expression to pass a reference (representing the immediately enclosing instance) to the member class's constructor. Consequently, the Java programming language deems that a

non-private inner member class's constructor implicitly declares an initial parameter for the immediately enclosing instance.

2) In a class instance creation expression for a local class or anonymous class, §15.9.2 specifies the immediately enclosing instance of the local/anonymous class. The local/anonymous class is necessarily emitted by the same compiler as the class instance creation expression. That compiler can represent the immediately enclosing instance how ever it wishes. There is no need for the Java programming language to implicitly declare a parameter in the local/anonymous class's constructor.

3) In a class instance creation expression for an anonymous class, and where the anonymous class's superclass is inner, §15.9.2 specifies the immediately enclosing instance with respect to the superclass. Similar to (1), the superclass may have been emitted by a compiler which is different than the compiler of the class instance creation expression. Therefore, there must be a standard way for the compiler of the creation expression to pass a reference (representing the immediately enclosing instance with respect to the superclass) to the superclass's constructor. Consequently, the Java programming language deems that an anonymous class's constructor implicitly declares an initial parameter for the immediately enclosing instance with respect to the superclass.

The fact that a non-private inner member class may be accessed by a different compiler than compiled it, whereas a local or anonymous class is always accessed by the same compiler that compiled it, explains why the binary name of a non-private inner member class is defined to be predictable but the binary name of a local or anonymous class is not (§13.1).

8.9.2 Enum Body Declarations

If an enum type contains no constructor declarations, then a default constructor with no formal parameters (to match the implicit empty argument list (§8.9.1)) is implicitly declared. The default constructor has no `throws` clause and is `private`.

In practice, a Java compiler is likely to mirror the `Enum` type by declaring `String` and `int` parameters for an enum's default constructor. However, these parameters are not specified as "implicitly declared" because different compilers do not need to agree on the form of an enum's default constructor. Only the compiler of the enum itself knows how to create the enum's constants. Other compilers, when compiling expressions that use enum constants, can simply rely on the `public static` fields of the enum type - which *are* implicitly declared - without regard for how those fields were initialized.

13.1 The Form of a Binary

A construct emitted by a Java compiler must be marked as *synthetic* if it does not correspond to a construct declared explicitly or implicitly in source code, unless the emitted construct is a class initialization method (JVMS 2.9).

A construct emitted by a Java compiler must be marked as *mandated* if it corresponds to a formal parameter declared implicitly in source code (§8.8.9, §8.9.2, §15.9.5.1).

The following formal parameters are declared implicitly in source code:

- the first formal parameter of a constructor of a non-private inner member class (§8.8.9)
- the first formal parameter of an anonymous constructor of an anonymous class whose superclass is inner (§15.9.5.1)
- the formal parameter called `name` of the `valueOf` method which is implicitly declared in an enum (§8.9.2)

Because only formal parameters have a `ClassFile` representation which lets them be marked as *mandated* (JVMS 4.7.22), a construct emitted by a Java compiler does not have to be marked as *mandated* if it corresponds to any of the following non-parameter constructs declared implicitly in source code:

- default constructors of classes (§8.8.9) and enums (§8.9.2)
- anonymous constructors (§15.9.5.1)
- the `values` and `valueOf` methods of enums (§8.9.2)
- certain `public` fields of enums (§8.9.2)
- certain `public` methods of interfaces (§9.2)
- container annotations (§9.7.5)

15.9.5.1 Anonymous Constructors

An anonymous class cannot have an explicitly declared constructor. *Instead, an anonymous constructor is implicitly declared for an anonymous class.* The form of the anonymous constructor of an anonymous class *`C`* with direct superclass *`S`* is as follows: ...

2.3 Core Reflection API

To expose information about formal parameters of methods and constructors in Java SE 8, Oracle proposes to:

- Refine the specification of the `java.lang.reflect.Executable` class (which in Java SE 8 is the superclass of `java.lang.reflect.Method` and `java.lang.reflect.Constructor`) by adding a method `getParameters()` which returns an array of element type `java.lang.reflect.Parameter`.

The class `java.lang.reflect.Parameter` is as follows:

```
package java.lang.reflect;
public final class Parameter implements AnnotatedElement {
  // Object methods
  public boolean equals(Object)
  public int     hashCode()
  public String  toString()

  // General aspects of a formal parameter (name, finality, etc)
  public Executable getDeclaringExecutable()
  public int        getModifiers()
  public String     getName()
  public Type       getParameterizedType()
  public Class<?>   getType()
  public boolean    isImplicit()
  public boolean    isNamePresent()
  public boolean    isSynthetic()
  public boolean    isVarArgs()

  // Declaration annotations (AnnotatedElement methods)
  public boolean isAnnotationPresent(Class<? extends Annotation>)
  public <T extends Annotation> T   getAnnotation(Class<T>)
  public Annotation[]               getAnnotations()
  public <T extends Annotation> T[] getAnnotationsByType(Class<T>)
  public <T extends Annotation> T   getDeclaredAnnotation(Class<T>)
  public Annotation[]               getDeclaredAnnotations()
  public <T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T>)

  // Type annotations
  public AnnotatedType getAnnotatedType()
}
```

If a method in a `class` file has no `MethodParameters` attribute, then the `getParameters()` method of `java.lang.reflect.Executable` must act as if each parameter of the method was explicitly declared (i.e. is not implicit or synthetic), has no modifiers, and has a name `arg`*`I`* where *`I`* is the position of the parameter's corresponding parameter descriptor in the method descriptor (JVMS 4.3.3), with the first parameter descriptor having position 0. For the parameter corresponding to the last parameter descriptor in the method descriptor, it is as if the parameter is variable arity if the method is variable arity, and the parameter is not variable arity if the method is not variable arity.

If a `parameters` element in a `MethodParameters` attribute indicates a formal parameter with no name, then the `getParameters()` method of `java.lang.reflect.Executable` must act as if that `parameters` element indicates a method parameter with the name `arg`*`I`* where *`I`* is the position of the parameter's corresponding parameter descriptor in the method descriptor (JVMS 4.3.3), with the first parameter descriptor having position 0.

The paragraph above mentions only methods, not constructors, because `java.lang.reflect.Parameter` offers a `class` file view of parameters and a `class` file contains no constructors which could have parameters. Constructors in the Java programming language are compiled to methods called `<init>` in the `class` file.

The "real" 0-indexed parameter to an instance method (that is, to most methods) is the receiver object, but it is not represented in the method descriptor. Therefore, the 0'th parameter descriptor in the method descriptor - "arg0" - represents the first parameter.

The specification of key methods is as follows:

`getName`

Returns the name of the parameter. If the parameter's name is present, then this method returns the name provided by the `class` file. Otherwise, this method synthesizes a name of the form `arg`*`I`*, where *`I`* is the index of the parameter in the descriptor of the method which declares the parameter.

`isNamePresent`

Returns true if the parameter has a name according to the `class` file; returns false otherwise. Whether a parameter has a name is determined by the `MethodParameters` attribute of the method which declares the parameter.

`isImplicit`

Returns true if the parameter is implicitly declared in source code (§13.1). Returns false otherwise.

`isSynthetic`

Returns true if the parameter is not explicitly or implicitly declared in source code (§13.1). Returns false otherwise.

`toString`

Returns a string describing this parameter. The format is the modifiers for the parameter, if any, in canonical order as recommended by *The Java Language Specification, Java SE 8 Edition,* followed by the fully-qualified type of the parameter (excluding the last `[]` if the parameter is variable arity), followed by "..." if the parameter is variable arity, followed by a space, followed by the name of the parameter.

If a method in a `class` file has a `MethodParameters` attribute, then the `getParameters()` method of `java.lang.reflect.Executable` must throw `MalformedParametersException` (a subclass of `RuntimeException`) if any of the following are true for the `Executable`'s `MethodParameters` attribute:

- The number of parameters indicated by the attribute is different than the number of parameter descriptors in the enclosing method's descriptor.

- The attribute contains a parameter with a non-zero `name_index` item, and the corresponding constant pool entry is of the wrong type or does not hold a valid unqualified name.
- The attribute contains a parameter with an illegal `access_flags` item.

2.4 Language Model API

In Java SE 7, a formal parameter of a method or constructor is represented by `javax.lang.model.element.VariableElement`. However, almost all information about the parameter is obtained via a superinterface, `javax.lang.model.element.Element`.

For `javax.lang.model.element.Element` in Java SE 8, Oracle proposes to:

- Refine the specification of `getSimpleName()` so that: "If this element represents a method or constructor parameter, the name of the parameter is returned."
- Refine the implementation of `getEnclosingElement()` so that, if the element is a method or constructor parameter, the declaring method or constructor is returned. This behavior is permitted by the method's specification in Java SE 7.
- Refine the implementation of `getModifiers()` so that, if the element is a method or constructor parameter, a `final` modifier is returned if present. This behavior is permitted by the method's specification in Java SE 7.

Oracle does not propose to modify `javax.lang.model.element.Element` (or `VariableElement`) to expose whether a method or constructor parameter is explicitly declared, implicitly declared, or variable arity.