

[Scala 3 Reference](#) / [Other New Features](#) / [The @targetName annotation](#)

LEARN

INSTALL

PLAYGROUND

FIND A LIBRARY

COMMUNITY

BLOG

# The @targetName annotation

[Edit this page on GitHub](#)

A `@targetName` annotation on a definition defines an alternate name for the implementation of that definition. Example:

```
import scala.annotation.targetName

object VecOps:
  extension [T](xs: Vec[T])
    @targetName("append")
    def += [T] (ys: Vec[T]): Vec[T] = ...
```

Here, the `+=` operation is implemented (in Byte code or native code) under the name `append`. The implementation name affects the code that is generated, and is the name under which code from other languages can call the method. For instance, `+=` could be invoked from Java like this:

```
VecOps.append(vec1, vec2)
```

The `@targetName` annotation has no bearing on Scala usages. Any application of that method in Scala has to use `+=`, not `append`.

## Details

1. `@targetName` is defined in package `scala.annotation`. It takes a single argument of type `String`. That string is called the *external name* of the definition that's annotated.
2. A `@targetName` annotation can be given for all kinds of definitions.
3. The name given in a `@targetName` annotation must be a legal name for the defined entities on the host platform.

4. It is recommended that definitions with symbolic names have a `@targetName` annotation. This will establish an alternate name that is easier to search for and will avoid cryptic encodings in runtime diagnostics.
5. Definitions with names in backticks that are not legal host platform names should also have a `@targetName` annotation.

## Relationship with Overriding

`@targetName` annotations are significant for matching two method definitions to decide whether they conflict or override each other. Two method definitions match if they have the same name, signature, and erased name. Here,

- The *signature* of a definition consists of the names of the erased types of all (value-) parameters and the method's result type.
- The *erased name* of a method definition is its target name if a `@targetName` annotation is given and its defined name otherwise.

This means that `@targetName` annotations can be used to disambiguate two method definitions that would otherwise clash. For instance.

```
def f(x: => String): Int = x.length
def f(x: => Int): Int = x + 1 // error: double definition
```

The two definitions above clash since their erased parameter types are both `Function0`, which is the type of the translation of a by-name-parameter. Hence they have the same names and signatures. But we can avoid the clash by adding a `@targetName` annotation to either method or to both of them. Example:

```
@targetName("f_string")
def f(x: => String): Int = x.length
def f(x: => Int): Int = x + 1 // OK
```

This will produce methods `f_string` and `f` in the generated code.

However, `@targetName` annotations are not allowed to break overriding relationships between two definitions that have otherwise the same names and types. So the following would be in error:

```
import annotation.targetName
class A:
  def f(): Int = 1
```

```
class B extends A:
  @targetName("g") def f(): Int = 2
```

The compiler reports here:

```
-- Error: test.scala:6:23 -----
6 |   @targetName("g") def f(): Int = 2
  |                        ^
  |error overriding method f in class A of type (): Int;
  |method f of type (): Int should not have a @targetName
  |annotation since the overridden member hasn't one either
```

The relevant overriding rules can be summarized as follows:


- Two members can override each other if their names and signatures are the same, and they either have the same erased names or the same types.
- If two members override, then both their erased names and their types must be the same.

As usual, any overriding relationship in the generated code must also be present in the original code. So the following example would also be in error:

```
import annotation.targetName
class A:
  def f(): Int = 1
class B extends A:
  @targetName("f") def g(): Int = 2
```

Here, the original methods `g` and `f` do not override each other since they have different names. But once we switch to target names, there is a clash that is reported by the compiler:

```
-- [E120] Naming Error: test.scala:4:6 -----
4 | class B extends A:
  |      ^
  |      Name clash between defined and inherited member:
  |      def f(): Int in class A at line 3 and
  |      def g(): Int in class B at line 5
  |      have the same name and type after erasure.
1 | error found
```

 Scaladoc

Copyright (c) 2002-2022, LAMP/EPFL

