Annotations on Java® Types

JSR 308 Expert Group

2013-10-16

Specification: JSR-000308 Annotations on Java® Types ("Specification")
Version: Java SE 8
Status: Public Review
Release: October 2013

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Oracle USA, Inc. ("Oracle") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Oracle's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;

(ii) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and

(iii) includes the following notice: "This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

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

The Java Language Specification

1.1 Core Grammar Changes

Annotation on Java® types are enabled by adding the term *{Annotation}* throughout the *PrimitiveType* and *ReferenceType* hierarchies, as close as possible to the identifier which denotes the simple name of a type. The *Type* production is not modified.

The syntax *{x}* on the right-hand side of a production denotes zero or more occurrences of *x*.

The syntax *[x]* on the right-hand side of a production denotes zero or one occurrences of *x*. That is, *x* is an *optional symbol*. The alternative which contains the optional symbol actually defines two alternatives: one that omits the optional symbol and one that includes it.

The grammar below makes it possible to distinguish annotatable locations from non-annotatable locations on the basis of syntax alone - a considerable achievement. The following paragraphs give rationale.

In JLS7 4.3, *ClassType* and *InterfaceType* were stratified into either a basic *TypeName* nonterminal or a complex *TypeDeclSpecifier* nonterminal which relied on *TypeName*. The obvious way to allow annotations in class and interface types would be to sprinkle *{Annotation}* throughout *TypeDeclSpecifier* and *TypeName*. However, there are two disadvantages to this approach:

- First, it builds on *TypeDeclSpecifier* which is one of the more obscure nonterminals in the *The Java Language Specification*. It is used in only a handful of situations: the `extends` and `implements` clauses of a class declaration (8.1.4, 8.1.5), the `extends` clause of an interface declaration (9.1.3), and the syntax of a class instance creation expression (15.9). The reason for its use is to prohibit wildcard type arguments (though 9.1.3 actually fails to do so), but this can be achieved equally well without a dedicated non-terminal.
- Second, it means that *TypeName* in 4.3 is logically distinct from *TypeName* in 6.5. Historically, *TypeName* in 4.3 was reused in 6.5 to find occurrences of type names, by pattern matching against dotted identifier sequences throughout the Java programming language. Unfortunately, this includes occurrences which JSR 308 regards as "scoping

mechanisms" and which should therefore not be annotated. (An example of a scoping mechanism is the *TypeName* occurring in an `import` declaration.) It would be confusing for *TypeName* to sometimes match an annotated identifier sequence that denotes a type use, while at other times match an unannotated identifier sequence that denotes a scoping mechanism.

A more straightforward approach is to remove *TypeName* and *TypeDeclSpecifier* from the *ReferenceType* hierarchy, and build *ClassType* and *InterfaceType* directly from *Identifier*. It is obvious that an *Identifier* in *ClassType* and *InterfaceType* represents a "type use", so we can precede it with *{Annotation}*. It is also obvious that the *Identifier* in a type variable represents a "type use", as well as the wildcard in a type argument, so they too can be preceded with *{Annotation}*. The result is a very natural and consistent integration of annotations into the *ReferenceType* hierarchy. The well-formedness of a parameterized type that was hitherto embodied by semantic constraints on a *TypeDeclSpecifier* in 4.3 can be moved to the more natural location of 4.5 "Parameterized Types"; the additional constraints on *TypeDeclSpecifier* in 8.1.4, 8.1.5, 9.1.3, and 15.9 can point to centralized rules in 4.5.

A complete list of contexts which use *Type*, *ReferenceType*, *ClassType*, and *InterfaceType* is given in §4.11 "Where Types Are Used". These contexts correspond exactly to the locations identified by JSR 308 as "type uses".

The *TypeName* nonterminal in 6.5 is unchanged. It does not allow annotations on its identifiers, so any context whose grammar uses *TypeName* cannot be annotated. Happily, these contexts correspond exactly to the locations identified by JSR 308 as "scoping mechanisms". The contexts are found as the initial entries in the list in 6.5.1 which classifies a dotted identifier sequence as a *TypeName*. (A static member access like `Foo.x` is classified as an *ExpressionName* but `Foo` is eventually classified as a *TypeName*.) The same contexts are found in 6.1, where they form the first nine "non-generic" contexts.

4.2 Primitive Types and Values

PrimitiveType:
  {Annotation} NumericType
  {Annotation} `boolean`

4.3 Reference Types and Values

ReferenceType:
  ClassOrInterfaceType
  TypeVariable
  ArrayType

ClassOrInterfaceType:
  ClassType
  InterfaceType

ClassType:
  {Identifier .} {Annotation} Identifier [TypeArguments]
  ClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

InterfaceType:
  {Identifier .} {Annotation} Identifier [TypeArguments]
  ClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

TypeVariable:
  {Annotation} Identifier

ArrayType:
  PrimitiveType {{Annotation} `[ ]`*}*
  ClassOrInterfaceType {{Annotation} `[ ]`*}*
  TypeVariable {{Annotation} `[ ]`*}*

4.4 Type Variables

A type variable is introduced by the declaration of a *type parameter* of a generic class, interface, method, or constructor (8.1.2, 9.1.2, 8.4.4, 8.8.4).

TypeParameter:
  {TypeParameterModifier} Identifier [TypeBound]

TypeParameterModifier:
  Annotation

The right-hand side for *TypeParameter* is a correction to JLS3 and JLS7. They built a *TypeParameter* from a *TypeVariable*, but this is incorrect because a type parameter is a declared entity and so is denoted with an *Identifier*. Separately, see 1.2 for the rationale behind *TypeParameterModifier*.

The appearance of *{Annotation}* in *TypeVariable*, *ClassType*, and *InterfaceType* means that annotations come "for free" on types which denote the bounds of a type parameter. No changes are required to the *TypeBound* or *AdditionalBound* productions.

4.5.1 Type Arguments and Wildcards

TypeArgument:
  ReferenceType
  Wildcard

Wildcard:
  {Annotation} `?` *[WildcardBounds]*

WildcardBounds:
  `extends` *ReferenceType*
  `super` *ReferenceType*

4.11 Where Types Are Used

Types are used in most kinds of declaration and in certain kinds of expression. Specifically, there are 15 *type contexts* where types are used:

- In declarations:
  1. A type in the `extends` or `implements` clause of a class declaration (8.1.4, 8.1.5, 8.5, 9.5)
  2. A type in the `extends` clause of an interface declaration (9.1.3, 8.5, 9.5)
  3. The return type of a method (including the type of an element of an annotation type) (8.4.5, 9.4, 9.6.1) [*]
  4. A type in the `throws` clause of a method or constructor (8.4.6, 8.8.5, 9.4)
  5. A type in the `extends` clause of a type parameter declaration of a generic class, interface, method, or constructor (8.1.2, 9.1.2, 8.4.4, 8.8.4)
  6. The type in a field declaration of a class or interface (including an enum constant) (8.3, 9.3, 8.9.1) [*]
  7. The type in a formal parameter declaration of a method or constructor (8.4.1, 8.8.1, 9.4) [*]
  8. The type of the receiver parameter of a method (8.4.1)
  9. The type in a local variable declaration (14.4, 14.14.1, 14.14.2, 14.20.3) [*]
  10. The type in an exception parameter declaration (14.20) [*]
- In expressions:
  11. A type in the explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (8.8.7.1, 15.9, 15.12)
  12. In an unqualified class instance creation expression, as the class type to be instantiated (15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5)

13. The element type in an array creation expression (15.10)
14. The type in the cast operator of a cast expression (15.16)
15. The type that follows the `instanceof` relational operator (15.20.2)

Types are also used as:

- The element type of an array type in any of the above contexts; and
- A non-wildcard type argument, or a bound of a bounded wildcard type argument, of a parameterized type in any of the above contexts.

Finally, there is an implicit use of a type as the simple name of the class in a constructor declaration. This implicit use indicates the type of the constructed object.

> The five type contexts marked with a [*] each occupy a syntactic location in a program which is also a declaration context (JLS 9.6.3.1). This occurs where the modifiers for a declaration immediately precede the type of the declared entity: in field declarations (including enum constants), formal parameter declarations and exception parameter declarations (which historically are grouped into one declaration context), local variable declarations, and method/constructor declarations (including elements of annotation types). JLS 9.7.4 explains how an annotation in such a location is determined to appear in a type context or a declaration context (or both).

In any context where a type is used, it is possible to annotate the keyword denoting a primitive type or the *Identifier* denoting the simple name of a reference type. It is also possible to annotate an array type by writing an annotation to the left of the `[` at the desired level of nesting in the array type. Annotations in all these locations are called *type annotations*, and are specified in JLS 9.7.4.

- `@Foo int[] f;` annotates the primitive type `int`
- `int @Foo [] f;` annotates the array type `int[]`
- `int @Foo [][] f;` annotates the array type `int[][]`
- `int [] @Foo [] f;` annotates the array type `int[]` which is the component type of the array type `int[][]`

The meaning of types in type contexts is given by:

- 4.2, for primitive types
- 4.4, for type parameters
- 4.5, for class and interface types that are parameterized, or appear either as type arguments in a parameterized type or as bounds of wildcard type arguments in a parameterized type

- 4.8, for class and interface types that are raw
- 4.9, for intersection types in the bounds of type parameters
- 6.5, for class and interface types in contexts where genericity is unimportant (6.1)
- 10.1, for array types

Some type contexts restrict how a reference type may be parameterized:

- The following type contexts require that if a type is a parameterized reference type, it has no wildcard type arguments:
  - In an `extends` or `implements` clause of a class declaration (8.1.4, 8.1.5)
  - In an `extends` clause of an interface declaration (9.1.3)
  - In an unqualified class instance creation expression, as the class type to be instantiated (15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5)
  - In addition, no wildcard type arguments are permitted in the explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (8.8.7.1, 15.9, 15.12).
- The following type contexts require that if a type is a parameterized reference type, it has only unbounded wildcard type arguments (i.e. it is a reifiable type) :
  - As the element type in an array creation expression (15.10)
  - As the type that follows the `instanceof` relational operator (15.20.2)
- The following type contexts disallow a parameterized reference type altogether, because they interpret a type as an exception type which is semantically non-generic (6.1):
  - As the type of an exception that can be thrown by a method or constructor (8.4.6, 8.8.5, 9.4)
  - In an exception parameter declaration (14.20)

1.2 Consistent Use of Modifiers on Declarations

Most kinds of declaration include modifiers, denoted as a nonterminal ending in *Modifier*. For example, a class declaration includes class modifiers (8.1.1) which are denoted as *ClassModifier*. It is helpful to rely on the presence of modifiers when defining the term "declaration annotation" in 9.7. However, there are two kinds of declaration which have no modifiers, but rather allow annotations directly in their grammar: package declarations and enum constant declarations (7.4.1, 8.9.1). We introduce *Modifier* nonterminals to encapsulate annotations on these declarations, allowing us to speak uniformly of annotations as modifiers for declarations.

PackageDeclaration:
  {PackageModifier} `package` *PackageName* `;`

EnumConstant:
  {EnumConstantModifier} Identifier [Arguments] [ClassBody]

PackageModifier:
  Annotation

EnumConstantModifier:
  Annotation

1.3 Ambiguity Between Modifiers and Types

Modifiers may appear at the start of a field declaration, a formal or exception parameter declaration, or a local variable declaration (including a loop variable of a `for` statement and a resource variable of a `try`-with-resources statement). A modifier may be an annotation, yet the type in a variable declaration may also begin with an annotation.

Likewise, modifiers may appear at the start of a method/constructor declaration (including an element of an annotation type declaration). A modifier may be an annotation, yet the return type of a method (including an element of an annotation type) may also begin with an annotation.

We remove the ambiguity by defining the grammars of these declarations to use "unannotated types". Note that an annotation which is syntactically handled as a modifier on a declaration may be deemed to apply to the type used in the declaration by JLS 9.7.4.

FieldDeclaration:
  {FieldModifier} UnannType VariableDeclaratorList

FormalParameter:
  {VariableModifier} UnannType VariableDeclaratorId

LocalVariableDeclaration:
  {VariableModifier} UnannType VariableDeclaratorList

EnhancedForStatement:
  `for` `(` *{VariableModifier} UnannType VariableDeclaratorId*
    `:` *Expression* `)` *Statement*

CatchType:
  UnannClassType {`|` *ClassType}*

Result:
  UnannType
  `void`

AnnotationTypeElementDeclaration:
  {AbstractMethodModifier} UnannType Identifier `(` `)` *[Dims]*
    [DefaultValue] `;`
  ...

The definition of the *UnannType* hierarchy, which parallels the *Type* hierarchy, is to be given in 8.3 "Field Declarations".

UnannType:
  UnannPrimitiveType
  UnannReferenceType

UnannPrimitiveType:
  NumericType
  `boolean`

UnannReferenceType:
  UnannClassOrInterfaceType
  UnannTypeVariable
  UnannArrayType

UnannClassOrInterfaceType:
  UnannClassType
  UnannInterfaceType

UnannClassType:
  Identifier [TypeArguments]
  Identifier . {Identifier .} {Annotation} Identifier [TypeArguments]
  UnannClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

UnannInterfaceType:
Identifier [TypeArguments]
Identifier . {Identifier .} {Annotation} Identifier [TypeArguments]
UnannClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

UnannTypeVariable:
Identifier

UnannArrayType:
UnannType {Annotation} `[ ]`

1.4 Special Parameter Contexts

There are two considerations for formal parameters. First, the "..." syntax that indicates a variable-arity method is a synonym for an array type, so it may be annotated. Second, the object for which an instance method or inner class constructor is invoked - the *receiver* - has a type which may be annotated.

8.4.1 Formal Parameters

FormalParameterList:
LastFormalParameter
FormalParameters `,` *LastFormalParameter*

FormalParameters:
ReceiverParameter {`,` *FormalParameter}*
FormalParameter {`,` *FormalParameter}*

ReceiverParameter:
{VariableModifier} UnannType {Identifier .} `this`

FormalParameter:
{VariableModifier} UnannType VariableDeclaratorId

LastFormalParameter:
{VariableModifier} UnannType {Annotations} ... VariableDeclaratorId
FormalParameter

The *receiver parameter* of an instance method or inner class constructor is an optional syntactic device to represent the object for which the method or constructor is invoked. It exists solely to allow the type of the object to be denoted in source code. It is not a formal parameter. More precisely, it is not a declaration of any kind of variable (4.12.3), is never bound to any value passed as an argument in a method invocation expression, and has no effect whatsoever at run time.

A receiver parameter may only appear in the *FormalParameters* of an instance method or an inner class constructor; otherwise, a compile-time error occurs.

The name of the receiver parameter must either be this, or the text of a qualified this expression (15.8.3) which would, if present in the body of the method or constructor, denote the class of the method or constructor's receiver; otherwise, a compile-time error occurs.

The type of the receiver parameter must be the class or interface in which the method or constructor is declared; otherwise, a compile-time error occurs.

1.5 Declarations and Names

6.1 Declarations

Generic classes and generic interfaces are declared entities.

The declaration of a generic type class C<T> ... or interface C<T> ... declares two entities: C<T>, a generic type, and C, the corresponding non-generic type. The subtyping relationship between a generic type and its corresponding non-generic type is given in 4.10.

The generic type declaration includes the identifier C whose meaning depends on the context where it appears:

- In contexts where genericity is unimportant, such as single-type-import and static member access, the identifier C denotes the non-generic type C.
- In contexts where genericity is important, the identifier C denotes either:
  - The raw type C which is the erasure (4.6) of the generic type C<T>; or
  - A parameterized type which is a particular invocation (4.5) of the generic type C<T>.

The *non-generic contexts* are as follows:

1. In a single-type-import declaration (7.5.1)
2. To the left of the . in a single-static-import declaration (7.5.3)
3. To the left of the . in a static-import-on-demand declaration (7.5.4)

4. To the left of the `(` in a constructor declaration (8.8)
5. After the `@` sign in an annotation (9.7)
6. To the left of `.class` in a class literal (15.8.2)
7. To the left of `.this` in a qualified `this` expression (15.8.4)
8. To the left of `.super` in a qualified superclass field access expression (15.11.2)
9. In a qualified expression name in a postfix expression (15.14.1)
10. In a `throws` clause of a method or constructor (8.4.6, 8.8.5, 9.4)
11. In an exception parameter declaration (14.20)

The non-generic contexts come in three flavors:

- The first eight non-generic contexts correspond to the first eight syntactic contexts for a *TypeName* in 6.5.1.
- The ninth non-generic context is a postfix expression, where a qualified *ExpressionName* such as `C.x` may include a *TypeName* `C` to denote static member access.
- The tenth and eleventh non-generic contexts involve an exception type in `throws` clauses and `catch` clauses. Exception types are semantically non-generic (8.1.2).

From the viewpoint of JSR 308, the first nine non-generic contexts play the role of "scoping mechanism" rather than "type use", so should not allow annotations. Happily, these contexts are exactly those that use (directly or indirectly) the *TypeName* production which does not allow annotations. We effectively prohibit annotations in undesirable locations purely on the basis of syntax.

From the viewpoint of JSR 308, the tenth and eleventh non-generic contexts represent a "type use" rather than a "scoping mechanism". Happily, the syntax of `throws` clauses and `catch` clauses allows *ClassType*, whose syntax in turn allows annotations. Note that annotations will never decorate type arguments in these two contexts, because exception types cannot be generic and so a `throws` clause never contains parameterized types.

6.5.1 Syntactic Classification of a Name According to Context

A name is syntactically classified as a *TypeName* in these contexts:

- The first eight non-generic contexts (6.1):
  1. In a single-type-import declaration (7.5.1)
  2. To the left of the `.` in a single-static-import declaration (7.5.3)
  3. To the left of the `.` in a static-import-on-demand declaration (7.5.4)

4. To the left of the `(` in a constructor declaration (8.8) [+]
5. After the `@` sign in an annotation (9.7)
6. To the left of `.class` in a class literal (15.8.2) [*]
7. To the left of `.this` in a qualified `this` expression (15.8.4) [++]
8. To the left of `.super` in a qualified superclass field access expression (15.11.2) [++]

- As the *Identifier* or dotted sequence of *Identifiers* that constitutes any *ReferenceType* (including a *ReferenceType* to the left of the brackets in an array type, or to the left of the < in a parameterized type, or in a non-wildcard type argument of a parameterized type, or in an `extends` or `super` clause of a wildcard type argument of a parameterized type) in the 15 contexts where types are used (§4.11):

1. In the `extends` or `implements` clause of a class declaration (8.1.4, 8.1.5, 8.5, 9.5)
2. In the `extends` clause of an interface declaration (9.1.3)
3. The return type of a method (8.4, 9.4) (including the type of an element of an annotation type (9.6.1))
4. In the `throws` clause of a method or constructor (8.4.6, 8.8.5, 9.4) [**]
5. The `extends` clause of a type parameter declaration of a generic class, interface, method, or constructor (8.1.2, 9.1.2, 8.4.4, 8.8.4)
6. The type in a field declaration of a class or interface (8.3, 9.3)
7. In a formal parameter declaration of a method or constructor (8.4.1, 8.8.1, 9.4)
8. The type of the receiver parameter of a method (8.4.1)
9. The type in a local variable declaration (14.4, 14.14.1, 14.14.2, 14.20.3)
10. The type in an exception parameter declaration (14.20) [+++]
11. In an explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (8.8.7.1, 15.9, 15.12)
12. In an unqualified class instance creation expression, either as the class type to be instantiated (15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5)

13. The element type in an array creation expression (15.10)
14. The type in the cast operator of a cast expression (15.16)
15. The type that follows the `instanceof` relational operator (15.20.2)

The extraction of a *TypeName* from the identifiers of a *ReferenceType* in the 15 contexts above is intended to apply recursively to all sub-terms of the *ReferenceType*, such as its element type and any type arguments.

For example, suppose a field declaration uses the type `p.q.Foo[]`. The brackets of the array type are ignored, and the term `p.q.Foo` is extracted as a dotted sequence of *Identifiers* to the left of the brackets in an array type, and classified as a *TypeName*. A later step determines which of `p`, `q`, and `Foo` is a type name or a package name.

As another example, suppose a cast operator uses the type `p.q.Foo<? extends String>`. The term `p.q.Foo` is again extracted as a dotted sequence of *Identifiers*, this time to the left of the < in a parameterized type, and classified as a *TypeName*. The term `String` is extracted as an *Identifier* in an `extends` clause of a wildcard type argument of a parameterized type, and classified as a *TypeName*.

[+] For a constructor declaration, the grammar since JLS1 has used the undefined nonterminal *SimpleTypeName* to visually hint that a simple name of a type is required. A compile-time error reinforces the requirement. However, 6.5.1 never classified the name of a type in a constructor declaration context as a *TypeName*; we rectify that oversight here. (As a side note, 6.2 describes a constructor declaration as using the name of an existing type (versus introducing a new name like most declarations), and in fact it is the only context in the language where a simple name, not a qualified name, must be used.)

[*] For a class literal expression, JLS7's grammar uses *Type* before `.class`, and 15.8.2 requires a semantic error if a parameterized type or type variable is used. However, javac assumes a *TypeName* before `.class`, as evidenced by the fact that a syntax error occurs if a non-ground type is used. JLS8 should follow javac by defining a class literal expression as *TypeName*`.class`. JLS8 still needs a semantic error to prohibit type variables, which are syntactically valid before `.class`.

[++] For a qualified `this` expression *`T`*`.this` or a qualified field access expression beginning *`T`*`.super`, the grammar since JLS1 has used the undefined nonterminal *ClassName* to visually hint that a class is required rather than an interface. However, 6.5.1 assumed the presence of a *Type* non-terminal (or at least a child of *Type* such as *TypeDeclSpecifier* in a class instance creation expression) and therefore never classified the *`T`* as a type name; we rectify that oversight here.

[**] For a `throws` clause, JLS7's *ExceptionType* uses *TypeName* and javac gives a syntax error if a parameterized type is used. But morally, *ExceptionType* should use *ClassType* rather than *TypeName* (to syntactically allow annotations), and javac should enforce the semantic error required since JLS3 that a thrown type must not be a subtype of `Throwable` (this rule prohibits throwing a parameterized type since all subtypes of `Throwable` are non-generic (8.1.2)). *ExceptionType* should continue to allow *TypeVariable*.

[+++] For a `catch` clause, JLS7's *CatchType* correctly uses *ClassType*, and javac correctly enforces that a parameterized type in a `catch` clause is a semantic error.

1.6 Applicability of Annotation Types

9.6.3.1 `@Target`

An annotation of type `java.lang.annotation.Target` is used on the declaration of an annotation type *`T`* to specify the contexts in which *`T`* is *applicable*. `java.lang.annotation.Target` has a single element, `value`, of type `java.lang.annotation.ElementType[]`, to specify contexts.

For example, an `@Target` meta-annotation which indicates `java.lang.annotation.ElementType.FIELD` specifies that annotations may appear in only one context, namely field declarations.

Annotation types may be applicable in *declaration contexts*, where annotations apply to declarations, or in *type contexts*, where annotations apply to types used in declarations and expressions.

There are eight declaration contexts, each corresponding to an enum constant of `java.lang.annotation.ElementType`:

1. Package declarations (7.4.1)

   Corresponds to `java.lang.annotation.ElementType.PACKAGE`

2. Type declarations, *i.e.*, class, interface, enum, and annotation type declarations (8.1.1, 9.1.1, 8.5, 9.5, 8.9, 9.6)

   Corresponds to `java.lang.annotation.ElementType.TYPE`

   Additionally, annotation type declarations correspond to `java.lang.annotation.ElementType.ANNOTATION_TYPE`

3. Method declarations (including elements of annotation types) (8.4.3, 9.4, 9.6.1)

   Corresponds to `java.lang.annotation.ElementType.METHOD`

4. Constructor declarations (8.8.3)

   Corresponds to `java.lang.annotation.ElementType.CONSTRUCTOR`

5. Type parameter declarations of generic classes, interfaces, methods, and constructors (8.1.2, 9.1.2, 8.4.4, 8.8.4)

   Corresponds to `java.lang.annotation.ElementType.TYPE_PARAMETER`

6. Field declarations (including enum constants) (8.3.1, 9.3, 8.9.1)

   Corresponds to `java.lang.annotation.ElementType.FIELD`

7. Formal and exception parameter declarations (8.4.1, 9.4, 14.20)

   Corresponds to `java.lang.annotation.ElementType.PARAMETER`

8. Local variable declarations (including loop variables of `for` statements and resource variables of `try`-with-resources statements) (14.4, 14.14.1, 14.14.2, 14.20.3)

   Corresponds to `java.lang.annotation.ElementType.LOCAL_VARIABLE`

There are 15 type contexts (§4.11), all represented by the enum constant `TYPE_USE` of `java.lang.annotation.ElementType`.

It is a compile-time error if the same enum constant appears more than once in the `value` element of an annotation of type `java.lang.annotation.Target`.

If an annotation of type `java.lang.annotation.Target` is not present on the declaration of an annotation type *`T`*, then *`T`* is applicable in all declaration contexts except type parameter declarations.

> These contexts are the syntactic locations where annotations were allowed in Java SE 7.

9.6.3.a javadoc for `java.lang.annotation.Target`

Indicates the contexts in which an annotation type is applicable. The *declaration contexts* and *type contexts* in which an annotation type may be applicable are specified in JLS 9.6.3.1, and denoted in source code by enum constants of `java.lang.annotation.ElementType`.

If an `@Target` meta-annotation is not present on an annotation type *`T`*, then an annotation of type *`T`* may be written as a modifier for any declaration except a type parameter declaration.

If an `@Target` meta-annotation is present, the compiler will enforce the usage restrictions indicated by `java.lang.annotation.ElementType` enum constants, in line with JLS 9.7.4.

9.6.3.b javadoc for `java.lang.annotation.ElementType`

The constants of this enumerated type provide a simple classification of the syntactic locations where annotations may appear in a Java program. These constants are used in `@Target` meta-annotations to specify where it is legal to write annotations of a given type.

The syntactic locations where annotations may appear are split into *declaration contexts*, where annotations apply to declarations, and *type contexts*, where annotations apply to types used in declarations and expressions.

The constants `ANNOTATION_TYPE`, `CONSTRUCTOR`, `FIELD`, `LOCAL_VARIABLE`, `METHOD`, `PACKAGE`, `PARAMETER`, `TYPE`, and `TYPE_PARAMETER` correspond to the declaration contexts in JLS 9.6.3.1.

> For example, an annotation whose type is meta-annotated with `@Target(ElementType.FIELD)` may only be written as a modifier for a field declaration.

The constant `TYPE_USE` corresponds to the 15 type contexts in JLS 4.11, as well as to two declaration contexts: type declarations (including annotation type declarations) and type parameter declarations.

> For convenience, an annotation whose type is meta-annotated with `@Target(ElementType.TYPE_USE)` is permitted to be written on a type declaration or type parameter declaration as shorthand for writing it at all uses of the declared entity. For example, if the annotation type `Interned` is meta-annotated in this way, then `@Interned class C {...}` could indicate that all uses of `C` are "interned", even though for other classes some instances may be interned and other instances not interned. The meta-annotation `@Target(ElementType.TYPE)` would be insufficient for these latter cases involving other classes.

> For example, an annotation whose type is meta-annotated with `@Target(ElementType.TYPE_USE)` may be written on the type of a field (or within the type of the field, if it is a nested, parameterized, or array type), and may also appear as a modifier for, say, a class declaration.

1.7 Type Annotations

9.7 Annotations

An *annotation* is a marker which associates information with a program construct, but has no effect at run time. An annotation denotes a specific invocation of an annotation type (9.6) and often provides values for the elements of that type.

There are three kinds of annotations. The first kind is the most general, while the other kinds are merely shorthands for the first kind.

Annotation:
  NormalAnnotation
  MarkerAnnotation
  SingleElementAnnotation

Normal annotations are described in 9.7.1, marker annotations in 9.7.2, and single element annotations in 9.7.3. Annotations may appear at various syntactic locations in a program, as described in 9.7.4. The number of annotations of the same type that may appear at a location is determined by their type, as described in 9.7.5.

9.7.5 is not part of this document, but may be found in the specification for Repeating Annotations at `http://cr.openjdk.java.net/~abuckley/8misc.pdf`

9.7.4 Where Annotations May Appear

A *declaration annotation* is an annotation that applies to a declaration, and whose own type is applicable in the declaration context represented by that declaration.

A *type annotation* is an annotation that applies to a type (or any part of a type), and whose own type is applicable in type contexts.

For example, given the field declaration:

```
@Foo int f;
```

`@Foo` is a declaration annotation on `f` if `Foo` is meta-annotated by `@Target(ElementType.FIELD)`, and a type annotation on `int` if `Foo` is meta-annotated by `@Target(ElementType.TYPE_USE)`. It is possible for `@Foo` to be both a declaration annotation and a type annotation simultaneously.

Type annotations can apply to an array type or any component type thereof. For example, assuming that `A`, `B`, and `C` are annotation types meta-annotated with `@Target(ElementType.TYPE_USE)`, then given the field declaration:

```
@C int @A [] @B [] f;
```

`@A` applies to the array type `int[][]`, `@B` applies to its component type `int[]`, and `@C` applies to the element type `int`.

An important property of this syntax is that, in two declarations that differ only in the number of array levels, the annotations to the left of the type refer to the same type. For example, `@C` applies to the type `int` in all of the following declarations:

```
@C int f;
@C int[] f;
@C int[][] f;
```

It is possible for an annotation to appear at a syntactic location in a program where it could plausibly apply to a declaration, or a type, or both. This can happen in any of the five declaration contexts where modifiers immediately precede the type:

- Method declarations (including elements of annotation types)
- Constructor declarations
- Field declarations (including enum constants)
- Formal and exception parameter declarations
- Local variable declarations (including loop variables of `for` statements and resource variables of `try`-with-resources statements)

The grammar of the Java programming language unambiguously treats annotations at these locations as modifiers for a declaration (JLS 8.3), but that is purely a syntactic matter. Whether an annotation applies to a declaration, or the type of the declared entity, or both, depends on the applicability of the annotation's type:

- If the annotation's type is applicable in the declaration context corresponding to the declaration, and not in type contexts, then the annotation is deemed to apply only to the declaration.
- If the annotation's type is applicable in type contexts, and not in the declaration context corresponding to the declaration, then the annotation is deemed to apply only to the type which is closest to the annotation.
- If the annotation's type is applicable in the declaration context corresponding to the declaration *and* in type contexts, then the annotation is deemed to apply to both the declaration *and* the type which is closest to the annotation.

There are two special cases involving method/constructor declarations:

- If an annotation appears before a constructor declaration and is deemed to apply to the type which is closest to the annotation, that type is the type is the newly constructed object. The type of the newly constructed object is the fully qualified name of the type immediately enclosing the constructor declaration. Within that fully qualified name, the annotation applies to the simple type name indicated by the constructor declaration.
- If an annotation appears before a `void` method declaration and is deemed to apply only to the type which is closest to the annotation, a compile-time error occurs.

It is a compile-time error if an annotation of type *`T`* is syntactically a modifier for:

- a package declaration, but *`T`* is not applicable to package declarations.

- a class, interface, or enum declaration, but `T` is not applicable to type declarations or type contexts; or an annotation type declaration, but `T` is not applicable to annotation type declarations or type declarations or type contexts.
- a method declaration (including an element of an annotation type), but `T` is not applicable to method declarations or type contexts.
- a constructor declaration, but `T` is not applicable to constructor declarations or type contexts.
- a type parameter declaration of a generic class, interface, method, or constructor, but `T` is not applicable to type parameter declarations or type contexts.
- a field declaration (including an enum constant), but `T` is not applicable to field declarations or type contexts.
- a formal or exception parameter declaration, but `T` is not applicable to either formal and exception parameter declarations or type contexts.
- a receiver parameter, but `T` is not applicable to type contexts.
- a local variable declaration (including a loop variable of a `for` statement or a resource variable of a `try`-with-resources statement), but `T` is not applicable to local variable declarations or type contexts.

Note that most of the clauses above mention "... or type contexts", because even if an annotation does not apply to the declaration, it may still apply to the type of the declared entity.

A reference to these rules will replace, in every section concerned with a declaration context, the clumsy JLS3 rule of the form: "If an annotation 'a' on a ZZZ declaration corresponds to an annotation type T, and T has a (meta-)annotation 'm' that corresponds to `java.lang.annotation.Target`, then 'm' must have an element whose value is `java.lang.annotation.ElementType`.ZZZ, or a compile-time error occurs."

A type annotation is *admissible* if both of the following hold:

- The simple name to which the annotation is closest is classified as a *TypeName*, not a *PackageName*.
- If the simple name to which the annotation is closest is followed by `"."` and another *TypeName*, *i.e.*, the annotation appears as `@X T.U`, then `U` denotes an inner class of `T`.

It is a compile-time error if an annotation of type `T` applies to the outermost level of a type in a type context, and `T` is not applicable in type contexts or the declaration context (if any) which occupies the same syntactic location.

It is a compile-time error if an annotation of type `T` applies to a part of a type (that is, not the outermost level) in a type context, and `T` is not applicable in type contexts.

It is a compile-time error if an annotation of type `T` applies to a type (or any part of a type) in a type context, and `T` is applicable in type contexts, and the annotation is not admissible.

For example, assume an annotation type `TA` which is meta-annotated with just `@Target(ElementType.TYPE_USE)`. The terms `@TA java.lang.Object` and `java.@TA lang.Object` are illegal because the simple name to which `@TA` is closest is classified as a package name. On the other hand, `java.lang.@TA Object` is legal.

Note that the illegal terms are illegal "everywhere". The ban on annotating package names applies broadly: to locations which are solely type contexts, such as `class ... extends @TA java.lang.Object {...}`, and to locations which are both declaration and type contexts, such as `@TA java.lang.Object f;`. (There are no locations which are solely declaration contexts where a package name could be annotated, as class, package, and type parameter declarations use only simple names.)

If `TA` is additionally meta-annotated with `@Target(ElementType.FIELD)`, then the term `@TA java.lang.Object` is legal in locations which are both declaration and type contexts, such as a field declaration `@TA java.lang.Object f;`. Here, `@TA` is deemed to apply to the declaration of `f` (and not to the type `java.lang.Object`) because `TA` is applicable in the field declaration context.

CHAPTER 2

The Java Virtual Machine Specification

IN addition to the new sections below, minor changes are made to terminology in 4.7.16 through 4.7.19. First, in `RuntimeVisibleAnnotations` and `RuntimeInvisibleAnnotations`, the phrase "program construct" is replaced by "declaration". Second, in `RuntimeVisibleParameterAnnotations` and `RuntimeInvisibleParameterAnnotations`, the phrase "annotations on the formal parameters" is replaced by "annotations on the declarations of formal parameters", and the ordering of entries in the `parameter_annotations` table is described more simply.

To make room for the two new sections below, the sections for the `AnnotationDefault` and `BootstrapMethods` attributes are renumbered to 4.7.22 and 4.7.23 respectively.

4.7.20 The `RuntimeVisibleTypeAnnotations` Attribute

The `RuntimeVisibleTypeAnnotations` attribute is an variable-length attribute in the `attributes` table of a `ClassFile`, `field_info`, or `method_info` structure, or `Code` attribute (4.1, 4.5, 4.6, 4.7.3). The `RuntimeVisibleTypeAnnotations` attribute records run-time visible annotations on types used in the declaration of the corresponding class, field, or method, or in an expression in the corresponding method body. The `RuntimeVisibleTypeAnnotations` attribute also records run-time visible annotations on type parameter declarations of generic classes and interfaces. The Java Virtual Machine must make these annotations available so they can be returned by the appropriate reflective APIs.

There may be at most one `RuntimeVisibleTypeAnnotations` attribute in the `attributes` table of a `ClassFile`, `field_info`, or `method_info` structure, or `Code` attribute.

An `attributes` table contains a `RuntimeVisibleTypeAnnotations` attribute only if types are annotated in kinds of declaration or expression that correspond to the parent structure or attribute of the `attributes` table.

> For example, all annotations on types in the `implements` clause of a class declaration are recorded in the `RuntimeVisibleTypeAnnotations` attribute of the class's `ClassFile` structure. Meanwhile, all annotations on the type in a field declaration are recorded in the `RuntimeVisibleTypeAnnotations` attribute of the field's `field_info` structure.

```
RuntimeVisibleTypeAnnotations_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              num_annotations;
    type_annotation annotations[num_annotations];
}
```

The items of the `RuntimeVisibleTypeAnnotations_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 "`RuntimeVisibleTypeAnnotations`".

`attribute_length`

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

`num_annotations`

The value of the `num_annotations` item gives the number of run-time visible type annotations represented by the structure.

`annotations`

Each entry in the `annotations` table represents a single run-time visible annotation on a type used in a declaration or expression. The `type_annotation` structure has the following format:

```
type_annotation {
    u1 target_type;
    union {
        type_parameter_target;
        supertype_target;
        type_parameter_bound_target;
        empty_target;
        method_formal_parameter_target;
        throws_target;
        localvar_target;
        catch_target;
        offset_target;
        type_argument_target;
    } target_info;
    type_path target_path;
    u2        type_index;
    u2        num_element_value_pairs;
    {   u2            element_name_index;
        element_value value;
    } element_value_pairs[num_element_value_pairs];
}
```

The first three items - `target_type`, `target_info`, and `target_path` - specify the precise location of the annotated type. The last three items - `type_index`, `num_element_value_pairs`, and `element_value_pairs` - specify the annotation's own type and element-value pairs.

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

`target_type`

The value of the `target_type` item denotes the kind of target on which the annotation appears. The various kinds of target correspond to the contexts in the Java programming language where types are used in declarations and expressions (JLS 4.11).

The legal values of `target_type` are specified in Tables 4.11 and 4.12. Each value is a one-byte tag indicating which item of the `target_info` union follows the `target_type` item to give more information about the target.

The kinds of target in Tables 4.11 and 4.12 correspond to the type contexts in JLS 4.11. Namely, `target_type` values 0x10 to 0x17 correspond to type contexts 1-10, while `target_type` values 0x40 to 0x4B correspond to type contexts 11-17.

The value of the `target_type` item determines whether the `type_annotation` structure appears in a `RuntimeVisibleTypeAnnotations` attribute in a `ClassFile` structure, a `field_info` structure, a `method_info` structure, or a `Code` attribute. Table

4.13 gives the location of the `RuntimeVisibleTypeAnnotations` attribute for a `type_annotation` structure with each legal `target_type` value.

`target_info`

The value of the `target_info` item denotes precisely which type in a declaration or expression is annotated.

The items of the `target_info` union are specified in 4.7.20.1.

`target_path`

The value of the `target_path` item denotes precisely which part of the type indicated by `target_info` is annotated.

The format of the `type_path` structure is specified in 4.7.20.2.

`type_index, num_element_value_pairs, element_value_pairs`

The meaning of these items in the `type_annotation` structure is the same as their meaning in the `annotation` structure (4.7.16).

Table 4.11. Interpretation of `target_type` values (Part 1)

Value	**Kind of target**	`target_info` **item**
0x00	type parameter declaration of generic class or interface	`type_parameter_target`
0x01	type parameter declaration of generic method or constructor	`type_parameter_target`
0x10	type in `extends` clause of class or interface declaration (including the direct superclass of an anonymous class declaration), or in `implements` clause of interface declaration	`supertype_target`
0x11	type in bound of type parameter declaration of generic class or interface	`type_parameter_bound_target`
0x12	type in bound of type parameter declaration of generic method or constructor	`type_parameter_bound_target`
0x13	type in field declaration	`empty_target`
0x14	return type of method, or type of newly constructed object	`empty_target`
0x15	receiver type of method or constructor	`empty_target`
0x16	type in formal parameter declaration of method or constructor	`formal_parameter_target`
0x17	type in `throws` clause of method or constructor	`throws_target`

Table 4.12. Interpretation of `target_type` values (Part 2)

Value	Kind of target	`target_info` item
0x40	type in local variable declaration	`localvar_target`
0x41	type in resource variable declaration	`localvar_target`
0x42	type in exception parameter declaration	`catch_target`
0x43	type in `instanceof` expression	`offset_target`
0x44	type in `new` expression	`offset_target`
0x45	type in constructor reference expression	`offset_target`
0x46	type in method reference expression	`offset_target`
0x47	type in cast expression	`type_argument_target`
0x48	type argument for generic constructor in `new` expression or explicit constructor invocation statement	`type_argument_target`
0x49	type argument for generic method in method invocation expression	`type_argument_target`
0x4A	type argument for generic constructor in constructor reference expression	`type_argument_target`
0x4B	type argument for generic method in method reference expression	`type_argument_target`

Table 4.13. Location of enclosing attribute for `target_type` values

Value	Kind of target	Location
0x00	type parameter declaration of generic class or interface	`ClassFile`
0x01	type parameter declaration of generic method or constructor	`method_info`
0x10	type in `extends` clause of class or interface declaration, or in `implements` clause of interface declaration	`ClassFile`
0x11	type in bound of type parameter declaration of generic class or interface	`ClassFile`
0x12	type in bound of type parameter declaration of generic method or constructor	`method_info`
0x13	type in field declaration	`field_info`
0x14	return type of method or constructor	`method_info`
0x15	receiver type of method or constructor	`method_info`
0x16	type in formal parameter declaration of method or constructor	`method_info`
0x17	type in `throws` clause of method or constructor	`method_info`
0x40-0x4B	types in local variable declarations, resource variable declarations, exception parameter declarations, expressions	`Code`

4.7.20.1 The `target_info` union

The items of the `target_info` union (except for the first) specify precisely which type in a declaration or expression is annotated. The first item specifies not which type, but rather which declaration of a type parameter is annotated. The items are as follows:

- The `type_parameter_target` item indicates that an annotation appears on the declaration of the *i*'th type parameter of a generic class, generic interface, generic method, or generic constructor.

  ```
  type_parameter_target {
      u1 type_parameter_index;
  }
  ```

  The value of the `type_parameter_index` item specifies which type parameter declaration is annotated. A `type_parameter_index` value of `0` specifies the first type parameter declaration.

- The `supertype_target` item indicates that an annotation appears on a type in the `extends` or `implements` clause of a class or interface declaration.

  ```
  supertype_target {
      u2 supertype_index;
  }
  ```

  A `supertype_index` value of 65535 specifies that the annotation appears on the superclass in an `extends` clause of a class declaration.

  Any other `supertype_index` value is an index into the `interfaces` array of the enclosing `ClassFile` structure, and specifies that the annotation appears on that superinterface in either the `implements` clause of a class declaration or the `extends` clause of an interface declaration.

- The `type_parameter_bound_target` item indicates that an annotation appears on the *i*'th bound of the *j*'th type parameter declaration of a generic class, interface, method, or constructor.

  ```
  type_parameter_bound_target {
      u1 type_parameter_index;
      u1 bound_index;
  }
  ```

  The value of the of `type_parameter_index` item specifies which type parameter declaration has an annotated bound. A `type_parameter_index` value of `0` specifies the first type parameter declaration.

  The value of the `bound_index` item specifies which bound of the type parameter declaration indicated by `type_parameter_index` is annotated. A `bound_index` value of `0` specifies the first bound of a type parameter declaration.

  The `type_parameter_bound_target` item records that a bound is annotated, but does not record the type which constitutes the bound. The type may be found by inspecting the class signature or method signature stored in the appropriate `Signature` attribute.

- The `empty_target` item indicates that an annotation appears on either the type in a field declaration, the return type of a method, the type of a newly constructed object, or the receiver type of a method or constructor.

  ```
  empty_target {
  }
  ```

  Only one type appears in each of these locations, so there is no per-type information to represent in the `target_info` union.

- The `formal_parameter_target` item indicates that an annotation appears on the type in a formal parameter declaration of a method or constructor.

```
formal_parameter_target {
    u1 formal_parameter_index;
}
```

The value of the formal_parameter_index item specifies which formal parameter declaration has an annotated type. A formal_parameter_index value of 0 specifies the first formal parameter declaration.

> The formal_parameter_target item records that a formal parameter's type is annotated, but does not record the type itself. The type may be found by inspecting the method descriptor (4.3.3) of the method_info structure enclosing the RuntimeVisibleTypeAnnotations attribute. A formal_parameter_index value of 0 indicates the first parameter descriptor in the method descriptor.

- The throws_target item indicates that an annotation appears on the *i*'th type in the throws clause of a method or constructor declaration.

```
throws_target {
    u2 throws_type_index;
}
```

The value of the throws_type_index item is an index into the exception_index_table array of the Exceptions attribute of the method_info structure enclosing the RuntimeVisibleTypeAnnotations attribute.

- The localvar_target item indicates that an annotation appears on the type in a local variable declaration, including a variable declared as a resource in a try-with-resources statement.

```
localvar_target {
    u2 table_length;
    {   u2 start_pc;
        u2 length;
        u2 index;
    } table[table_length];
}
```

The value of the table_length item gives the number of entries in the table array. Each entry indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry contains the following three items:

start_pc, length

The given local variable has a value at indices into the code array in the interval [start_pc, start_pc + length), that is, between start_pc inclusive and start_pc + length exclusive.

`index`

The given local variable must be at `index` in the local variable array of the current frame.

If the local variable at `index` is of type `double` or `long`, it occupies both `index` and `index + 1`.

A table is needed to fully specify the local variable whose type is annotated, because a single local variable may be represented with different local variable indices over multiple live ranges. The `start_pc`, `length`, and `index` items in each table entry specify the same information as a `LocalVariableTable` attribute.

The `localvar_target` item records that a local variable's type is annotated, but does not record the type itself. The type may be found by inspecting the appropriate `LocalVariableTable` attribute.

- The `catch_target` item indicates that an annotation appears on the *i*'th type in an exception parameter declaration.

```
catch_target {
    u2 exception_table_index;
}
```

The value of the `exception_table_index` item is an index into the `exception_table` array of the `Code` attribute enclosing the `RuntimeVisibleTypeAnnotations` attribute.

The possibility of more than one type in an exception parameter declaration arises from the multi-`catch` clause of the `try` statement, where the type of the exception parameter is a union of types (JLS 14.20). A compiler usually creates one `exception_table` entry for each type in the union, which allows the `catch_target` item to distinguish them. This preserves the correspondence between a type and its annotations.

- The `offset_target` item indicates that an annotation appears on either the type in an `instanceof` expression or a `new` expression, or the type before the `::` in a method or constructor reference expression.

```
offset_target {
    u2 offset;
}
```

The value of the `offset` item specifies the `code` array offset of either the `instanceof` bytecode instruction corresponding to the `instanceof` expression, the `new` bytecode instruction corresponding to the `new` expression, or the bytecode instruction corresponding to the method or constructor reference expression.

- The `type_argument_target` item indicates that an annotation appears either on the *i*'th type in a cast expression, or on the *i*'th type argument in the explicit type argument list for any of the following: a `new` expression, an explicit constructor invocation statement, a method invocation expression, or a method or constructor reference expression.

```
type_argument_target {
    u2 offset;
    u1 type_argument_index;
}
```

The value of the `offset` item specifies the `code` array offset of either the bytecode instruction corresponding to the cast expression, the `new` bytecode instruction corresponding to the `new` expression, the bytecode instruction corresponding to the explicit constructor invocation statement, the bytecode instruction corresponding to the method invocation expression, or the bytecode instruction corresponding to the method or constructor reference expression.

For a cast expression, the value of the `type_argument_index` item specifies which type in the cast operator is annotated. A `type_argument_index` value of `0` specifies the first (or only) type in the cast operator.

> The possibility of more than one type in a cast expression arises from a cast to an intersection type.

For an explicit type argument list, the value of the `type_argument_index` item specifies which type argument is annotated. A `type_argument_index` value of `0` specifies the first type argument.

4.7.20.2 The `type_path` structure

Wherever a type is used in a declaration or expression, the `type_path` structure identifies which part of the type is annotated. An annotation may appear on the type itself, but if the type is a reference type, then there are additional locations where an annotation may appear:

- If an array type *`T[]`* is used in a declaration or expression, then an annotation may appear on any component type of the array type, including the element type.
- If a nested type *`T1.T2`* is used in a declaration or expression, then an annotation may appear on the name of the top level type or any member type.
- If a parameterized type *`T<A>`* or *`T<? extends A>`* or *`T<? super A>`* is used in a declaration or expression, then an annotation may appear on any type argument or on the bound of any wildcard type argument.

> For example, consider the different parts of `String[][]` that are annotated in:

```
@X String [] []
String @X [] []
String [] @X []
```

or the different parts of the nested type `Outer.Middle.Inner` that are annotated in:

```
@X Outer.Middle.Inner
Outer.@X Middle.Inner
Outer.Middle.@X Inner
```

or the different parts of the parameterized types `Map<String,Object>` and `List<...>` that are annotated in:

```
@X Map<String,Object>
Map<@X String,Object>
Map<String,@X Object>

List<@X ? extends String>
List<? extends @X String>
```

The `type_path` structure has the following format:

```
type_path {
    u1 path_length;
    {   u1 type_path_kind;
        u1 type_argument_index;
    } path[path_length];
}
```

The value of the `path_length` item gives the number of entries in the `path` array. If the value of `path_length` is `0`, then the annotation appears directly on the type itself. If the value of `path_length` is non-zero, then each entry in the `path` array represents an iterative, left-to-right step towards the precise location of the annotation in an array type, nested type, or parameterized type. (In an array type, the iteration visits the array type itself, then its component type, then the component type of that component type, and so on, until the element type is reached.) Each entry contains two items:

`type_path_kind`

The legal values for the `type_path_kind` item are listed in Table 4.14.

Table 4.14. Interpretation of `type_path_kind` values

Value	Interpretation
`0`	Annotation is deeper in an array type
`1`	Annotation is deeper in a nested type
`2`	Annotation is on the bound of a wildcard type argument of a parameterized type
`3`	Annotation is on a type argument of a parameterized type

`type_argument_index`

If the value of the `type_path_kind` item is `0`, `1`, or `2`, then the value of the `type_argument_index` item is `0`.

If the value of the `type_path_kind` item is `3`, then the value of the `type_argument_index` item specifies which type argument of a parameterized type is annotated. A `type_argument_index` value of `0` specifies the first type argument.

Table 2.5. `type_path` **structures for** `@A Map<@B ? extends @C String, @D List<@E Object>>`

Annotation	`path_length`	`path`
`@A`	`0`	`[]`
`@B`	`1`	`[{type_path_kind: 3; type_argument_index: 0}]`
`@C`	`2`	`[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 2; type_argument_index: 0}]`
`@D`	`1`	`[{type_path_kind: 3; type_argument_index: 1}]`
`@E`	`2`	`[{type_path_kind: 3; type_argument_index: 1}, {type_path_kind: 3; type_argument_index: 0}]`

Table 2.6. `type_path` **structures for** `@I String @F [] @G [] @H []`

Annotation	`path_length`	`path`
`@F`	`0`	`[]`
`@G`	`1`	`[{type_path_kind: 0; type_argument_index: 0}]`
`@H`	`2`	`[{type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]`
`@I`	`3`	`[{type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]`

Table 2.7. `type_path` **structures for** `@A List<@B Comparable<@F Object @C [] @D [] @E []>>`

Annotation	path_length	path
@A	0	[]
@B	1	[{type_path_kind: 3; type_argument_index: 0}]
@C	2	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}]
@D	3	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]
@E	4	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]
@F	5	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]

Table 2.8. `type_path` **structures for** `@C Outer . @B Middle . @A Inner`

Annotation	path_length	path
@A	2	[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}]
@B	1	[{type_path_kind: 1; type_argument_index: 0}]
@C	0	[]

Table 2.9. `type_path` **structures for** `Outer . Middle<@D Foo . @C Bar> . Inner<@B String @A []>`

Annotation	`path_length`	`path`
`@A`	`3`	`[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}]`
`@B`	`4`	`[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]`
`@C`	`3`	`[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}]`
`@D`	`2`	`[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}]`

4.7.21 The `RuntimeInvisibleTypeAnnotations` Attribute

The `RuntimeInvisibleTypeAnnotations` attribute is an variable-length attribute in the `attributes` table of a `ClassFile`, `field_info`, or `method_info` structure, or `Code` attribute (4.1, 4.5, 4.6, 4.7.3). The `RuntimeInvisibleTypeAnnotations` attribute records run-time invisible annotations on types used in the corresponding declaration of a class, field, or method, or in an expression in the corresponding method body. The `RuntimeInvisibleTypeAnnotations` attribute also records annotations on type parameter declarations of generic classes and interfaces.

There may be at most one `RuntimeInvisibleTypeAnnotations` attribute in the `attributes` table of a `ClassFile`, `field_info`, or `method_info` structure, or `Code` attribute.

An `attributes` table contains a `RuntimeInvisibleTypeAnnotations` attribute only if types are annotated in kinds of declaration or expression that correspond to the parent structure or attribute of the `attributes` table.

```
RuntimeInvisibleTypeAnnotations_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              num_annotations;
    type_annotation annotations[num_annotations];
}
```

The items of the `RuntimeInvisibleTypeAnnotations_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 "`RuntimeInvisibleTypeAnnotations`".

`attribute_length`

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

`num_annotations`

The value of the `num_annotations` item gives the number of run-time invisible type annotations represented by the structure.

`annotations`

Each entry in the `annotations` table represents a single run-time invisible annotation on a type used in a declaration or expression. The `type_annotation` structure is specified in 4.7.20.

CHAPTER 3

Reflection APIs

SUPPORT for type annotations in the Core Reflection API (`java.lang.reflect`) and the Language Model API (`javax.lang.model`) was introduced at `http://mail.openjdk.java.net/pipermail/type-annotations-spec-experts/2012-September/000001.html`.

The Language Model API was subsequently streamlined, as discussed at `http://mail.openjdk.java.net/pipermail/type-annotations-spec-experts/2013-February/000064.html`.

The new artifacts in these APIs are listed in the following sections. Diffs against Java SE 7 are available, under an evaluation and comment license, at `http://cr.openjdk.java.net/~jfranck/anno-work/spec/`.

Finally, the Annotation Processing API (`javax.annotation.processing`) underwent minor changes to recognize the existence of type annotations in the Java programming language. No types or methods were added, but the behavior of `Processor` was modified, as discussed at `http://mail.openjdk.java.net/pipermail/type-annotations-spec-comments/2013-March/000030.html`.

3.1 Core Reflection API

New methods in `java.lang`:

- `Class.getAnnotatedSuperclass()`
- `Class.getAnnotatedInterfaces()`

New methods in `java.lang.reflect`: (`Executable`, the superclass of `Constructor` and `Method`, was introduced in Java SE 8 independently of JSR 308, while `Parameter` was added by JEP 118)

- `Executable.getAnnotatedExceptionTypes()`

- `Executable.getAnnotatedParameterTypes()`
- `Executable.getAnnotatedReceiverType()`
- `Executable.getAnnotatedReturnType()`
- `Field.getAnnotatedType()`
- `Parameter.getAnnotatedType()`
- `TypeVariable.getAnnotatedBounds()`

New types in `java.lang.reflect`:

- `AnnotatedType`
- `AnnotatedArrayType` [extends `AnnotatedType`]
- `AnnotatedParameterizedType` [extends `AnnotatedType`]
- `AnnotatedTypeVariable` [extends `AnnotatedType`]
- `AnnotatedWildcardType` [extends `AnnotatedType`]

3.2 Language Model API

New type in `javax.lang.model`:

- `AnnotatedConstruct` [extended by `javax.lang.model.element.Element` and `javax.lang.model.type.TypeMirror`]

New methods in `javax.lang.model.element`:

- `ExecutableElement.getReceiverType()`
- `TypeParameterElement.getAnnotationMirrors()` [implementation of existing method]

New method in `javax.lang.model.type`:

- `ExecutableType.getReceiverType()`