Scala 3 Reference / Scala 3 Syntax Summary

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

Scala 3 Syntax Summary

Edit this page on GitHub

The following description of Scala tokens uses literal characters `‘c’` when referring to the ASCII fragment `\u0000` – `\u007F`.

Unicode escapes are used to represent the Unicode character with the given hexadecimal code:

```
UnicodeEscape ::= ‘\’ ‘u’ {‘u’} hexDigit hexDigit hexDigit hexDigit ;
hexDigit      ::= ‘0’ | … | ‘9’ | ‘A’ | … | ‘F’ | ‘a’ | … | ‘f’ ;
```

Informal descriptions are typeset as `“some comment”`.

Lexical Syntax

The lexical syntax of Scala is given by the following grammar in EBNF form.

```
whiteSpace       ::=  ‘\u0020’ | ‘\u0009’ | ‘\u000D’ | ‘\u000A’ ;
upper            ::=  ‘A’ | … | ‘Z’ | ‘\$’ | ‘_’  “… and Unicode category Lu”
lower            ::=  ‘a’ | … | ‘z’ “… and Unicode category Ll” ;
letter           ::=  upper | lower “… and Unicode categories Lo, Lt, Nl” ;
digit            ::=  ‘0’ | … | ‘9’ ;
paren            ::=  ‘(’ | ‘)’ | ‘[’ | ‘]’ | ‘{’ | ‘}’ | ‘'(’ | ‘'[’ | ‘'{’ ;
delim            ::=  ‘`’ | ‘'’ | ‘"’ | ‘.’ | ‘;’ | ‘,’ ;
opchar           ::=  “printableChar not matched by (whiteSpace | upper |
                       lower | letter | digit | paren | delim | opchar |
                       Unicode_Sm | Unicode_So)” ;
printableChar    ::=  “all characters in [\u0020, \u007F] inclusive” ;
charEscapeSeq    ::=  ‘\’ (‘b’ | ‘t’ | ‘n’ | ‘f’ | ‘r’ | ‘"’ | ‘'’ | ‘\’) ;

op               ::=  opchar {opchar} ;
varid            ::=  lower idrest ;
alphaid          ::=  upper idrest
                   |  varid ;
plainid          ::=  alphaid
```

```
                   |  op ;
id               ::=  plainid
                   |  ‘`’ { charNoBackQuoteOrNewline | UnicodeEscape | charEsca
idrest           ::=  {letter | digit} [‘_’ op] ;
quoteId          ::=  ‘'’ alphaid ;

integerLiteral   ::=  (decimalNumeral | hexNumeral) [‘L’ | ‘l’] ;
decimalNumeral   ::=  ‘0’ | nonZeroDigit [{digit | ‘_’} digit] ;
hexNumeral       ::=  ‘0’ (‘x’ | ‘X’) hexDigit [{hexDigit | ‘_’} hexDigit] ;
nonZeroDigit     ::=  ‘1’ | … | ‘9’ ;

floatingPointLiteral
                 ::=  [decimalNumeral] ‘.’ digit [{digit | ‘_’} digit] [exponen
                   |  decimalNumeral exponentPart [floatType]
                   |  decimalNumeral floatType ;
exponentPart     ::=  (‘E’ | ‘e’) [‘+’ | ‘-’] digit [{digit | ‘_’} digit] ;
floatType        ::=  ‘F’ | ‘f’ | ‘D’ | ‘d’ ;

booleanLiteral   ::=  ‘true’ | ‘false’ ;

characterLiteral ::=  ‘'’ (printableChar | charEscapeSeq) ‘'’ ;

stringLiteral    ::=  ‘"’ {stringElement} ‘"’
                   |  ‘"""’ multiLineChars ‘"""’ ;
stringElement    ::=  printableChar \ (‘"’ | ‘\’)
                   |  UnicodeEscape
                   |  charEscapeSeq ;
multiLineChars   ::=  {[‘"’] [‘"’] char \ ‘"’} {‘"’} ;
processedStringLiteral
                 ::=  alphaid ‘"’ {[‘\’] processedStringPart | ‘\\’ | ‘\"’} ‘"
                   |  alphaid ‘"""’ {[‘"’] [‘"’] char \ (‘"’ | ‘$’) | escape}
processedStringPart
                 ::= printableChar \ (‘"’ | ‘$’ | ‘\’) | escape ;
escape           ::=  ‘$$’
                   |  ‘$’ letter { letter | digit }
                   |  ‘{’ Block  [‘;’ whiteSpace stringFormat whiteSpace] ‘}’
stringFormat     ::=  {printableChar \ (‘"’ | ‘}’ | ‘ ’ | ‘\t’ | ‘\n’)} ;

symbolLiteral    ::=  ‘'’ plainid // until 2.13 ;

comment          ::=  ‘/*’ “any sequence of characters; nested comments are al
                   |  ‘//’ “any sequence of characters up to end of line” ;

nl               ::=  “new line character” ;
semi             ::=  ‘;’ |  nl {nl} ;
```

Optional Braces

The lexical analyzer also inserts `indent` and `outdent` tokens that represent regions of indented code at certain points.

In the context-free productions below we use the notation `<<< ts >>>` to indicate a token sequence `ts` that is either enclosed in a pair of braces `{ ts }` or that constitutes an indented region `indent ts outdent`. Analogously, the notation `:<<< ts >>>` indicates a token sequence `ts` that is either enclosed in a pair of braces `{ ts }` or that constitutes an indented region `indent ts outdent` that follows a `:` at the end of a line.

```
 <<< ts >>>   ::=  ‘{’ ts ‘}’
                |  indent ts outdent ;
:<<< ts >>>   ::=  [nl] ‘{’ ts ‘}’
                |  `:` indent ts outdent ;
```

Keywords

Regular keywords

```
abstract  case      catch     class     def       do        else
enum      export    extends   false     final     finally   for
given     if        implicit  import    lazy      match     new
null      object    override  package   private   protected return
sealed    super     then      throw     trait     true      try
type      val       var       while     with      yield
:         =         <-        =>        <:        >:        #
@         =>>       ?=>
```

Soft keywords

```
as  derives  end  extension  infix  inline  opaque  open  throws
transparent  using  |  *  +  -
```

See the separate section on soft keywords for additional details on where a soft keyword is recognized.

Context-free Syntax

The context-free syntax of Scala is given by the following EBNF grammar:

Literals and Paths

```
SimpleLiteral     ::=  [‘-’] integerLiteral
                    |  [‘-’] floatingPointLiteral
                    |  booleanLiteral
                    |  characterLiteral
                    |  stringLiteral ;
Literal           ::=  SimpleLiteral
                    |  processedStringLiteral
                    |  symbolLiteral
                    |  ‘null’ ;

QualId            ::=  id {‘.’ id} ;
ids               ::=  id {‘,’ id} ;

SimpleRef         ::=  id
                    |  [id ‘.’] ‘this’
                    |  [id ‘.’] ‘super’ [ClassQualifier] ‘.’ id ;

ClassQualifier    ::=  ‘[’ id ‘]’ ;
```

Types

```
Type              ::=  FunType
                    |  HkTypeParamClause ‘=>>’ Type
                    |  FunParamClause ‘=>>’ Type
                    |  MatchType
                    |  InfixType ;
FunType           ::=  FunTypeArgs (‘=>’ | ‘?=>’) Type
                    |  HKTypeParamClause '=>' Type ;
FunTypeArgs       ::=  InfixType
                    |  ‘(’ [ FunArgTypes ] ‘)’
                    |  FunParamClause ;
FunParamClause    ::=  ‘(’ TypedFunParam {‘,’ TypedFunParam } ‘)’ ;
TypedFunParam     ::=  id ‘:’ Type ;
MatchType         ::=  InfixType `match` <<< TypeCaseClauses >>> ;
InfixType         ::=  RefinedType {id [nl] RefinedType} ;
RefinedType       ::=  AnnotType {[nl] Refinement} ;
AnnotType         ::=  SimpleType {Annotation} ;

SimpleType        ::=  SimpleLiteral
                    |  ‘?’ TypeBounds
                    |  id
                    |  Singleton ‘.’ id
                    |  Singleton ‘.’ ‘type’
                    |  ‘(’ Types ‘)’
                    |  Refinement
                    |  ‘$’ ‘{’ Block ‘}’
                    |  ‘$’ ‘{’ Pattern ‘}’
                    |  SimpleType1 TypeArgs
```

```
                     |  SimpleType1 ‘#’ id ;
Singleton           ::=  SimpleRef
                     |  SimpleLiteral
                     |  Singleton ‘.’ id ;

FunArgType          ::=  Type
                     |  ‘=>’ Type ;
FunArgTypes         ::=  FunArgType { ‘,’ FunArgType } ;
ParamType           ::=  [‘=>’] ParamValueType ;
ParamValueType      ::=  Type [‘*’] ;
TypeArgs            ::=  ‘[’ Types ‘]’ ;
Refinement          ::=  ‘{’ [RefineDcl] {semi [RefineDcl]} ‘}’ ;
TypeBounds          ::=  [‘>:’ Type] [‘<:’ Type] ;
TypeParamBounds     ::=  TypeBounds {‘:’ Type} ;
Types               ::=  Type {‘,’ Type} ;
```

Expressions

```
Expr                ::=  FunParams (‘=>’ | ‘?=>’) Expr
                     |  HkTypeParamClause ‘=>’ Expr
                     |  Expr1 ;
BlockResult         ::=  FunParams (‘=>’ | ‘?=>’) Block
                     |  HkTypeParamClause ‘=>’ Block
                     |  Expr1 ;
FunParams           ::=  Bindings
                     |  id
                     |  ‘_’ ;
Expr1               ::=  [‘inline’] ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Ex
                     |  [‘inline’] ‘if’  Expr ‘then’ Expr [[semi] ‘else’ Expr]
                     |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
                     |  ‘while’ Expr ‘do’ Expr
                     |  ‘try’ Expr Catches [‘finally’ Expr]
                     |  ‘try’ Expr [‘finally’ Expr]
                     |  ‘throw’ Expr
                     |  ‘return’ [Expr]
                     |  ForExpr
                     |  [SimpleExpr ‘.’] id ‘=’ Expr
                     |  PrefixOperator SimpleExpr ‘=’ Expr
                     |  SimpleExpr ArgumentExprs ‘=’ Expr
                     |  PostfixExpr [Ascription]
                     |  ‘inline’ InfixExpr MatchClause ;
Ascription          ::=  ‘:’ InfixType
                     |  ‘:’ Annotation {Annotation} ;
Catches             ::=  ‘catch’ (Expr | ExprCaseClause) ;
PostfixExpr         ::=  InfixExpr [id]
InfixExpr           ::=  PrefixExpr
                     |  InfixExpr id [nl] InfixExpr
                     |  InfixExpr MatchClause ;
```

```
MatchClause       ::=  ‘match’ <<< CaseClauses >>> ;
PrefixExpr        ::=  [PrefixOperator] SimpleExpr ;
PrefixOperator    ::=  ‘-’ | ‘+’ | ‘~’ | ‘!’ ;
SimpleExpr        ::=  SimpleRef
                    |  Literal
                    |  ‘_’
                    |  BlockExpr
                    |  ‘$’ ‘{’ Block ‘}’
                    |  ‘$’ ‘{’ Pattern ‘}’
                    |  Quoted
                    |  quoteId
                    |  ‘new’ ConstrApp {‘with’ ConstrApp} [TemplateBody]
                    |  ‘new’ TemplateBody
                    |  ‘(’ ExprsInParens ‘)’
                    |  SimpleExpr ‘.’ id
                    |  SimpleExpr ‘.’ MatchClause
                    |  SimpleExpr TypeArgs
                    |  SimpleExpr ArgumentExprs ;
Quoted            ::=  ‘'’ ‘{’ Block ‘}’
                    |  ‘'’ ‘[’ Type ‘]’ ;
ExprsInParens     ::=  ExprInParens {‘,’ ExprInParens} ;
ExprInParens      ::=  PostfixExpr ‘:’ Type
                    |  Expr ;
ParArgumentExprs  ::=  ‘(’ [‘using’] ExprsInParens ‘)’
                    |  ‘(’ [ExprsInParens ‘,’] PostfixExpr ‘*’ ‘)’ ;
ArgumentExprs     ::=  ParArgumentExprs
                    |  BlockExpr ;
BlockExpr         ::=  <<< (CaseClauses | Block) >>> ;
Block             ::=  {BlockStat semi} [BlockResult] ;
BlockStat         ::=  Import
                    |  {Annotation {nl}} {LocalModifier} Def
                    |  Extension
                    |  Expr1
                    |  EndMarker ;

ForExpr           ::=  ‘for’ ‘(’ Enumerators0 ‘)’ {nl} [‘do‘ | ‘yield’] Expr
                    |  ‘for’ ‘{’ Enumerators0 ‘}’ {nl} [‘do‘ | ‘yield’] Expr
                    |  ‘for’     Enumerators0          (‘do‘ | ‘yield’) Expr ;
Enumerators0      ::=  {nl} Enumerators [semi] ;
Enumerators       ::=  Generator {semi Enumerator | Guard} ;
Enumerator        ::=  Generator
                    |  Guard {Guard}
                    |  Pattern1 ‘=’ Expr ;
Generator         ::=  [‘case’] Pattern1 ‘<-’ Expr ;
Guard             ::=  ‘if’ PostfixExpr ;

CaseClauses       ::=  CaseClause { CaseClause } ;
CaseClause        ::=  ‘case’ Pattern [Guard] ‘=>’ Block ;
ExprCaseClause    ::=  ‘case’ Pattern [Guard] ‘=>’ Expr ;
```

```
TypeCaseClauses   ::=  TypeCaseClause { TypeCaseClause } ;
TypeCaseClause    ::=  ‘case’ InfixType ‘=>’ Type [semi] ;

Pattern           ::=  Pattern1 { ‘|’ Pattern1 } ;
Pattern1          ::=  Pattern2 [‘:’ RefinedType] ;
Pattern2          ::=  [id ‘@’] InfixPattern [‘*’] ;
InfixPattern      ::=  SimplePattern { id [nl] SimplePattern } ;
SimplePattern     ::=  PatVar
                    |  Literal
                    |  ‘(’ [Patterns] ‘)’
                    |  Quoted
                    |  SimplePattern1 [TypeArgs] [ArgumentPatterns]
                    |  ‘given’ RefinedType ;
SimplePattern1    ::=  SimpleRef
                    |  SimplePattern1 ‘.’ id ;
PatVar            ::=  varid
                    |  ‘_’ ;
Patterns          ::=  Pattern {‘,’ Pattern} ;
ArgumentPatterns  ::=  ‘(’ [Patterns] ‘)’
                    |  ‘(’ [Patterns ‘,’] PatVar ‘*’ ‘)’ ;
```

Type and Value Parameters

```
ClsTypeParamClause::=  ‘[’ ClsTypeParam {‘,’ ClsTypeParam} ‘]’ ;
ClsTypeParam      ::=  {Annotation} [‘+’ | ‘-’] id [HkTypeParamClause] TypePara

DefTypeParamClause::=  ‘[’ DefTypeParam {‘,’ DefTypeParam} ‘]’ ;
DefTypeParam      ::=  {Annotation} id [HkTypeParamClause] TypeParamBounds ;

TypTypeParamClause::=  ‘[’ TypTypeParam {‘,’ TypTypeParam} ‘]’ ;
TypTypeParam      ::=  {Annotation} id [HkTypeParamClause] TypeBounds ;

HkTypeParamClause ::=  ‘[’ HkTypeParam {‘,’ HkTypeParam} ‘]’ ;
HkTypeParam       ::=  {Annotation} [‘+’ | ‘-’] (id [HkTypeParamClause] | ‘_’)

ClsParamClauses   ::=  {ClsParamClause} [[nl] ‘(’ [‘implicit’] ClsParams ‘)’]
ClsParamClause    ::=  [nl] ‘(’ ClsParams ‘)’
                    |  [nl] ‘(’ ‘using’ (ClsParams | FunArgTypes) ‘)’ ;
ClsParams         ::=  ClsParam {‘,’ ClsParam} ;
ClsParam          ::=  {Annotation} [{Modifier} (‘val’ | ‘var’) | ‘inline’] Pa
Param             ::=  id ‘:’ ParamType [‘=’ Expr] ;

DefParamClauses   ::=  {DefParamClause} [[nl] ‘(’ [‘implicit’] DefParams ‘)’]
DefParamClause    ::=  [nl] ‘(’ DefParams ‘)’ | UsingParamClause ;
UsingParamClause  ::=  [nl] ‘(’ ‘using’ (DefParams | FunArgTypes) ‘)’ ;
DefParams         ::=  DefParam {‘,’ DefParam} ;
DefParam          ::=  {Annotation} [‘inline’] Param ;
```

Bindings and Imports

```
Bindings          ::=  ‘(’ [Binding {‘,’ Binding}] ‘)’ ;
Binding           ::=  (id | ‘_’) [‘:’ Type] ;

Modifier          ::=  LocalModifier
                    |  AccessModifier
                    |  ‘override’
                    |  ‘opaque’ ;
LocalModifier     ::=  ‘abstract’
                    |  ‘final’
                    |  ‘sealed’
                    |  ‘open’
                    |  ‘implicit’
                    |  ‘lazy’
                    |  ‘inline’ ;
AccessModifier    ::=  (‘private’ | ‘protected’) [AccessQualifier] ;
AccessQualifier   ::=  ‘[’ id ‘]’ ;

Annotation        ::=  ‘@’ SimpleType1 {ParArgumentExprs} ;

Import            ::=  ‘import’ ImportExpr {‘,’ ImportExpr} ;
Export            ::=  ‘export’ ImportExpr {‘,’ ImportExpr} ;
ImportExpr        ::=  SimpleRef {‘.’ id} ‘.’ ImportSpec
                    |  SimpleRef ‘as’ id ;
ImportSpec        ::=  NamedSelector
                    |  WildcardSelector
                    | ‘{’ ImportSelectors) ‘}’ ;
NamedSelector     ::=  id [‘as’ (id | ‘_’)] ;
WildCardSelector  ::=  ‘*’ | ‘given’ [InfixType] ;
ImportSelectors   ::=  NamedSelector [‘,’ ImportSelectors]
                    |  WildCardSelector {‘,’ WildCardSelector} ;

EndMarker         ::=  ‘end’ EndMarkerTag    -- when followed by EOL ;
EndMarkerTag      ::=  id | ‘if’ | ‘while’ | ‘for’ | ‘match’ | ‘try’
                    |  ‘new’ | ‘this’ | ‘given’ | ‘extension’ | ‘val’ ;
```

Declarations and Definitions

```
RefineDcl         ::=  ‘val’ ValDcl
                    |  ‘def’ DefDcl
                    |  ‘type’ {nl} TypeDcl ;
Dcl               ::=  RefineDcl
                    |  ‘var’ VarDcl ;
ValDcl            ::=  ids ‘:’ Type ;
VarDcl            ::=  ids ‘:’ Type ;
DefDcl            ::=  DefSig ‘:’ Type ;
```

```
DefSig            ::=  id [DefTypeParamClause] DefParamClauses ;
TypeDcl           ::=  id [TypeParamClause] {FunParamClause} TypeBounds [‘=’ Ty

Def               ::=  ‘val’ PatDef
                    |  ‘var’ PatDef
                    |  ‘def’ DefDef
                    |  ‘type’ {nl} TypeDcl
                    |  TmplDef ;
PatDef            ::=  ids [‘:’ Type] ‘=’ Expr
                    |  Pattern2 [‘:’ Type] ‘=’ Expr ;
DefDef            ::=  DefSig [‘:’ Type] ‘=’ Expr
                    |  ‘this’ DefParamClause DefParamClauses ‘=’ ConstrExpr ;

TmplDef           ::=  ([‘case’] ‘class’ | ‘trait’) ClassDef
                    |  [‘case’] ‘object’ ObjectDef
                    |  ‘enum’ EnumDef
                    |  ‘given’ GivenDef ;
ClassDef          ::=  id ClassConstr [Template] ;
ClassConstr       ::=  [ClsTypeParamClause] [ConstrMods] ClsParamClauses ;
ConstrMods        ::=  {Annotation} [AccessModifier] ;
ObjectDef         ::=  id [Template] ;
EnumDef           ::=  id ClassConstr InheritClauses EnumBody ;
GivenDef          ::=  [GivenSig] (AnnotType [‘=’ Expr] | StructuralInstance)
GivenSig          ::=  [id] [DefTypeParamClause] {UsingParamClause} ‘:’
StructuralInstance ::=  ConstrApp {‘with’ ConstrApp} [‘with’ TemplateBody] ;
Extension         ::=  ‘extension’ [DefTypeParamClause] {UsingParamClause}
                       ‘(’ DefParam ‘)’ {UsingParamClause} ExtMethods ;
ExtMethods        ::=  ExtMethod | [nl] <<< ExtMethod {semi ExtMethod} >>> ;
ExtMethod         ::=  {Annotation [nl]} {Modifier} ‘def’ DefDef ;
Template          ::=  InheritClauses [TemplateBody] ;
InheritClauses    ::=  [‘extends’ ConstrApps] [‘derives’ QualId {‘,’ QualId}]
ConstrApps        ::=  ConstrApp ({‘,’ ConstrApp} | {‘with’ ConstrApp}) ;
ConstrApp         ::=  SimpleType1 {Annotation} {ParArgumentExprs} ;
ConstrExpr        ::=  SelfInvocation
                    |  <<< SelfInvocation {semi BlockStat} >>> ;
SelfInvocation    ::=  ‘this’ ArgumentExprs {ArgumentExprs} ;

TemplateBody      ::=  :<<< [SelfType] TemplateStat {semi TemplateStat} >>> ;
TemplateStat      ::=  Import
                    |  Export
                    |  {Annotation [nl]} {Modifier} Def
                    |  {Annotation [nl]} {Modifier} Dcl
                    |  Extension
                    |  Expr1
                    |  EndMarker
                    | ;
SelfType          ::=  id [‘:’ InfixType] ‘=>’
                    |  ‘this’ ‘:’ InfixType ‘=>’ ;
EnumBody          ::=  :<<< [SelfType] EnumStat {semi EnumStat} >>> ;
EnumStat          ::=  TemplateStat
                    |  {Annotation [nl]} {Modifier} EnumCase ;
EnumCase          ::=  ‘case’ (id ClassConstr [‘extends’ ConstrApps]] | ids) ;

TopStats          ::=  TopStat {semi TopStat} ;
TopStat           ::=  Import
                    |  Export
                    |  {Annotation [nl]} {Modifier} Def
                    |  Extension
                    |  Packaging
                    |  PackageObject
                    |  EndMarker
                    | ;
Packaging         ::=  ‘package’ QualId :<<< TopStats >>> ;
PackageObject     ::=  ‘package’ ‘object’ ObjectDef ;

CompilationUnit   ::=  {‘package’ QualId semi} TopStats ;
```

‹ Tupled ...

Langua... ›