# Chapter 19. Syntax

This chapter repeats the syntactic grammar given in Chapters 4, 6-10, 14, and 15, as well as key parts of the lexical grammar from Chapter 3, using the notation from §2.4.

**Productions from §3 (*Lexical Structure*)**

```
Identifier:
  IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
  JavaLetter {JavaLetterOrDigit}

JavaLetter:
  any Unicode character that is a "Java letter"

JavaLetterOrDigit:
  any Unicode character that is a "Java letter-or-digit"

TypeIdentifier:
  Identifier but not var

Literal:
  IntegerLiteral
  FloatingPointLiteral
  BooleanLiteral
  CharacterLiteral
  StringLiteral
  NullLiteral
```

**Productions from §4 (*Types, Values, and Variables*)**

```
Type:
  PrimitiveType
  ReferenceType

PrimitiveType:
  {Annotation} NumericType
  {Annotation} boolean

NumericType:
  IntegralType
  FloatingPointType

IntegralType:
  (one of)
  byte short int long char

FloatingPointType:
  (one of)
  float double

ReferenceType:
  ClassOrInterfaceType
  TypeVariable
  ArrayType

ClassOrInterfaceType:
```





```
  ClassType
  InterfaceType

ClassType:
  {Annotation} TypeIdentifier [TypeArguments]
  PackageName . {Annotation} TypeIdentifier [TypeArguments]
  ClassOrInterfaceType . {Annotation} TypeIdentifier [TypeArguments]

InterfaceType:
  ClassType

TypeVariable:
  {Annotation} TypeIdentifier

ArrayType:
  PrimitiveType Dims
  ClassOrInterfaceType Dims
  TypeVariable Dims

Dims:
  {Annotation} [ ] {{Annotation} [ ]}

TypeParameter:
  {TypeParameterModifier} TypeIdentifier [TypeBound]

TypeParameterModifier:
  Annotation

TypeBound:
  extends TypeVariable
  extends ClassOrInterfaceType {AdditionalBound}

AdditionalBound:
  & InterfaceType

TypeArguments:
  < TypeArgumentList >

TypeArgumentList:
  TypeArgument {, TypeArgument}

TypeArgument:
  ReferenceType
  Wildcard

Wildcard:
  {Annotation} ? [WildcardBounds]

WildcardBounds:
  extends ReferenceType
  super ReferenceType
```

**Productions from §6 (*Names*)**

```
ModuleName:
  Identifier
  ModuleName . Identifier

PackageName:
  Identifier
  PackageName . Identifier

TypeName:
```

```
  TypeIdentifier
  PackageOrTypeName . TypeIdentifier

ExpressionName:
  Identifier
  AmbiguousName . Identifier

MethodName:
  Identifier

PackageOrTypeName:
  Identifier
  PackageOrTypeName . Identifier

AmbiguousName:
  Identifier
  AmbiguousName . Identifier
```

**Productions from §7 (*Packages and Modules*)**

```
CompilationUnit:
  OrdinaryCompilationUnit
  ModularCompilationUnit

OrdinaryCompilationUnit:
  [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

ModularCompilationUnit:
  {ImportDeclaration} ModuleDeclaration

PackageDeclaration:
  {PackageModifier} package Identifier {. Identifier} ;

PackageModifier:
  Annotation

ImportDeclaration:
  SingleTypeImportDeclaration
  TypeImportOnDemandDeclaration
  SingleStaticImportDeclaration
  StaticImportOnDemandDeclaration

SingleTypeImportDeclaration:
  import TypeName ;

TypeImportOnDemandDeclaration:
  import PackageOrTypeName . * ;

SingleStaticImportDeclaration:
  import static TypeName . Identifier ;

StaticImportOnDemandDeclaration:
  import static TypeName . * ;

TypeDeclaration:
  ClassDeclaration
  InterfaceDeclaration
  ;

ModuleDeclaration:
  {Annotation} [open] module Identifier {. Identifier} { {ModuleDirective} }

ModuleDirective:
```





```
  requires {RequiresModifier} ModuleName ;
  exports PackageName [to ModuleName {, ModuleName}] ;
  opens PackageName [to ModuleName {, ModuleName}] ;
  uses TypeName ;
  provides TypeName with TypeName {, TypeName} ;

RequiresModifier:
  (one of)
  transitive static
```

**Productions from §8 (*Classes*)**

```
ClassDeclaration:
  NormalClassDeclaration
  EnumDeclaration

NormalClassDeclaration:
  {ClassModifier} class TypeIdentifier [TypeParameters] [Superclass] [Superinterfaces] ClassBody

ClassModifier:
  (one of)
  Annotation public protected private
  abstract static final strictfp

TypeParameters:
  < TypeParameterList >

TypeParameterList:
  TypeParameter {, TypeParameter}

Superclass:
  extends ClassType

Superinterfaces:
  implements InterfaceTypeList

InterfaceTypeList:
  InterfaceType {, InterfaceType}

ClassBody:
  { {ClassBodyDeclaration} }

ClassBodyDeclaration:
  ClassMemberDeclaration
  InstanceInitializer
  StaticInitializer
  ConstructorDeclaration

ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  ;

FieldDeclaration:
  {FieldModifier} UnannType VariableDeclaratorList ;

FieldModifier:
  (one of)
  Annotation public protected private
  static final transient volatile
```

*VariableDeclaratorList:*
  *VariableDeclarator {, VariableDeclarator}*

*VariableDeclarator:*
  *VariableDeclaratorId [= VariableInitializer]*

*VariableDeclaratorId:*
  *Identifier [Dims]*

*VariableInitializer:*
  *Expression*
  *ArrayInitializer*

*UnannType:*
  *UnannPrimitiveType*
  *UnannReferenceType*

*UnannPrimitiveType:*
  *NumericType*
  boolean

*UnannReferenceType:*
  *UnannClassOrInterfaceType*
  *UnannTypeVariable*
  *UnannArrayType*

*UnannClassOrInterfaceType:*
  *UnannClassType*
  *UnannInterfaceType*

*UnannClassType:*
  *TypeIdentifier [TypeArguments]*
  *PackageName* . *{Annotation} TypeIdentifier [TypeArguments]*
  *UnannClassOrInterfaceType* . *{Annotation} TypeIdentifier [TypeArguments]*

*UnannInterfaceType:*
  *UnannClassType*

*UnannTypeVariable:*
  *TypeIdentifier*

*UnannArrayType:*
  *UnannPrimitiveType Dims*
  *UnannClassOrInterfaceType Dims*
  *UnannTypeVariable Dims*

*MethodDeclaration:*
  *{MethodModifier} MethodHeader MethodBody*

*MethodModifier:*
  *(one of)*
  *Annotation* public protected private
  abstract static final synchronized native strictfp

*MethodHeader:*
  *Result MethodDeclarator [Throws]*
  *TypeParameters {Annotation} Result MethodDeclarator [Throws]*

*Result:*
  *UnannType*
  void

*MethodDeclarator:*
  *Identifier* ( *[ReceiverParameter ,] [FormalParameterList]* ) *[Dims]*





*ReceiverParameter:*
  *{Annotation} UnannType [Identifier .]* this

*FormalParameterList:*
  *FormalParameter {, FormalParameter}*

*FormalParameter:*
  *{VariableModifier} UnannType VariableDeclaratorId*
  *VariableArityParameter*

*VariableArityParameter:*
  *{VariableModifier} UnannType {Annotation}* ... *Identifier*

*VariableModifier:*
  *Annotation*
  final

*Throws:*
  throws *ExceptionTypeList*

*ExceptionTypeList:*
  *ExceptionType {, ExceptionType}*

*ExceptionType:*
  *ClassType*
  *TypeVariable*

*MethodBody:*
  *Block*
  ;

*InstanceInitializer:*
  *Block*

*StaticInitializer:*
  static *Block*

*ConstructorDeclaration:*
  *{ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody*

*ConstructorModifier:*
  *(one of)*
  *Annotation* public protected private

*ConstructorDeclarator:*
  *[TypeParameters] SimpleTypeName* ( *[ReceiverParameter ,] [FormalParameterList]* )

*SimpleTypeName:*
  *TypeIdentifier*

*ConstructorBody:*
  { *[ExplicitConstructorInvocation] [BlockStatements]* }

*ExplicitConstructorInvocation:*
  *[TypeArguments]* this ( *[ArgumentList]* ) ;
  *[TypeArguments]* super ( *[ArgumentList]* ) ;
  *ExpressionName* . *[TypeArguments]* super ( *[ArgumentList]* ) ;
  *Primary* . *[TypeArguments]* super ( *[ArgumentList]* ) ;

*EnumDeclaration:*
  *{ClassModifier}* enum *TypeIdentifier [Superinterfaces] EnumBody*

*EnumBody:*
  { *[EnumConstantList] [,] [EnumBodyDeclarations]* }

*EnumConstantList:*

*EnumConstant* {, *EnumConstant*}

*EnumConstant:*
{*EnumConstantModifier*} *Identifier* [( [*ArgumentList*] )] [*ClassBody*]

*EnumConstantModifier:*
*Annotation*

*EnumBodyDeclarations:*
; {*ClassBodyDeclaration*}

**Productions from §9 (*Interfaces*)**

*InterfaceDeclaration:*
*NormalInterfaceDeclaration*
*AnnotationTypeDeclaration*

*NormalInterfaceDeclaration:*
{*InterfaceModifier*} interface *TypeIdentifier* [*TypeParameters*] [*ExtendsInterfaces*] *InterfaceBody*

*InterfaceModifier:*
*(one of)*
*Annotation* public protected private
abstract static strictfp

*ExtendsInterfaces:*
extends *InterfaceTypeList*

*InterfaceBody:*
{ {*InterfaceMemberDeclaration*} }

*InterfaceMemberDeclaration:*
*ConstantDeclaration*
*InterfaceMethodDeclaration*
*ClassDeclaration*
*InterfaceDeclaration*
;

*ConstantDeclaration:*
{*ConstantModifier*} *UnannType* *VariableDeclaratorList* ;

*ConstantModifier:*
*(one of)*
*Annotation* public
static final

*InterfaceMethodDeclaration:*
{*InterfaceMethodModifier*} *MethodHeader* *MethodBody*

*InterfaceMethodModifier:*
*(one of)*
*Annotation* public private
abstract default static strictfp

*AnnotationTypeDeclaration:*
{*InterfaceModifier*} @ interface *TypeIdentifier* *AnnotationTypeBody*

*AnnotationTypeBody:*
{ {*AnnotationTypeMemberDeclaration*} }

*AnnotationTypeMemberDeclaration:*





*AnnotationTypeElementDeclaration*
*ConstantDeclaration*
*ClassDeclaration*
*InterfaceDeclaration*
;

*AnnotationTypeElementDeclaration:*
{*AnnotationTypeElementModifier*} *UnannType* *Identifier* ( ) [*Dims*] [*DefaultValue*] ;

*AnnotationTypeElementModifier:*
*(one of)*
*Annotation* public
abstract

*DefaultValue:*
default *ElementValue*

*Annotation:*
*NormalAnnotation*
*MarkerAnnotation*
*SingleElementAnnotation*

*NormalAnnotation:*
@ *TypeName* ( [*ElementValuePairList*] )

*ElementValuePairList:*
*ElementValuePair* {, *ElementValuePair*}

*ElementValuePair:*
*Identifier* = *ElementValue*

*ElementValue:*
*ConditionalExpression*
*ElementValueArrayInitializer*
*Annotation*

*ElementValueArrayInitializer:*
{ [*ElementValueList*] [,] }

*ElementValueList:*
*ElementValue* {, *ElementValue*}

*MarkerAnnotation:*
@ *TypeName*

*SingleElementAnnotation:*
@ *TypeName* ( *ElementValue* )

**Productions from §10 (*Arrays*)**

*ArrayInitializer:*
{ [*VariableInitializerList*] [,] }

*VariableInitializerList:*
*VariableInitializer* {, *VariableInitializer*}

**Productions from §14 (*Blocks and Statements*)**

*Block:*
{ [*BlockStatements*] }

*BlockStatements:*
*BlockStatement* {*BlockStatement*}

*BlockStatement:*
*LocalVariableDeclarationStatement*
*ClassDeclaration*
*Statement*

*LocalVariableDeclarationStatement:*
*LocalVariableDeclaration* ;

*LocalVariableDeclaration:*
{*VariableModifier*} *LocalVariableType* *VariableDeclaratorList*

*LocalVariableType:*
*UnannType*
var

*Statement:*
*StatementWithoutTrailingSubstatement*
*LabeledStatement*
*IfThenStatement*
*IfThenElseStatement*
*WhileStatement*
*ForStatement*

*StatementNoShortIf:*
*StatementWithoutTrailingSubstatement*
*LabeledStatementNoShortIf*
*IfThenElseStatementNoShortIf*
*WhileStatementNoShortIf*
*ForStatementNoShortIf*

*StatementWithoutTrailingSubstatement:*
*Block*
*EmptyStatement*
*ExpressionStatement*
*AssertStatement*
*SwitchStatement*
*DoStatement*
*BreakStatement*
*ContinueStatement*
*ReturnStatement*
*SynchronizedStatement*
*ThrowStatement*
*TryStatement*

*EmptyStatement:*
;

*LabeledStatement:*
*Identifier* : *Statement*

*LabeledStatementNoShortIf:*
*Identifier* : *StatementNoShortIf*

*ExpressionStatement:*
*StatementExpression* ;

*StatementExpression:*
*Assignment*
*PreIncrementExpression*
*PreDecrementExpression*
*PostIncrementExpression*





*PostDecrementExpression*
*MethodInvocation*
*ClassInstanceCreationExpression*

*IfThenStatement:*
if ( *Expression* ) *Statement*

*IfThenElseStatement:*
if ( *Expression* ) *StatementNoShortIf* else *Statement*

*IfThenElseStatementNoShortIf:*
if ( *Expression* ) *StatementNoShortIf* else *StatementNoShortIf*

*AssertStatement:*
assert *Expression* ;
assert *Expression* : *Expression* ;

*SwitchStatement:*
switch ( *Expression* ) *SwitchBlock*

*SwitchBlock:*
{ {*SwitchBlockStatementGroup*} {*SwitchLabel*} }

*SwitchBlockStatementGroup:*
*SwitchLabels* *BlockStatements*

*SwitchLabels:*
*SwitchLabel* {*SwitchLabel*}

*SwitchLabel:*
case *ConstantExpression* :
case *EnumConstantName* :
default :

*EnumConstantName:*
*Identifier*

*WhileStatement:*
while ( *Expression* ) *Statement*

*WhileStatementNoShortIf:*
while ( *Expression* ) *StatementNoShortIf*

*DoStatement:*
do *Statement* while ( *Expression* ) ;

*ForStatement:*
*BasicForStatement*
*EnhancedForStatement*

*ForStatementNoShortIf:*
*BasicForStatementNoShortIf*
*EnhancedForStatementNoShortIf*

*BasicForStatement:*
for ( [*ForInit*] ; [*Expression*] ; [*ForUpdate*] ) *Statement*

*BasicForStatementNoShortIf:*
for ( [*ForInit*] ; [*Expression*] ; [*ForUpdate*] ) *StatementNoShortIf*

*ForInit:*
*StatementExpressionList*
*LocalVariableDeclaration*

*ForUpdate:*
*StatementExpressionList*

```
StatementExpressionList:
  StatementExpression {, StatementExpression}

EnhancedForStatement:
  for ( {VariableModifier} LocalVariableType VariableDeclaratorId : Expression ) Statement

EnhancedForStatementNoShortIf:
  for ( {VariableModifier} LocalVariableType VariableDeclaratorId : Expression ) StatementNoShortIf

BreakStatement:
  break [Identifier] ;

ContinueStatement:
  continue [Identifier] ;

ReturnStatement:
  return [Expression] ;

ThrowStatement:
  throw Expression ;

SynchronizedStatement:
  synchronized ( Expression ) Block

TryStatement:
  try Block Catches
  try Block [Catches] Finally
  TryWithResourcesStatement

Catches:
  CatchClause {CatchClause}

CatchClause:
  catch ( CatchFormalParameter ) Block

CatchFormalParameter:
  {VariableModifier} CatchType VariableDeclaratorId

CatchType:
  UnannClassType {| ClassType}

Finally:
  finally Block

TryWithResourcesStatement:
  try ResourceSpecification Block [Catches] [Finally]

ResourceSpecification:
  ( ResourceList [;] )

ResourceList:
  Resource {; Resource}

Resource:
  {VariableModifier} LocalVariableType Identifier = Expression
  VariableAccess
```

**Productions from §15 (*Expressions*)**

```
Primary:
  PrimaryNoNewArray
  ArrayCreationExpression
```





```
PrimaryNoNewArray:
  Literal
  ClassLiteral
  this
  TypeName . this
  ( Expression )
  ClassInstanceCreationExpression
  FieldAccess
  ArrayAccess
  MethodInvocation
  MethodReference

ClassLiteral:
  TypeName {[ ]} . class
  NumericType {[ ]} . class
  boolean {[ ]} . class
  void . class

ClassInstanceCreationExpression:
  UnqualifiedClassInstanceCreationExpression
  ExpressionName . UnqualifiedClassInstanceCreationExpression
  Primary . UnqualifiedClassInstanceCreationExpression

UnqualifiedClassInstanceCreationExpression:
  new [TypeArguments] ClassOrInterfaceTypeToInstantiate ( [ArgumentList] ) [ClassBody]

ClassOrInterfaceTypeToInstantiate:
  {Annotation} Identifier {. {Annotation} Identifier} [TypeArgumentsOrDiamond]

TypeArgumentsOrDiamond:
  TypeArguments
  <>

FieldAccess:
  Primary . Identifier
  super . Identifier
  TypeName . super . Identifier

ArrayAccess:
  ExpressionName [ Expression ]
  PrimaryNoNewArray [ Expression ]

MethodInvocation:
  MethodName ( [ArgumentList] )
  TypeName . [TypeArguments] Identifier ( [ArgumentList] )
  ExpressionName . [TypeArguments] Identifier ( [ArgumentList] )
  Primary . [TypeArguments] Identifier ( [ArgumentList] )
  super . [TypeArguments] Identifier ( [ArgumentList] )
  TypeName . super . [TypeArguments] Identifier ( [ArgumentList] )

ArgumentList:
  Expression {, Expression}

MethodReference:
  ExpressionName :: [TypeArguments] Identifier
  Primary :: [TypeArguments] Identifier
  ReferenceType :: [TypeArguments] Identifier
  super :: [TypeArguments] Identifier
  TypeName . super :: [TypeArguments] Identifier
  ClassType :: [TypeArguments] new
  ArrayType :: new

ArrayCreationExpression:
```

```
  new PrimitiveType DimExprs [Dims]
  new ClassOrInterfaceType DimExprs [Dims]
  new PrimitiveType Dims ArrayInitializer
  new ClassOrInterfaceType Dims ArrayInitializer

DimExprs:
  DimExpr {DimExpr}

DimExpr:
  {Annotation} [ Expression ]

Expression:
  LambdaExpression
  AssignmentExpression

LambdaExpression:
  LambdaParameters -> LambdaBody

LambdaParameters:
  ( [LambdaParameterList] )
  Identifier

LambdaParameterList:
  LambdaParameter {, LambdaParameter}
  Identifier {, Identifier}

LambdaParameter:
  {VariableModifier} LambdaParameterType VariableDeclaratorId
  VariableArityParameter

LambdaParameterType:
  UnannType
  var

LambdaBody:
  Expression
  Block

AssignmentExpression:
  ConditionalExpression
  Assignment

Assignment:
  LeftHandSide AssignmentOperator Expression

LeftHandSide:
  ExpressionName
  FieldAccess
  ArrayAccess

AssignmentOperator:
  (one of)

  =  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=

ConditionalExpression:
  ConditionalOrExpression
  ConditionalOrExpression ? Expression : ConditionalExpression
  ConditionalOrExpression ? Expression : LambdaExpression

ConditionalOrExpression:
  ConditionalAndExpression
  ConditionalOrExpression || ConditionalAndExpression
```





```
ConditionalAndExpression:
  InclusiveOrExpression
  ConditionalAndExpression && InclusiveOrExpression

InclusiveOrExpression:
  ExclusiveOrExpression
  InclusiveOrExpression | ExclusiveOrExpression

ExclusiveOrExpression:
  AndExpression
  ExclusiveOrExpression ^ AndExpression

AndExpression:
  EqualityExpression
  AndExpression & EqualityExpression

EqualityExpression:
  RelationalExpression
  EqualityExpression == RelationalExpression
  EqualityExpression != RelationalExpression

RelationalExpression:
  ShiftExpression
  RelationalExpression < ShiftExpression
  RelationalExpression > ShiftExpression
  RelationalExpression <= ShiftExpression
  RelationalExpression >= ShiftExpression
  RelationalExpression instanceof ReferenceType

ShiftExpression:
  AdditiveExpression
  ShiftExpression << AdditiveExpression
  ShiftExpression >> AdditiveExpression
  ShiftExpression >>> AdditiveExpression

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

MultiplicativeExpression:
  UnaryExpression
  MultiplicativeExpression * UnaryExpression
  MultiplicativeExpression / UnaryExpression
  MultiplicativeExpression % UnaryExpression

UnaryExpression:
  PreIncrementExpression
  PreDecrementExpression
  + UnaryExpression
  - UnaryExpression
  UnaryExpressionNotPlusMinus

PreIncrementExpression:
  ++ UnaryExpression

PreDecrementExpression:
  -- UnaryExpression

UnaryExpressionNotPlusMinus:
  PostfixExpression
  ~ UnaryExpression
  ! UnaryExpression
  CastExpression
```

```
PostfixExpression:
  Primary
  ExpressionName
  PostIncrementExpression
  PostDecrementExpression

PostIncrementExpression:
  PostfixExpression ++

PostDecrementExpression:
  PostfixExpression --

CastExpression:
  ( PrimitiveType ) UnaryExpression
  ( ReferenceType {AdditionalBound} ) UnaryExpressionNotPlusMinus
  ( ReferenceType {AdditionalBound} ) LambdaExpression

ConstantExpression:
  Expression
```

Prev Chapter 18. Type Inference | Home | Next Appendix A. Limited License Grant

Legal Notice