Repeating Annotations and Method Parameter Reflection

Alex Buckley
Joe Darcy

2012-11-09

Specification: JSR-000901 Java™ Language Specification ("Specification")
Version: Java SE 8
Status: Draft
Release: November 2012

NOTICE

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Oracle intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Oracle if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY ORACLE. ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specif ication in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. ORACLE MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF ORACLE AND/

OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Oracle (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Oracle with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

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 automatically stored in an array-valued element of a "container annotation". A little two-way setup is required to associate a "repeatable" annotation type with its "containing" annotation type:

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

@ContainerFor(Foo.class)
@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 that represents class `A` offers reflective operations that automatically "look through" `@FooContainer` and expose the two `@Foo` annotations directly.

Together, the meta-annotations `@ContainedBy` and `@ContainerFor` enable *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's own declaration to limit where it may

appear), in Java SE 8 an annotation may appear zero times, once, or more than once. The role of each meta-annotation is:

- `@ContainedBy` constrains the compile-time translation of multiple annotations into a single array-valued annotation.
- `@ContainerFor` supports the Java SE platform reflection API in differentiating automatically-generated container annotations from legacy annotations which served as idiomatic containers prior to Java SE 8.

A note on terminology: *The Java™ Language Specification* 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*

9.6 Annotation Types

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

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

- The declaration of *`TC`* is (meta-)annotated with an `@ContainerFor` annotation whose `value` element indicates *`T`*.
- The declaration of *`T`* is (meta-)annotated with an `@ContainedBy` annotation whose `value` element indicates *`TC`*.
- *`TC`* declares a `value()` method whose return type is *`T`*`[]`.
- Any methods declared by *`TC`* other than `value()` have a default value (JLS 9.6.2).
- *`TC`* is retained for at least as long as *`T`*, where retention is expressed explicitly or implicitly with the `@Retention` annotation (JLS 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`.

- *T* is applicable to at least the targets where *TC* is applicable. Specifically:
  - If the declaration of *T* has a (meta-)annotation m_1 that corresponds to `java.lang.annotation.Target`, then the declaration of *TC* must have a (meta-)annotation m_2 that corresponds to `java.lang.annotation.Target`, and m_2 must have an element whose value indicates a set of program element types which is the same as, or a subset of, the set of program element types indicated by the value of the element in m_1.

    For the purpose of this rule, the program element type `java.lang.annotation.ElementType.ANNOTATION_TYPE` is a subset of the program element type `java.lang.annotation.ElementType.TYPE` since annotation types are logically reference types.

    This rule implements the policy that an annotation type may be *repeatable* on only some of the kinds of program element where the annotation type is *applicable*.

    Assume `Foo` is a repeatable annotation type and `FooContainer` is its containing annotation type. An annotation type "has a target" if the annotation type's declaration has a (meta-)annotation that corresponds to `java.lang.annotation.Target`. Then:

    - If `Foo` has no target and `FooContainer` has no target, then `@Foo` may appear at any annotatable location.
    - If `Foo` has no target but `FooContainer` has a target, then `@Foo` may only be repeated on program elements where `@FooContainer` may appear.
    - If `Foo` has a target, then in the judgment of the designers of the Java programming language, `FooContainer` *must* be declared with knowledge of that target. Specifically, `FooContainer`'s target must be the same as, or a subset of, `Foo`'s target.

    For example, if `Foo`'s target is fields and methods, then `FooContainer` may legitimately restrict its own target to just fields (preventing `@Foo` from being repeated on methods) or just methods (preventing `@Foo` from being repeated on fields). However, `FooContainer` must not restrict its own target to just, say, parameters, because parameters are not germane to `Foo` and their mention by `FooContainer` indicates a misconception of `Foo`'s purpose. Similarly, `FooContainer` must not restrict its own target to fields and parameters, as this is not deemed a legitimate request to make `Foo` repeatable on fields only (the intersection of `Foo`'s target and `FooContainer`'s target).

- 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`.

- 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`.

It is a compile-time error if an annotation type *T* is (meta-)annotated with an `@ContainedBy` annotation whose `value` element indicates a type other than the *containing annotation type of T*.

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

Consider the following declarations:

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

@ContainerFor(Foo.class)
@interface FooContainer { Object[] value(); }
```

Compiling the `Foo` declaration produces a compile-time error because `Foo` uses `@ContainedBy` to nominate `FooContainer` as its containing annotation type, but `FooContainer` is not in fact the *containing annotation type of* `Foo`. (The return type of `FooContainer.value()` is not `Foo[]`.)

Compiling the `FooContainer` declaration produces a compile-time error because `FooContainer` uses `@ContainerFor` to nominate itself as the *containing annotation type of* `Foo`, but again, `FooContainer` is not in fact the *containing annotation type of* `Foo`.

Consider the following declarations:

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

@ContainerFor(Bar.class)
@interface FooContainer { Foo[] value(); }

@ContainedBy(QuuxContainer.class)
@interface Bar {}
```

Compiling the `Foo` declaration produces a compile-time error because `Foo` uses `@ContainedBy` to nominate `FooContainer` as its containing annotation type, but `FooContainer` is not in fact the *containing annotation type of* `Foo`. (The `@ContainerFor` (meta-)annotation on `FooContainer` does not indicate `Foo.class`.)

Compiling the `FooContainer` declaration produces a compile-time error because `FooContainer` uses `@ContainerFor` to nominate itself as the *containing annotation type of* `Bar`, but `FooContainer` is not in fact the *containing annotation type of* `Bar`. (The return type of `FooContainer.value()` is not `Bar`, and the `@ContainedBy` (meta-)annotation on `Bar` does not indicate `FooContainer.class`.)

An annotation type can have only one containing annotation type.

This is by design. Any scheme that associates more than one containing annotation type with a given annotation type declaration causes an undesirable choice at compile-time, when multiple annotations of a repeatable annotation type are logically replaced with a "container". Also, if an annotation type declaration was (meta-)annotated with multiple `@ContainedBy` annotations, then the declaration of `java.lang.annotation.ContainedBy` would have to be (meta-)annotated with `@ContainedBy`; such recursion would unduly complicate implementations.

An annotation type can be the containing annotation type of only 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 also implied by the requirement on the `value()` method of the containing annotation type. Specifically, if an annotation type *`A`* specified itself (via `@ContainedBy`) 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 (JLS 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
@ContainedBy(FooContainer.class)
@interface Foo { int value(); }

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

// FooContainerContainer: Containing annotation type of FooContainer
@ContainerFor(FooContainer.class)
@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). 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 *including when synthesized as a container annotation (§9.7)*) unless: ...

9.6.3.8 `@ContainedBy`

The annotation type `java.lang.annotation.ContainedBy` is used to indicate the *containing annotation type* (§9.6) for the annotation type whose declaration is (meta-)annotated with `@ContainedBy`.

> Note that an `@ContainedBy` 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`.

9.6.3.9 `@ContainerFor`

The annotation type `java.lang.annotation.ContainerFor` is used to indicate the *repeatable annotation type* (JLS 9.6) whose own declaration is (meta-)annotated with an `@ContainedBy` annotation that indicates the annotation type whose declaration is (meta-)annotated with `@ContainerFor`.

9.7 Annotations

It is a compile-time error if a declaration is annotated with more than one annotation of a given annotation type, *unless the annotation type is repeatable (§9.6), and the annotated declaration is a valid target (JLS 9.6.3.1) of both the repeatable annotation type and the repeatable annotation type's containing annotation type*.

> This rule implements the policy that an annotation may repeat at only some of the locations where the annotation may appear. See §9.6 for more details.

If and only if a declaration has multiple annotations of a given repeatable annotation type `T`, then those annotations are logically equivalent to a single annotation `a` whose type is the *containing annotation type of* `T`. `a` is called a *container annotation*.

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

> It is conventional to write multiple annotations of a repeatable annotation type 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.
>
> A container annotation is considered synthesized (compiler-generated but user-visible), not synthetic (compiler-generated and user-invisible).

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 synthesized, 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.

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()

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

- Add `get[Declared]Annotations(Class<T>)`, the repeating-annotation-aware version of `getAnnotation(Class<T>)`.
- Add `getDeclaredAnnotation(Class<T>)` for completeness. The behavior is that of `getAnnotation(Class<T>)` but ignoring inherited annotations on classes.

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(Class<T>)	getAnnotations(Class<T>)
getDeclaredAnnotations()	getAnnotations()

The declaration of a class type may inherit annotations from 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`* on the superclass.

When *T* is repeatable (§9.6), the question is how to extend the policy to handle multiple annotations of type *T* on the superclass or subclass. Oracle proposes the following policy for Java SE 8:

- If a class declaration does not have any "directly present" annotations of type *T*, the class declaration may have "present" annotations of type *T* due to inheritance.
- If a class declaration does have one or more "directly present" annotations of type *T*, they are deemed to "override" *every* annotation of type *T* on the superclass.

The policy for Java SE 8 is reified in the following definitions of *directly present* and *present*. The phrase "annotations of a program element" is taken to mean the `RuntimeVisibleAnnotations` or `RuntimeVisibleParameterAnnotations` attribute associated with that element.

An annotation *A* is *directly present* on an element *E* if either:

- The annotations of *E* contain *A*; or
- The annotations of *E* contain exactly one annotation *C* whose type is the containing annotation type of *A*'s type (§9.6) and whose `value` element contains *A*.

An annotation *A* is *present* on an element *E* if either:

- *A* is *directly present* on *E*; or
- There are no annotations of *A*'s type which are *directly present* on *E*, and *E* is a class, and *A*'s type is inheritable (JLS 9.6.3.3), and *A* is *present* on the superclass of *E*.

In addition, Oracle proposes to:

- Refine the specification of `getAnnotation(Class<T>)` to look through a container annotation (if present) if the supplied annotation type is repeatable.
- Refine the specifications of `get[Declared]Annotations()` to look through container annotations and return the annotations contained therein.
- Refine the implementations of `get[Declared]Annotations()` to return all annotations of an annotation type on an element, rather than just one, to support the case where a container annotation is looked through. This behavior is permitted by the methods' specifications in Java SE 7.
- Refine the specification of `isAnnotationPresent(`*X*`)` to be equivalent to:

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

If the reflection libraries of the Java SE platform load an annotation type `T` which is (meta-)annotated with an `@ContainedBy` annotation whose `value` element indicates a type other than the *containing annotation type of* `T`, then an exception of type `java.lang.annotation.AnnotationFormatError` is thrown.

If the reflection libraries of the Java SE platform load an annotation type `TC` which is (meta-)annotated with an `@ContainerFor` annotation whose `value` element indicates the type `T`, but `TC` is not the *containing annotation type of* `T`, then an exception of type `java.lang.annotation.AnnotationFormatError` is thrown.

> Throwing these exceptions to indicate an ill-formed relationship between a prospective repeatable annotation type and its prospective containing annotation type mirrors the compile-time rules in §9.6.

Example 1.2-1. Repeating an annotation is behaviorally compatible

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

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

SE 7 behavior of SE 7 methods:

```
A.class.getAnnotation(Foo.class)          = @Foo(1)
A.class.getAnnotation(FooContainer.class) = null
A.class.getAnnotations()                  = [ @Foo(1) ]
A.class.getDeclaredAnnotations()          = [ @Foo(1) ]

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

The behavior of these methods in SE 8 is unchanged.

Now suppose the `Foo` annotation type 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 {}
```

To support a legacy consumer running on SE 8, we do not expose the synthesized `@FooContainer` via `get[Declared]Annotations()`. Instead, reflection looks through `@FooContainer` on A to return answers at least as good as SE 7.

SE 8 behavior of SE 7 methods:

```
A.class.getAnnotation(Foo.class)          = @Foo(1)
A.class.getAnnotation(FooContainer.class) = @FooContainer(...)    // NEW
A.class.getAnnotations()                  = [ @Foo(1), @Foo(2) ]  // NEW
A.class.getDeclaredAnnotations()          = [ @Foo(1), @Foo(2) ]  // NEW

B.class.getAnnotation(Foo.class)          = @Foo(1)
B.class.getAnnotation(FooContainer.class) = @FooContainer(...)    // NEW
B.class.getAnnotations()                  = [ @Foo(1), @Foo(2) ]  // NEW
B.class.getDeclaredAnnotations()          = [ ]
```

SE 8 behavior of SE 8 methods:

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

A.class.getAnnotations(Foo.class)                  = [ @Foo(1), @Foo(2) ]
A.class.getAnnotations(FooContainer.class)         = [ @FooContainer(...) ]
A.class.getDeclaredAnnotations(Foo.class)          = [ @Foo(1), @Foo(2) ]
A.class.getDeclaredAnnotations(FooContainer.class) = [ @FooContainer(...) ]

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

B.class.getAnnotations(Foo.class)                  = [ @Foo(1), @Foo(2) ]
B.class.getAnnotations(FooContainer.class)         = [ @FooContainer(...) ]
B.class.getDeclaredAnnotations(Foo.class)          = [ ]
B.class.getDeclaredAnnotations(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 behavior of the following SE 7 methods in SE 8 is the same as in SE 7, because in essence, `@Foo(3)` on `B` was always deemed to "override" every `@Foo` annotation on `A`:

SE 8 behavior of SE 7 methods:

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

SE 8 behavior of SE 8 methods:

```
B.class.getDeclaredAnnotation(Foo.class)           = @Foo(3)
B.class.getDeclaredAnnotation(FooContainer.class)  = null

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

Example 1.2-2. Idiomatic container continues to work

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

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

SE 7 behavior of SE 7 methods:

```
A.class.getAnnotation(Foo.class)          = null
A.class.getAnnotation(FooContainer.class) = @FooContainer(...)
A.class.get[Declared]Annotations()        = [ @FooContainer(...) ]
```

Now suppose the `Foo` annotation type is made repeatable with `FooContainer` as its containing annotation type. This "opt-in" by the author of the annotation types has a visible effect on the behavior of reflection, even without recompiling class `A`.

SE 8 behavior of SE 7 methods:

```
A.class.getAnnotation(Foo.class)          = @Foo(1)  // NEW
A.class.getAnnotation(FooContainer.class) = @FooContainer(...)
A.class.get[Declared]Annotations()        = [ @Foo(1), @Foo(2) ]  // NEW
```

SE 8 behavior of SE 8 methods:

```
A.class.getDeclaredAnnotation(Foo.class)             = @Foo(1)
A.class.getDeclaredAnnotation(FooContainer.class)    = @FooContainer(...)
A.class.get[Declared]Annotations(Foo.class)          = [ @Foo(1), @Foo(2) ]
A.class.get[Declared]Annotations(FooContainer.class) = [ @FooContainer(...) ]
```

Note that the presence of a container annotation is visible via `get[Declared]Annotation(FooContainer.class)` and `get[Declared]Annotations(FooContainer.class)`, but not via the legacy `get[Declared]Annotations()`.

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 behavior of SE 7 methods:

```
A.class.getAnnotation(Foo.class)          = @Foo(0)
A.class.getAnnotation(FooContainer.class) = @FooContainer(...)
A.class.get[Declared]Annotations()        = [ @Foo(0), @FooContainer(...) ]
```

Now suppose the `Foo` annotation type is made repeatable with `FooContainer` as its containing annotation type. This "opt-in" by the author of the annotation types has a visible effect on the behavior of reflection, even without recompiling class `A`.

SE 8 behavior of SE 7 methods:

```
A.class.getAnnotation(Foo.class)          = @Foo(0)
A.class.getAnnotation(FooContainer.class) = @FooContainer(...)
A.class.getAnnotations()                  = [ @Foo(0), @Foo(1), @Foo(2) ] // NEW
A.class.getDeclaredAnnotations()          = [ @Foo(0), @Foo(1), @Foo(2) ] // NEW
```

SE 8 behavior of SE 8 methods:

```
A.class.getDeclaredAnnotation(Foo.class)             = @Foo(0)
A.class.getDeclaredAnnotation(FooContainer.class)    = @FooContainer(...)
A.class.get[Declared]Annotations(Foo.class)       = [ @Foo(0), @Foo(1), @Foo(2) ]
A.class.get[Declared]Annotations(FooContainer.class) = [ @FooContainer(...) ]
```

Let us return to a declaration with one `@Foo` annotation and an `@FooContainer` annotation written by hand to serve as an idiomatic container for `@Foo` annotations. That is, the `Foo` annotation type is not repeatable - but now assume it *is* inheritable:

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

SE 7 behavior of SE 7 methods:

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

Now suppose the `Foo` annotation type is made repeatable with `FooContainer` as its containing annotation type. (Per §9.6, `FooContainer` must be inheritable because `Foo` is inheritable.) This "opt-in" by the author of the annotation types has a visible effect on the behavior of reflection, even without recompiling class `A`.

SE 8 behavior of SE 7 methods:

```
B.class.getAnnotation(Foo.class)          = @Foo(0)
B.class.getAnnotation(FooContainer.class) = @FooContainer(...)
B.class.getAnnotations()                  = [ @Foo(1), @Foo(2) ] // NEW
B.class.getDeclaredAnnotations()          = [ @Foo(1), @Foo(2) ] // NEW
```

SE 8 behavior of SE 8 methods:

```
B.class.getDeclaredAnnotation(Foo.class)          = @Foo(1)
B.class.getDeclaredAnnotation(FooContainer.class) = @FooContainer(...)

B.class.getDeclaredAnnotations(Foo.class)         = [ @Foo(1), @Foo(2) ]
B.class.getDeclaredAnnotations(FooContainer.class = [ @FooContainer(...) ]
B.class.getAnnotations(Foo.class)                 = [ @Foo(1), @Foo(2) ]
B.class.getAnnotations(FooContainer.class)        = [ @FooContainer(...) ]
```

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 a given type, 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`, refine the specification of `getAnnotation(Class<T>)` for consistency with the same method in `java.lang.reflect.AnnotatedElement`. The method will look through a container annotation (if present) if the supplied annotation type is repeatable.
- In `javax.lang.model.element.Element`, add `getAnnotations(Class<T>)` and `getAnnotations()` for consistency with `java.lang.reflect.AnnotatedElement`. As described in §1.2, `getAnnotations(Class<T>)` will expose a container annotation if the supplied

annotation type is a containing annotation type, but will look through a container annotation if the supplied annotation type is a repeatable annotation type. `getAnnotations()` always looks through container annotations.

- In `javax.lang.model.element.Element`, refine the implementation of `getAnnotationMirrors()` to return mirrors for all annotations which are *directly present* on an element. This method offers a literal representation of source code, so any container annotations whose logical presence is implied by §9.7 are *not* exposed.
- In `javax.lang.model.util.Elements`, refine the implementation of `getAllAnnotationMirrors(Element)` to return mirrors for all annotations which are *present* on an element. This method offers a literal representation of source code, so any container annotations whose logical presence is implied by §9.7 are *not* exposed.

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 runtime 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. 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 -g: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*

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 Java SE releases, 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`, which may appear only, and at most once, in the `attributes` table of a `method_info` structure:

```
MethodParameters_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 parameters_count;
    {   u2 parameter_name_index;
        u4 parameter_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 items that "enhance" method parameters (e.g. with annotations, or names) with the method descriptor's parameters is traditionally done by the reflection libraries of the Java SE platform.

`parameters_count` is one byte because JVMS 4.3.4 limits a method descriptor to 255 parameters.

`parameters`

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

`parameter_name_index`

The value of the `parameter_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 name of a method parameter.

Perhaps a nameless parameter could be represented with a `parameter_name_index` of zero.

`parameter_flags`

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

`0x0010 (ACC_FINAL)`

Indicates that the method parameter was declared `final`.

`0x1000 (ACC_SYNTHETIC)`

Indicates that the method parameter is synthetic; not physically or logically present in source code.

`0x10000 (ACC_SYNTHESIZED)`

Indicates that the method parameter is synthesized; logically present but not physically present in source code.

`parameter_flags` uses the traditional values for `ACC_FINAL` and `ACC_SYNTHETIC`. To flag a synthesized method parameter, we have a problem because the only unused flag bit in a u2 is 0x8000. This is already claimed by the Java Module System spec as `ACC_MODULE` in `ClassFile.access_flags`, indicating that the `ClassFile` represents a module rather than a class. It is not feasible to represent `ACC_MODULE` with 0x10000 because the access_flags of _every `ClassFile`_ would need to expand from u2 to u4 (we don't use u3's). It might be possible to represent "module-ness" by other means (e.g. a marker attribute), in which case 0x8000 could be used for `ACC_SYNTHESIZED`. Certainly it would be nice if the "synthesized" concept could be encoded in a u2, since the concept is expected to spread beyond method parameters and module dependences.

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 which are typed in the descriptor, it would unduly complicate the `ClassFile` format given that the vast majority of compilers are likely to produce a `MethodParameters` attribute denoting every parameter which is 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 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 method parameter (name, immutability, etc)
  public Executable getDeclaringExecutable()
  public int        getModifiers()
  public String     getName()
  public Type       getParameterizedType()
  public Class<?>   getType()
  public boolean    isSynthesized()
  public boolean    isSynthetic()
  public boolean    isVarArgs()

  // AnnotatedElement methods
  public <T extends Annotation> T getDeclaredAnnotation(Class<T>)
  public <T extends Annotation> T getDeclaredAnnotations(Class<T>)
  public Annotation[]             getDeclaredAnnotations()
  public <T extends Annotation> T getAnnotation(Class<T>)
  public <T extends Annotation> T getAnnotations(Class<T>)
  public Annotation[]             getAnnotations()
  public boolean isAnnotationPresent(Class<? extends Annotation>)
}
```

The specification of `toString()` is:

```
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, 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.
```

2.3 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 synthesized, synthetic, or variable arity.