

[Scala 3 Reference](#) / [Other New Features](#) / [Optional Braces](#)**LEARN**

INSTALL

PLAYGROUND

FIND A LIBRARY

COMMUNITY

BLOG

# Optional Braces

[✎ Edit this page on GitHub](#)

Scala 3 enforces some rules on indentation and allows some occurrences of braces `{ ... }` to be optional:

- First, some badly indented programs are flagged with warnings.
- Second, some occurrences of braces `{ ... }` are made optional. Generally, the rule is that adding a pair of optional braces will not change the meaning of a well-indented program.

These changes can be turned off with the compiler flag `-no-indent`.

## Indentation Rules

The compiler enforces two rules for well-indented programs, flagging violations as warnings.

1. In a brace-delimited region, no statement is allowed to start to the left of the first statement after the opening brace that starts a new line.

This rule is helpful for finding missing closing braces. It prevents errors like:

```
if (x < 0) {  
  println(1)  
  println(2)  
  
println("done") // error: indented too far to the left
```

2. If significant indentation is turned off (i.e. under Scala 2 mode or under `-no-indent`) and we are at the start of an indented sub-part of an expression, and the indented part ends in a newline, the next statement must start at an indentation width less than the sub-part. This prevents errors where an opening brace was forgotten, as in

```
if (x < 0)
  println(1)
  println(2) // error: missing `{`
```

These rules still leave a lot of leeway how programs should be indented. For instance, they do not impose any restrictions on indentation within expressions, nor do they require that all statements of an indentation block line up exactly.

The rules are generally helpful in pinpointing the root cause of errors related to missing opening or closing braces. These errors are often quite hard to diagnose, in particular in large programs.

## Optional Braces

The compiler will insert `<indent>` or `<outdent>` tokens at certain line breaks. Grammatically, pairs of `<indent>` and `<outdent>` tokens have the same effect as pairs of braces `{` and `}`.

The algorithm makes use of a stack `IW` of previously encountered indentation widths. The stack initially holds a single element with a zero indentation width. The *current indentation width* is the indentation width of the top of the stack.

There are two rules:

1. An `<indent>` is inserted at a line break, if
  - An indentation region can start at the current position in the source, and
  - the first token on the next line has an indentation width strictly greater than the current indentation width

An indentation region can start

- after the leading parameters of an `extension`, or
- after a `with` in a given instance, or
- after a `:` at end of line" token (see below)
- after one of the following tokens:

```
= => ?=> <- catch do else finally for
if match return then throw try while yield
```

- after the closing `)` of a condition in an old-style `if` or `while`.
- after the closing `)` or `}` of the enumerations of an old-style `for` loop

- after the closing `}` or `]` of the enumerations of an old style `for` loop without a `do`.



If an `<indent>` is inserted, the indentation width of the token on the next line is pushed onto `IW`, which makes it the new current indentation width.

2. An `<outdent>` is inserted at a line break, if

- the first token on the next line has an indentation width strictly less than the current indentation width, and
- the last token on the previous line is not one of the following tokens which indicate that the previous statement continues:

```
then else do catch finally yield match
```

- if the first token on the next line is a [leading infix operator](#). then its indentation width is less than the current indentation width, and it either matches a previous indentation width or is also less than the enclosing indentation width.

If an `<outdent>` is inserted, the top element is popped from `IW`. If the indentation width of the token on the next line is still less than the new current indentation width, step (2) repeats. Therefore, several `<outdent>` tokens may be inserted in a row.

The following two additional rules support parsing of legacy code with ad-hoc layout. They might be withdrawn in future language versions:

- An `<outdent>` is also inserted if the next token following a statement sequence starting with an `<indent>` closes an indentation region, i.e. is one of `then`, `else`, `do`, `catch`, `finally`, `yield`, `}`, `)`, `]` or `case`.

An `<outdent>` is finally inserted in front of a comma that follows a statement sequence starting with an `<indent>` if the indented region is itself enclosed in parentheses

It is an error if the indentation width of the token following an `<outdent>` does not match the indentation of some previous line in the enclosing indentation region. For instance, the following would be rejected.

```
if x < 0 then
  -x
else // error: `else` does not align correctly
  x
```

Indentation tokens are only inserted in regions where newline statement separators are also inferred: at the top-level, inside braces `{ ... }`, but not inside parentheses `( ... )`, patterns or types.

Note: The rules for leading infix operators above are there to make sure that

```
one
+ two.match
  case 1 => b
  case 2 => c
+ three
```

is parsed as `one + (two.match ... ) + three`. Also, that

```
if x then
  a
+ b
+ c
else d
```

is parsed as `if x then a + b + c else d`.

## Optional Braces Around Template Bodies

The Scala grammar uses the term *template body* for the definitions of a class, trait, or object that are normally enclosed in braces. The braces around a template body can also be omitted by means of the following rule.

If at the point where a template body can start there is a `:` that occurs at the end of a line, and that is followed by at least one indented statement, the recognized token is changed from ":" to ": at end of line". The latter token is one of the tokens that can start an indentation region. The Scala grammar is changed so an optional ": at end of line" is allowed in front of a template body.

Analogous rules apply for enum bodies and local packages containing nested definitions.

With these new rules, the following constructs are all valid:

```
trait A:
  def f: Int

class C(x: Int) extends A:
  def f = x
```

```
object O:
  def f = 3

enum Color:
  case Red, Green, Blue

new A:
  def f = 3

package p:
  def a = 1

package q:
  def b = 2
```

In each case, the `:` at the end of line can be replaced without change of meaning by a pair of braces that enclose the following indented definition(s).

The syntax changes allowing this are as follows:

```
Template      ::= InheritClauses [colonEol] [TemplateBody]
EnumDef       ::= id ClassConstr InheritClauses [colonEol] EnumBody
Packaging     ::= 'package' QualId [nl | colonEol] '{' TopStatSeq '}'
SimpleExpr    ::= 'new' ConstrApp {'with' ConstrApp} [[colonEol] TemplateBody]
```

Here, `colonEol` stands for "": at end of line", as described above. The lexical analyzer is modified so that a `:` at the end of a line is reported as `colonEol` if the parser is at a point where a `colonEol` is valid as next token.

## Spaces vs Tabs

Indentation prefixes can consist of spaces and/or tabs. Indentation widths are the indentation prefixes themselves, ordered by the string prefix relation. So, for instance "2 tabs, followed by 4 spaces" is strictly less than "2 tabs, followed by 5 spaces", but "2 tabs, followed by 4 spaces" is incomparable to "6 tabs" or to "4 spaces, followed by 2 tabs". It is an error if the indentation width of some line is incomparable with the indentation width of the region that's current at that point. To avoid such errors, it is a good idea not to mix spaces and tabs in the same source file.

## Indentation and Braces

Indentation can be mixed freely with braces `{ ... }`, as well as brackets `[ ... ]` and parentheses `( ... )`. For interpreting indentation inside such regions, the following rules apply.



1. The assumed indentation width of a multiline region enclosed in braces is the indentation width of the first token that starts a new line after the opening brace.
2. The assumed indentation width of a multiline region inside brackets or parentheses is:
  - if the opening bracket or parenthesis is at the end of a line, the indentation width of token following it,
  - otherwise, the indentation width of the enclosing region.
3. On encountering a closing brace `}`, bracket `]` or parenthesis `)`, as many `<outdent>` tokens as necessary are inserted to close all open nested indentation regions.

For instance, consider:

```
{  
  val x = f(x: Int, y =>  
    x * (  
      y + 1  
    ) +  
    (x +  
      x)  
  )  
}
```

- Here, the indentation width of the region enclosed by the braces is 3 (i.e. the indentation width of the statement starting with `val` ).
- The indentation width of the region in parentheses that follows `f` is also 3, since the opening parenthesis is not at the end of a line.
- The indentation width of the region in parentheses around `y + 1` is 9 (i.e. the indentation width of `y + 1` ).
- Finally, the indentation width of the last region in parentheses starting with `(x` is 6 (i.e. the indentation width of the indented region following the `=>` ).

## Special Treatment of Case Clauses

The indentation rules for `match` expressions and `catch` clauses are refined as follows:

- An indentation region is opened after a `match` or `catch` also if the following `case` appears at the indentation width that's current for the `match` itself.
- In that case, the indentation region closes at the first token at that same

in that case, the indentation region closes at the first token at that same indentation width that is not a `case`, or at any token with a smaller indentation width, whichever comes first.

The rules allow to write `match` expressions where cases are not indented themselves, as in the example below:

```
x match
case 1 => print("I")
case 2 => print("II")
case 3 => print("III")
case 4 => print("IV")
case 5 => print("V")

println(".")
```

## Using Indentation to Signal Statement Continuation

Indentation is used in some situations to decide whether to insert a virtual semicolon between two consecutive lines or to treat them as one statement. Virtual semicolon insertion is suppressed if the second line is indented more relative to the first one, and either the second line starts with "`(`", "`[`", or "`{`" or the first line ends with `return`. Examples:

```
f(x + 1)
  (2, 3)      // equivalent to `f(x + 1)(2, 3)`

g(x + 1)
(2, 3)      // equivalent to `g(x + 1); (2, 3)`

h(x + 1)
  {}        // equivalent to `h(x + 1){}`

i(x + 1)
  {}        // equivalent to `i(x + 1); {}`

if x < 0 then return
  a + b      // equivalent to `if x < 0 then return a + b`

if x < 0 then return
println(a + b) // equivalent to `if x < 0 then return; println(a + b)`
```

In Scala 2, a line starting with "`{`" always continues the function call on the preceding line, irrespective of indentation, whereas a virtual semicolon is inserted in all other cases. The Scala-2 behavior is retained under source `-no-indent` or `-source 3.0-`

migration .



## The End Marker

Indentation-based syntax has many advantages over other conventions. But one possible problem is that it makes it hard to discern when a large indentation region ends, since there is no specific token that delineates the end. Braces are not much better since a brace by itself also contains no information about what region is closed.

To solve this problem, Scala 3 offers an optional `end` marker. Example:

```
def largeMethod(...) =  
  ...  
  if ... then ...  
  else  
    ... // a large block  
  end if  
  ... // more code  
end largeMethod
```

An `end` marker consists of the identifier `end` and a follow-on specifier token that together constitute all the tokens of a line. Possible specifier tokens are identifiers or one of the following keywords

```
if   while   for   match   try   new   this   val   given
```

End markers are allowed in statement sequences. The specifier token `s` of an end marker must correspond to the statement that precedes it. This means:

- If the statement defines a member `x` then `s` must be the same identifier `x`.
- If the statement defines a constructor then `s` must be `this`.
- If the statement defines an anonymous given, then `s` must be `given`.
- If the statement defines an anonymous extension, then `s` must be `extension`.
- If the statement defines an anonymous class, then `s` must be `new`.
- If the statement is a `val` definition binding a pattern, then `s` must be `val`.
- If the statement is a package clause that refers to package `p`, then `s` must be the same identifier `p`.
- If the statement is an `if`, `while`, `for`, `try`, or `match` statement, then `s` must be that same token.

For instance, the following end markers are all legal:





```
package p1.p2:

  abstract class C():

    def this(x: Int) =
      this()
      if x > 0 then
        val a :: b =
          x :: Nil
        end val
        var y =
          x
        end y
        while y > 0 do
          println(y)
          y -= 1
        end while
        try
          x match
            case 0 => println("0")
            case _ =>
          end match
          finally
            println("done")
          end try
        end if
      end this

    def f: String
  end C

object C:
  given C =
    new C:
      def f = "!"
      end f
    end new
  end given
end C

extension (x: C)
  def ff: String = x.f ++ x.f
end extension

end p2
```

## When to Use End Markers

It is recommended that `end` markers are used for code where the extent of an

indentation region is not immediately apparent "at a glance". People will have



different preferences what this means, but one can nevertheless give some guidelines that stem from experience. An end marker makes sense if

- the construct contains blank lines, or
- the construct is long, say 15-20 lines or more,
- the construct ends heavily indented, say 4 indentation levels or more.

If none of these criteria apply, it's often better to not use an end marker since the code will be just as clear and more concise. If there are several ending regions that satisfy one of the criteria above, we usually need an end marker only for the outermost closed region. So cascades of end markers as in the example above are usually better avoided.

## Syntax

```
EndMarker      ::= 'end' EndMarkerTag    -- when followed by EOL
EndMarkerTag   ::= id | 'if' | 'while' | 'for' | 'match' | 'try'
                | 'new' | 'this' | 'given' | 'extension' | 'val'
BlockStat      ::= ... | EndMarker
TemplateStat    ::= ... | EndMarker
TopStat        ::= ... | EndMarker
```

## Example

Here is a (somewhat meta-circular) example of code using indentation. It provides a concrete representation of indentation widths as defined above together with efficient operations for constructing and comparing indentation widths.

```
enum IndentWidth:
  case Run(ch: Char, n: Int)
  case Conc(l: IndentWidth, r: Run)

def <= (that: IndentWidth): Boolean = this match
  case Run(ch1, n1) =>
    that match
      case Run(ch2, n2) => n1 <= n2 && (ch1 == ch2 || n1 == 0)
      case Conc(l, r)   => this <= l
  case Conc(l1, r1) =>
    that match
      case Conc(l2, r2) => l1 == l2 && r1 <= r2
      case _             => false
```

```

def < (that: IndentWidth): Boolean =
  this <= that && !(that <= this)

override def toString: String =
  this match
    case Run(ch, n) =>
      val kind = ch match
        case ' ' => "space"
        case '\t' => "tab"
        case _ => s"$ch-character"
      val suffix = if n == 1 then "" else "s"
      s"$n $kind$suffix"
    case Conc(l, r) =>
      s"$l, $r"

object IndentWidth:
  private inline val MaxCached = 40

  private val spaces = IArray.tabulate(MaxCached + 1)(new Run(' ', _))
  private val tabs = IArray.tabulate(MaxCached + 1)(new Run('\t', _))

  def Run(ch: Char, n: Int): Run =
    if n <= MaxCached && ch == ' ' then
      spaces(n)
    else if n <= MaxCached && ch == '\t' then
      tabs(n)
    else
      new Run(ch, n)
  end Run


  val Zero = Run(' ', 0)
end IndentWidth

```

## Settings and Rewrites

Significant indentation is enabled by default. It can be turned off by giving any of the options `-no-indent`, `-old-syntax` and `-source 3.0-migration`. If indentation is turned off, it is nevertheless checked that indentation conforms to the logical program structure as defined by braces. If that is not the case, the compiler issues a warning.

The Scala 3 compiler can rewrite source code to indented code and back. When invoked with options `-rewrite -indent` it will rewrite braces to indented regions where possible. When invoked with options `-rewrite -no-indent` it will rewrite in the reverse direction, inserting braces for indentation regions. The `-indent` option only works on [new-style syntax](https://docs.scala-lang.org/scala3/reference/other-new-features/indentation.html). So to go from old-style syntax to new-style indented

code one has to invoke the compiler twice, first with options `-rewrite -new-syntax`, then again with options `-rewrite -indent`. To go in the opposite direction, from 

indented code to old-style syntax, it's `-rewrite -no-indent`, followed by `-rewrite -old-syntax`.

## Variant: Indentation Marker :

Generally, the possible indentation regions coincide with those regions where braces `{ ... }` are also legal, no matter whether the braces enclose an expression or a set of definitions. There is one exception, though: Arguments to function can be enclosed in braces but they cannot be simply indented instead. Making indentation always significant for function arguments would be too restrictive and fragile.

To allow such arguments to be written without braces, a variant of the indentation scheme is implemented under language import

```
import language.experimental.fewerBraces
```

This variant is more contentious and less stable than the rest of the significant indentation scheme. In this variant, a colon `:` at the end of a line is also one of the possible tokens that opens an indentation region. Examples:

```
times(10):  
  println("ah")  
  println("ha")
```

or

```
xs.map:  
  x =>  
    val y = x - 1  
    y * y
```

The colon is usable not only for lambdas and by-name parameters, but also even for ordinary parameters:

```
credentials ++ :  
  val file = Path.userHome / ".credentials"  
  if file.exists  
  then Seq(Credentials(file))  
  else Seq()
```

How does this syntax variant work? Colons at the end of lines are their own token

How does this syntax variant work? Colons at the end of lines are their own token, distinct from normal `:`. The Scala grammar is changed so that colons at end of lines are accepted at all points where an opening brace enclosing an argument is legal. Special provisions are taken so that method result types can still use a colon on the end of a line, followed by the actual type on the next.

[< New C...](#)[Safe Ini... >](#)

Copyright (c) 2002-2022, LAMP/EPFL

