Annotations

Annotations

```
Annotation       ::=  '@' SimpleType {ArgumentExprs}
ConstrAnnotation ::=  '@' SimpleType ArgumentExprs
```

Definition

Annotations associate meta-information with definitions. A simple annotation has the form `@$c$` or `@$c(a_1 , \ldots , a_n)$`. Here, c is a constructor of a class C, which must conform to the class `scala.Annotation`.

Annotations may apply to definitions or declarations, types, or expressions. An annotation of a definition or declaration appears in front of that definition. An annotation of a type appears after that type. An annotation of an expression e appears after the expression e, separated by a colon. More than one annotation clause may apply to an entity. The order in which these annotations are given does not matter.

Examples:

```
@deprecated("Use D", "1.0") class C { ... } // Class annotation
@transient @volatile var m: Int             // Variable annotation
String @local                               // Type annotation
(e: @unchecked) match { ... }               // Expression annotation
```

Predefined Annotations

Java Platform Annotations

The meaning of annotation clauses is implementation-dependent. On the Java platform, the following annotations have a standard meaning.

- `@transient` Marks a field to be non-persistent; this is equivalent to the `transient` modifier in Java.
- `@volatile` Marks a field which can change its value outside the control of the program; this is equivalent to the `volatile` modifier in Java.

- `@SerialVersionUID(<longlit>)` Attaches a serial version identifier (a `long` constant) to a class. This is equivalent to a the following field definition in Java:

  ```
  private final static SerialVersionUID = <longlit>
  ```

- `@throws(<classlit>)` A Java compiler checks that a program contains handlers for checked exceptions by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the `throws` clause for the method or constructor must mention the class of that exception or one of the superclasses of the class of that exception.

Java Beans Annotations

- `@scala.beans.BeanProperty` When prefixed to a definition of some variable `X`, this annotation causes getter and setter methods `getX`, `setX` in the Java bean style to be added in the class containing the variable. The first letter of the variable appears capitalized after the `get` or `set`. When the annotation is added to the definition of an immutable value definition `X`, only a getter is generated. The construction of these methods is part of code-generation; therefore, these methods become visible only once a classfile for the containing class is generated.
- `@scala.beans.BooleanBeanProperty` This annotation is equivalent to `scala.reflect.BeanProperty`, but the generated getter method is named `isX` instead of `getX`.

Deprecation Annotations

- `@deprecated(message: <stringlit>, since: <stringlit>)` Marks a definition as deprecated. Accesses to the defined entity will then cause a deprecated warning mentioning the *message* `<stringlit>` to be issued from the compiler. The argument *since* documents since when the definition should be considered deprecated. Deprecated warnings are suppressed in code that belongs itself to a definition that is labeled deprecated.
- `@deprecatedName(name: <stringlit>, since: <stringlit>)` Marks a formal parameter name as deprecated. Invocations of this entity using named parameter syntax referring to the deprecated parameter name cause a deprecation warning.

Scala Compiler Annotations

- `@unchecked` When applied to the selector of a `match` expression, this attribute suppresses any warnings about non-exhaustive pattern matches

which would otherwise be emitted. For instance, no warnings would be produced for the method definition below.

```
def f(x: Option[Int]) = (x: @unchecked) match {
case Some(y) => y
}
```

Without the `@unchecked` annotation, a Scala compiler could infer that the pattern match is non-exhaustive, and could produce a warning because `Option` is a `sealed` class.

- `@uncheckedStable` When applied a value declaration or definition, it allows the defined value to appear in a path, even if its type is volatile. For instance, the following member definitions are legal:

  ```
  type A { type T }
  type B
  @uncheckedStable val x: A with B // volatile type
  val y: x.T                       // OK since `x' is still a path
  ```

  Without the `@uncheckedStable` annotation, the designator `x` would not be a path since its type `A with B` is volatile. Hence, the reference `x.T` would be malformed.

  When applied to value declarations or definitions that have non-volatile types, the annotation has no effect.

- `@specialized` When applied to the definition of a type parameter, this annotation causes the compiler to generate specialized definitions for primitive types. An optional list of primitive types may be given, in which case specialization takes into account only those types. For instance, the following code would generate specialized traits for `Unit`, `Int` and `Double`

  ```
  trait Function0[@specialized(Unit, Int, Double) T] {
    def apply: T
  }
  ```

  Whenever the static type of an expression matches a specialized variant of a definition, the compiler will instead use the specialized version. See the specialization sid for more details of the implementation.

User-defined Annotations

Other annotations may be interpreted by platform- or application-dependent tools. The class `scala.annotation.Annotation` is the base class for user-defined annotations. It has two sub-traits: - `scala.annotation.StaticAnnotation`: Instances of a subclass of this trait will be stored in the generated class files, and therefore accessible to runtime reflection and later compilation runs. - `scala.annotation.ConstantAnnotation`: Instances of a subclass of this trait may only have arguments which are constant expressions, and are also stored

in the generated class files. - If an annotation class inherits from neither `scala.ConstantAnnotation` nor `scala.StaticAnnotation`, its instances are visible only locally during the compilation run that analyzes them.

Host-platform Annotations

The host platform may define its own annotation format. These annotations do not extend any of the classes in the `scala.annotation` package, but can generally be used in the same way as Scala annotations. The host platform may impose additional restrictions on the expressions which are valid as annotation arguments.