Kotlin language specification

Version 1.5-rfc+0.1

Marat Akhin Mikhail Belyaev

Contents

Part I

Kotlin/Core

Introduction

Kotlin took inspiration from many programming languages, including (but not limited to) Java, Scala, C# and Groovy. One of the main ideas behind Kotlin is being *pragmatic*, i.e., being a programming language useful for day-to-day development, which helps the users get the job done via its features and its tools. Thus, a lot of design decisions were and still are influenced by how beneficial these decisions are for Kotlin users.

Kotlin is a multiplatform, statically typed, general-purpose programming language. Currently, as of version 1.4, it supports compilation to the following platforms.

- JVM (Java Virtual Machine)
- JS (JavaScript)
- Native (native binaries for various architectures)

Furthermore, it supports transparent interoperability between different platforms via its Kotlin Multiplatform Project (Kotlin MPP) feature.

The type system of Kotlin distinguishes at compile time between nullable and not-nullable types, achieving null-safety, i.e., guaranteeing the absence of runtime errors caused by the absence of value (i.e., `null` value). Kotlin also extends its static type system with elements of gradual and flow typing, for better interoperability with other languages and ease of development.

Kotlin is an object-oriented language which also has a lot of functional programming elements. From the object-oriented side, it supports nominal subtyping with bounded parametric polymorphism (akin to generics) and mixed-site variance. From the functional programming side, it has first-class support for higher-order functions and lambda literals.

This specification covers Kotlin/Core, i.e., fundamental parts of Kotlin which should function *mostly* the same way irregardless of the underlying platform. These parts include such important things as language expressions, declarations, type system and overload resolution.

> Important: due to the complexities of platform-specific implementations, platforms may extend, reduce or change the way some as-

pects of Kotlin/Core function. We mark these platform-dependent Kotlin/Core fragments in the specification to the best of our abilities.

Platform-specific parts of Kotlin and its multiplatform capabilities will be covered in their respective sub-specifications, i.e., Kotlin/JVM, Kotlin/JS and Kotlin/Native.

Compatibility

Kotlin Language Specification is still in progress and has **experimental** stability level, meaning no compatibility should be expected between even incremental releases, any functionality can be added, removed or changed without warning.

Experimental features

In several cases this specification discusses *experimental* Kotlin features, i.e., features which are still in active development and which may be changed in the future. When so, the specification talks about the *current* state of said features, with no guarantees of their future stability (or even existence in the language).

The experimental features are marked as such in the specification to the best of our abilities.

Acknowledgments

We would like to thank the following people for their invaluable help and feedback during the writing of this specification.

Note: the format is “First name Last name”, ordered by last name

- Zalim Bashorov
- Andrey Breslav
- Roman Elizarov
- Stanislav Erokhin
- Dmitrii Petrov
- Victor Petukhov
- Dmitry Savvinov
- Anastasiia Spaseeva
- Mikhail Zarechenskii
- Denis Zharkov

We would also like to thank Pandoc, its authors and community, as this specification would be much harder to implement without Pandoc's versatility and support.

Feedback

If you have any feedback for this document, feel free to create an issue at our GitHub. In case you prefer to use email, you can use marat.akhin@jetbrains.com and mikhail.belyaev@jetbrains.com.

Reference

If one needs to reference this specification, they may use the following:

> Marat Akhin, Mikhail Belyaev et al. "Kotlin language specification: Kotlin/Core", JetBrains / JetBrains Research, 2020

Chapter 1

Syntax and grammar

1.1 Lexical grammar

1.1.1 Whitespace and comments

LF: *<unicode character Line Feed U+000A>*

CR:
<unicode character Carriage Return U+000D>

ShebangLine:
`'#!'` {*<any character excluding CR and LF >*}

DelimitedComment:
`'/*'` { *DelimitedComment* | *<any character>* } `'*/'`

LineComment:
`'//'` {*<any character excluding CR and LF >*}

WS:
<one of the following characters: SPACE U+0020, TAB U+0009, Form Feed U+000C>

NL: *LF* | (*CR* [*LF*])

Hidden:
DelimitedComment | *LineComment* | *WS*

1.1.2 Keywords and operators

RESERVED:
`'...'`

DOT:
'.'

COMMA:
','

LPAREN:
'('

RPAREN:
')'

LSQUARE:
'['

RSQUARE:
']'

LCURL:
'{'

RCURL:
'}'

MULT:
'*'

MOD:
'%'

DIV:
'/'

ADD:
'+'

SUB:
'-'

INCR:
'++'

DECR:
'--'

CONJ:
'&&'

DISJ:
'||'

EXCL_WS:
'!' *Hidden*

EXCL_NO_WS:
'!'

COLON:
':'

SEMICOLON:
';'

ASSIGNMENT:
'='

ADD_ASSIGNMENT:
'+='

SUB_ASSIGNMENT:
'-='

MULT_ASSIGNMENT:
'*='

DIV_ASSIGNMENT:
'/='

MOD_ASSIGNMENT:
'%='

ARROW:
'->'

DOUBLE_ARROW:
'=>'

RANGE:
'..'

COLONCOLON:
'::'

DOUBLE_SEMICOLON:
';;'

HASH:
'#'

AT_NO_WS:
'@'

AT_POST_WS:
'@' (*Hidden* | *NL*)

AT_PRE_WS:
(*Hidden* | *NL*) '@'

AT_BOTH_WS:
(*Hidden* | *NL*) `'@'` (*Hidden* | *NL*)

QUEST_WS:
`'?'` *Hidden*

QUEST_NO_WS:
`'?'`

LANGLE:
`'<'`

RANGLE:
`'>'`

LE: `'<='`

GE:
`'>='`

EXCL_EQ:
`'!='`

EXCL_EQEQ:
`'!=='`

AS_SAFE:
`'as?'`

EQEQ:
`'=='`

EQEQEQ:
`'==='`

SINGLE_QUOTE:
`'\''`

RETURN_AT:
`'return@'` *Identifier*

CONTINUE_AT:
`'continue@'` *Identifier*

BREAK_AT:
`'break@'` *Identifier*

THIS_AT:
`'this@'` *Identifier*

SUPER_AT:
`'super@'` *Identifier*

FILE:
`'file'`

FIELD:
    'field'

PROPERTY:
    'property'

GET:
    'get'

SET:
    'set'

RECEIVER:
    'receiver'

PARAM:
    'param'

SETPARAM:
    'setparam'

DELEGATE:
    'delegate'

PACKAGE:
    'package'

IMPORT:
    'import'

CLASS:
    'class'

INTERFACE:
    'interface'

FUN:
    'fun'

OBJECT:
    'object'

VAL:
    'val'

VAR:
    'var'

TYPE_ALIAS:
    'typealias'

CONSTRUCTOR:
    'constructor'

BY:
    `'by'`

COMPANION:
    `'companion'`

INIT:
    `'init'`

THIS:
    `'this'`

SUPER:
    `'super'`

TYPEOF:
    `'typeof'`

WHERE:
    `'where'`

IF: `'if'`

ELSE:
    `'else'`

WHEN:
    `'when'`

TRY:
    `'try'`

CATCH:
    `'catch'`

FINALLY:
    `'finally'`

FOR:
    `'for'`

DO:
    `'do'`

WHILE:
    `'while'`

THROW:
    `'throw'`

RETURN:
    `'return'`

CONTINUE:
    `'continue'`

BREAK:
    `'break'`

AS: `'as'`

IS: `'is'`

IN: `'in'`

NOT_IS:
    `'!is'` (*Hidden* | *NL*)

NOT_IN:
    `'!in'` (*Hidden* | *NL*)

OUT:
    `'out'`

DYNAMIC:
    `'dynamic'`

PUBLIC:
    `'public'`

PRIVATE:
    `'private'`

PROTECTED:
    `'protected'`

INTERNAL:
    `'internal'`

ENUM:
    `'enum'`

SEALED:
    `'sealed'`

ANNOTATION:
    `'annotation'`

DATA:
    `'data'`

INNER:
    `'inner'`

TAILREC:
    `'tailrec'`

OPERATOR:
    `'operator'`

INLINE:
    `'inline'`

INFIX:
    `'infix'`

EXTERNAL:
    `'external'`

SUSPEND:
    `'suspend'`

OVERRIDE:
    `'override'`

ABSTRACT:
    `'abstract'`

FINAL:
    `'final'`

OPEN:
    `'open'`

CONST:
    `'const'`

LATEINIT:
    `'lateinit'`

VARARG:
    `'vararg'`

NOINLINE:
    `'noinline'`

CROSSINLINE:
    `'crossinline'`

REIFIED:
    `'reified'`

EXPECT:
    `'expect'`

ACTUAL:
    `'actual'`

1.1.3 Literals

DecDigitNoZero:
    `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`

***DecDigit*:**
'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

***DecDigitOrSeparator*:**
DecDigit | '_'

***DecDigits*:**
DecDigit {*DecDigitOrSeparator*} *DecDigit*
| *DecDigit*

***DoubleExponent*:**
('e' | 'E') [('+' | '-')] *DecDigits*

***RealLiteral*:**
FloatLiteral | *DoubleLiteral*

***FloatLiteral*:**
DoubleLiteral ('f' | 'F')
| *DecDigits* ('f' | 'F')

***DoubleLiteral*:**
[*DecDigits*] '.' *DecDigits* [*DoubleExponent*]
| [*DecDigits*] [*DoubleExponent*]

***IntegerLiteral*:**
DecDigitNoZero {*DecDigitOrSeparator*} *DecDigit*
| *DecDigit*

***HexDigit*:**
DecDigit
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f'

***HexDigitOrSeparator*:**
HexDigit | '_'

HexLiteral
'0' ('x' | 'X') *HexDigit* {*HexDigitOrSeparator*} *HexDigit*
| '0' ('x' | 'X') *HexDigit*

BinDigit
'0' | '1'

BinDigitOrSeparator
BinDigit | '_'

BinLiteral
'0' ('b' | 'B') *BinDigit* {*BinDigitOrSeparator*} *BinDigit*
| '0' ('b' | 'B') *BinDigit*

UnsignedLiteral
(*IntegerLiteral* | *HexLiteral* | *BinLiteral*) ('u' | 'U') ['L']

LongLiteral
(*IntegerLiteral* | *HexLiteral* | *BinLiteral*) `'L'`

BooleanLiteral
`'true'` | `'false'`

NullLiteral
`'null'`

CharacterLiteral
`'''` (*EscapeSeq* | <any character excluding CR, LF, `'''` or `'\'`>) `'''`

UniCharacterLiteral
`'\' 'u'` *HexDigit HexDigit HexDigit HexDigit*

EscapedIdentifier
`'\'` (`'t'` | `'b'` | `'r'` | `'n'` | `'''` | `'"'` | `'\'` | `'$'`)

EscapeSeq
UniCharacterLiteral | *EscapedIdentifier*

1.1.4 Identifiers

Letter
<any unicode character of classes LL, LM, LO, LT, LU or NL>

QuotedSymbol
<any character excluding CR, LF and `'`'>

UnicodeDigit
<any unicode character of class ND>

Identifier
(*Letter* | `'_'`) {*Letter* | `'_'` | *UnicodeDigit*}
| `'`'` *QuotedSymbol* {*QuotedSymbol*} `'`'`

Kotlin supports *escaping* identifiers by enclosing any sequence of characters into backtick (`` ` ``) characters, allowing to use any name as an identifier. This allows not only using non-alphanumeric characters (like `@` or `#`) in names, but also using keywords like `if` or `when` as identifiers. Actual set of characters that is allowed to be escaped may, however, be a subject to platform restrictions. Consult particular platform sections for details.

> Note: for example, the following characters are not allowed in identifiers used as declaration names on the JVM platform even when escaped due to JVM restrictions: `(`, `)`, `{`, `}`, `[`, `]`, `.`

Escaped identifiers are treated the same as corresponding non-escaped identifier if it is allowed. For example, an escaped identifier `` `foo` `` and non-escaped identifier `foo` may be used interchangeably and refer to the same program entity.

IdentifierOrSoftKey

Identifier
| *ABSTRACT*
| *ANNOTATION*
| *BY*
| *CATCH*
| *COMPANION*
| *CONSTRUCTOR*
| *CROSSINLINE*
| *DATA*
| *DYNAMIC*
| *ENUM*
| *EXTERNAL*
| *FINAL*
| *FINALLY*
| *IMPORT*
| *INFIX*
| *INIT*
| *INLINE*
| *INNER*
| *INTERNAL*
| *LATEINIT*
| *NOINLINE*
| *OPEN*
| *OPERATOR*
| *OUT*
| *OVERRIDE*
| *PRIVATE*
| *PROTECTED*
| *PUBLIC*
| *REIFIED*
| *SEALED*
| *TAILREC*
| *VARARG*
| *WHERE*
| *GET*
| *SET*
| *FIELD*
| *PROPERTY*
| *RECEIVER*
| *PARAM*
| *SETPARAM*
| *DELEGATE*
| *FILE*
| *EXPECT*
| *ACTUAL*

| *CONST*
| *SUSPEND*

Some of the keywords used in Kotlin are allowed to be used as identifiers even when not escaped. Such keywords are called *soft keywords*. You can see the complete list of soft keyword in the rule above. All other keywords are considered *hard keywords* and may only be used as identifiers if escaped.

Note: for example, this is a valid property declaration in Kotlin:

```
val field = 2
```

even though `field` is a keyword

1.1.5 String mode grammar

QUOTE_OPEN
`'"'`

TRIPLE_QUOTE_OPEN
`'"""'`

FieldIdentifier
`'$'` *IdentifierOrSoftKey*

Opening a double quote (QUOTE_OPEN) rule puts the lexical grammar into the special "line string" mode with the following rules. Closing double quote (QUOTE_CLOSE) rule exits this mode.

QUOTE_CLOSE
`'"'`

LineStrRef
FieldIdentifier

LineStrText
{<any character except `'\'`, `'"'` or `'$'`>} | `'$'`

LineStrEscapedChar
EscapedIdentifier | *UniCharacterLiteral*

LineStrExprStart
`'${'`

Opening a triple double quote (TRIPLE_QUOTE_OPEN) rule puts the lexical grammar into the special "multiline string" mode with the following rules. Closing triple double quote (TRIPLE_QUOTE_CLOSE) rule exits this mode.

TRIPLE_QUOTE_CLOSE
[*MultilineStringQuote*] `'"""'`

MultilineStringQuote
    '"""' {'"'}

MultiLineStrRef
    FieldIdentifier

MultiLineStrText
    {<any character except '"' or '$'>} | '$'

MultiLineStrExprStart
    '${'

1.1.6 Tokens

These are all the valid tokens in one rule. Note that syntax grammar ignores tokens *DelimitedComment*, *LineComment* and *WS*.

KotlinToken
    ShebangLine
    | *DelimitedComment*
    | *LineComment*
    | *WS*
    | *NL*
    | *RESERVED*
    | *DOT*
    | *COMMA*
    | *LPAREN*
    | *RPAREN*
    | *LSQUARE*
    | *RSQUARE*
    | *LCURL*
    | *RCURL*
    | *MULT*
    | *MOD*
    | *DIV*
    | *ADD*
    | *SUB*
    | *INCR*
    | *DECR*
    | *CONJ*
    | *DISJ*
    | *EXCL_WS*
    | *EXCL_NO_WS*
    | *COLON*
    | *SEMICOLON*
    | *ASSIGNMENT*
    | *ADD_ASSIGNMENT*

| *SUB_ASSIGNMENT*
| *MULT_ASSIGNMENT*
| *DIV_ASSIGNMENT*
| *MOD_ASSIGNMENT*
| *ARROW*
| *DOUBLE_ARROW*
| *RANGE*
| *COLONCOLON*
| *DOUBLE_SEMICOLON*
| *HASH*
| *AT_NO_WS*
| *AT_POST_WS*
| *AT_PRE_WS*
| *AT_BOTH_WS*
| *QUEST_WS*
| *QUEST_NO_WS*
| *LANGLE*
| *RANGLE*
| *LE*
| *GE*
| *EXCL_EQ*
| *EXCL_EQEQ*
| *AS_SAFE*
| *EQEQ*
| *EQEQEQ*
| *SINGLE_QUOTE*
| *RETURN_AT*
| *CONTINUE_AT*
| *BREAK_AT*
| *THIS_AT*
| *SUPER_AT*
| *FILE*
| *FIELD*
| *PROPERTY*
| *GET*
| *SET*
| *RECEIVER*
| *PARAM*
| *SETPARAM*
| *DELEGATE*
| *PACKAGE*
| *IMPORT*
| *CLASS*
| *INTERFACE*
| *FUN*
| *OBJECT*

| *VAL*
| *VAR*
| *TYPE_ALIAS*
| *CONSTRUCTOR*
| *BY*
| *COMPANION*
| *INIT*
| *THIS*
| *SUPER*
| *TYPEOF*
| *WHERE*
| *IF*
| *ELSE*
| *WHEN*
| *TRY*
| *CATCH*
| *FINALLY*
| *FOR*
| *DO*
| *WHILE*
| *THROW*
| *RETURN*
| *CONTINUE*
| *BREAK*
| *AS*
| *IS*
| *IN*
| *NOT_IS*
| *NOT_IN*
| *OUT*
| *DYNAMIC*
| *PUBLIC*
| *PRIVATE*
| *PROTECTED*
| *INTERNAL*
| *ENUM*
| *SEALED*
| *ANNOTATION*
| *DATA*
| *INNER*
| *TAILREC*
| *OPERATOR*
| *INLINE*
| *INFIX*
| *EXTERNAL*
| *SUSPEND*

| *OVERRIDE*
| *ABSTRACT*
| *FINAL*
| *OPEN*
| *CONST*
| *LATEINIT*
| *VARARG*
| *NOINLINE*
| *CROSSINLINE*
| *REIFIED*
| *EXPECT*
| *ACTUAL*
| *Identifier*
| *RealLiteral*
| *IntegerLiteral*
| *HexLiteral*
| *BinLiteral*
| *LongLiteral*
| *BooleanLiteral*
| *NullLiteral*
| *CharacterLiteral*
| *QUOTE_OPEN*
| *QUOTE_CLOSE*
| *TRIPLE_QUOTE_OPEN*
| *TRIPLE_QUOTE_CLOSE*
| *LineStrRef*
| *LineStrText*
| *LineStrEscapedChar*
| *LineStrExprStart*
| *MultilineStringQuote*
| *MultiLineStrRef*
| *MultiLineStrText*
| *MultiLineStrExprStart*

EOF
<end of input>

1.2 Syntax grammar

The grammar below replaces some lexical grammar rules with explicit literals (where such replacement in trivial and always correct, for example, for keywords) for better readability.

***kotlinFile*:**
[*shebangLine*]

{*NL*}
{*fileAnnotation*}
packageHeader
importList
{*topLevelObject*}
EOF

***script*:**
[*shebangLine*]
{*NL*}
{*fileAnnotation*}
packageHeader
importList
{*statement semi*}
EOF

***shebangLine*:**
ShebangLine (*NL* {*NL*})

***fileAnnotation*:**
(*AT_NO_WS* | *AT_PRE_WS*)
`'file'`
{*NL*}
`':'`
{*NL*}
((`'['` (*unescapedAnnotation* {*unescapedAnnotation*}) `']'`) | *unescapedAnnotation*)
{*NL*}

***packageHeader*:**
[`'package'` *identifier* [*semi*]]

***importList*:**
{*importHeader*}

***importHeader*:**
`'import'` *identifier* [(`'.'` `'*'`) | *importAlias*] [*semi*]

***importAlias*:**
`'as'` *simpleIdentifier*

***topLevelObject*:**
declaration [*semis*]

***typeAlias*:**
[*modifiers*]
`'typealias'`
{*NL*}
simpleIdentifier
[{*NL*} *typeParameters*]

{*NL*}
'='
{*NL*}
type

***declaration*:**
classDeclaration
| *objectDeclaration*
| *functionDeclaration*
| *propertyDeclaration*
| *typeAlias*

***classDeclaration*:**
[*modifiers*]
('class' | (['fun' {*NL*}] 'interface'))
{*NL*}
simpleIdentifier
[{*NL*} *typeParameters*]
[{*NL*} *primaryConstructor*]
[{*NL*} ':' {*NL*} *delegationSpecifiers*]
[{*NL*} *typeConstraints*]
[({*NL*} *classBody*) | ({*NL*} *enumClassBody*)]

***primaryConstructor*:**
[[*modifiers*] 'constructor' {*NL*}] *classParameters*

***classBody*:**
'{'
{*NL*}
classMemberDeclarations
{*NL*}
'}'

***classParameters*:**
'('
{*NL*}
[*classParameter* {{*NL*} ',' {*NL*} *classParameter*} [{*NL*} ',']]
{*NL*}
')'

***classParameter*:**
[*modifiers*]
['val' | 'var']
{*NL*}
simpleIdentifier
':'
{*NL*}
type
[{*NL*} '=' {*NL*} *expression*]

***delegationSpecifiers*:**
annotatedDelegationSpecifier {{*NL*} ',' {*NL*} *annotatedDelegationSpecifier*}

***delegationSpecifier*:**
constructorInvocation
| *explicitDelegation*
| *userType*
| *functionType*

***constructorInvocation*:**
userType valueArguments

***annotatedDelegationSpecifier*:**
{*annotation*} {*NL*} *delegationSpecifier*

***explicitDelegation*:**
(*userType* | *functionType*)
{*NL*}
'by'
{*NL*}
expression

***typeParameters*:**
'<'
{*NL*}
typeParameter
{{*NL*} ',' {*NL*} *typeParameter*}
[{*NL*} ',']
{*NL*}
'>'

***typeParameter*:**
[*typeParameterModifiers*] {*NL*} *simpleIdentifier* [{*NL*} ':' {*NL*} *type*]

***typeConstraints*:**
'where' {*NL*} *typeConstraint* {{*NL*} ',' {*NL*} *typeConstraint*}

***typeConstraint*:**
{*annotation*}
simpleIdentifier
{*NL*}
':'
{*NL*}
type

***classMemberDeclarations*:**
{*classMemberDeclaration* [*semis*]}

***classMemberDeclaration*:**
declaration

| *companionObject*
| *anonymousInitializer*
| *secondaryConstructor*

***anonymousInitializer*:**
'init' {*NL*} *block*

***companionObject*:**
[*modifiers*]
'companion'
{*NL*}
'object'
[{*NL*} *simpleIdentifier*]
[{*NL*} ':' {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

***functionValueParameters*:**
'('
{*NL*}
[*functionValueParameter* {{*NL*} ',' {*NL*} *functionValueParameter*} [{*NL*}
','']]
{*NL*}
')'

***functionValueParameter*:**
[*parameterModifiers*] *parameter* [{*NL*} '=' {*NL*} *expression*]

***functionDeclaration*:**
[*modifiers*]
'fun'
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} '.']
{*NL*}
simpleIdentifier
{*NL*}
functionValueParameters
[{*NL*} ':' {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionBody*:**
block
| ('=' {*NL*} *expression*)

***variableDeclaration*:**
{*annotation*} {*NL*} *simpleIdentifier* [{*NL*} ':' {*NL*} *type*]

***multiVariableDeclaration*:**
'('
{*NL*}

variableDeclaration
{{*NL*} ',' {*NL*} *variableDeclaration*}
[{*NL*} ',']
{*NL*}
')'

***propertyDeclaration*:**
[*modifiers*]
('val' | 'var')
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} '.']
({*NL*} (*multiVariableDeclaration* | *variableDeclaration*))
[{*NL*} *typeConstraints*]
[{*NL*} (('=' {*NL*} *expression*) | *propertyDelegate*)]
[(*NL* {*NL*}) ';']
{*NL*}
(([*getter*] [{*NL*} [*semi*] *setter*]) | ([*setter*] [{*NL*} [*semi*] *getter*]))

***propertyDelegate*:**
'by' {*NL*} *expression*

***getter*:**
[*modifiers*] 'get' [{*NL*} '(' {*NL*} ')' [{*NL*} ':' {*NL*} *type*] {*NL*} *functionBody*]

***setter*:**
[*modifiers*] 'set' [{*NL*} '(' {*NL*} *functionValueParameterWithOptionalType* [{*NL*} ','] {*NL*} ')' [{*NL*} ':' {*NL*} *type*] {*NL*} *functionBody*]

***parametersWithOptionalType*:**
'('
{*NL*}
[*functionValueParameterWithOptionalType* {{*NL*} ',' {*NL*} *functionValueParameterWithOptionalType*} [{*NL*} ',']]
{*NL*}
')'

***functionValueParameterWithOptionalType*:**
[*parameterModifiers*] *parameterWithOptionalType* [{*NL*} '=' {*NL*} *expression*]

***parameterWithOptionalType*:**
simpleIdentifier {*NL*} [':' {*NL*} *type*]

***parameter*:**
simpleIdentifier
{*NL*}
':'
{*NL*}
type

***objectDeclaration*:**
[*modifiers*]
`'object'`
{*NL*}
simpleIdentifier
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

***secondaryConstructor*:**
[*modifiers*]
`'constructor'`
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *constructorDelegationCall*]
{*NL*}
[*block*]

***constructorDelegationCall*:**
(`'this'` | `'super'`) {*NL*} *valueArguments*

***enumClassBody*:**
`'{'`
{*NL*}
[*enumEntries*]
[{*NL*} `';'` {*NL*} *classMemberDeclarations*]
{*NL*}
`'}'`

***enumEntries*:**
enumEntry {{*NL*} `','` {*NL*} *enumEntry*} {*NL*} [`','`]

***enumEntry*:**
[*modifiers* {*NL*}] *simpleIdentifier* [{*NL*} *valueArguments*] [{*NL*} *classBody*]

***type*:**
[*typeModifiers*] (*parenthesizedType* | *nullableType* | *typeReference* | *functionType*)

***typeReference*:**
userType
| `'dynamic'`

***nullableType*:**
(*typeReference* | *parenthesizedType*) {*NL*} (*quest* {*quest*})

***quest*:**
QUEST_NO_WS
| *QUEST_WS*

***userType*:**
simpleUserType {{*NL*} `'.'` {*NL*} *simpleUserType*}

***simpleUserType*:**
simpleIdentifier [{*NL*} *typeArguments*]

***typeProjection*:**
([*typeProjectionModifiers*] *type*)
| `'*'`

***typeProjectionModifiers*:**
typeProjectionModifier {*typeProjectionModifier*}

***typeProjectionModifier*:**
(*varianceModifier* {*NL*})
| *annotation*

***functionType*:**
[*receiverType* {*NL*} `'.'` {*NL*}]
functionTypeParameters
{*NL*}
`'->'`
{*NL*}
type

***functionTypeParameters*:**
`'('`
{*NL*}
[*parameter* | *type*]
{{*NL*} `','` {*NL*} (*parameter* | *type*)}
[{*NL*} `','`]
{*NL*}
`')'`

***parenthesizedType*:**
`'('`
{*NL*}
type
{*NL*}
`')'`

***receiverType*:**
[*typeModifiers*] (*parenthesizedType* | *nullableType* | *typeReference*)

***parenthesizedUserType*:**
`'('`
{*NL*}
(*userType* | *parenthesizedUserType*)
{*NL*}
`')'`

***statements*:**
[*statement* {*semis statement*}] [*semis*]

***statement*:**
{*label* | *annotation*} (*declaration* | *assignment* | *loopStatement* | *expression*)

***label*:**
simpleIdentifier (*AT_NO_WS* | *AT_POST_WS*) {*NL*}

***controlStructureBody*:**
block
| *statement*

***block*:**
'{'
{*NL*}
statements
{*NL*}
'}'

***loopStatement*:**
forStatement
| *whileStatement*
| *doWhileStatement*

***forStatement*:**
'for'
{*NL*}
'('
{*annotation*}
(*variableDeclaration* | *multiVariableDeclaration*)
'in'
expression
')'
{*NL*}
[*controlStructureBody*]

***whileStatement*:**
'while'
{*NL*}
'('
expression
')'
{*NL*}
(*controlStructureBody* | ';')

***doWhileStatement*:**
'do'
{*NL*}
[*controlStructureBody*]
{*NL*}
'while'

{*NL*}
'('
expression
')'

***assignment*:**
((*directlyAssignableExpression* '=') | (*assignableExpression assignmentAndOperator*)) {*NL*} *expression*

***semi*:**
((';' | *NL*) {*NL*})
| *EOF*

***semis*:**
(';' | *NL* {';' | *NL*})
| *EOF*

***expression*:**
disjunction

***disjunction*:**
conjunction {{*NL*} '||' {*NL*} *conjunction*}

***conjunction*:**
equality {{*NL*} '&&' {*NL*} *equality*}

***equality*:**
comparison {*equalityOperator* {*NL*} *comparison*}

***comparison*:**
genericCallLikeComparison {*comparisonOperator* {*NL*} *genericCallLikeComparison*}

***genericCallLikeComparison*:**
infixOperation {*callSuffix*}

***infixOperation*:**
elvisExpression {(*inOperator* {*NL*} *elvisExpression*) | (*isOperator* {*NL*} *type*)}

***elvisExpression*:**
infixFunctionCall {{*NL*} *elvis* {*NL*} *infixFunctionCall*}

***elvis*:**
QUEST_NO_WS ':'

***infixFunctionCall*:**
rangeExpression {*simpleIdentifier* {*NL*} *rangeExpression*}

***rangeExpression*:**
additiveExpression {'..' {*NL*} *additiveExpression*}

***additiveExpression*:**
multiplicativeExpression {*additiveOperator* {*NL*} *multiplicativeExpression*}

***multiplicativeExpression*:**
asExpression {*multiplicativeOperator* {*NL*} *asExpression*}

***asExpression*:**
prefixUnaryExpression {{*NL*} *asOperator* {*NL*} *type*}

***prefixUnaryExpression*:**
{*unaryPrefix*} *postfixUnaryExpression*

***unaryPrefix*:**
annotation
| *label*
| (*prefixUnaryOperator* {*NL*})

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***directlyAssignableExpression*:**
(*postfixUnaryExpression assignableSuffix*)
| *simpleIdentifier*
| *parenthesizedDirectlyAssignableExpression*

***parenthesizedDirectlyAssignableExpression*:**
'('
{*NL*}
directlyAssignableExpression
{*NL*}
')'

***assignableExpression*:**
prefixUnaryExpression
| *parenthesizedAssignableExpression*

***parenthesizedAssignableExpression*:**
'('
{*NL*}
assignableExpression
{*NL*}
')'

***assignableSuffix*:**
typeArguments
| *indexingSuffix*
| *navigationSuffix*

***indexingSuffix*:**
'['
{*NL*}
expression
{{*NL*} ',' {*NL*} *expression*}
[{*NL*} ',']
{*NL*}
']'

***navigationSuffix*:**
memberAccessOperator {*NL*} (*simpleIdentifier* | *parenthesizedExpression* |
'class')

***callSuffix*:**
[*typeArguments*] (([*valueArguments*] *annotatedLambda*) | *valueArguments*)

***annotatedLambda*:**
{*annotation*} [*label*] {*NL*} *lambdaLiteral*

***typeArguments*:**
'<'
{*NL*}
typeProjection
{{*NL*} ',' {*NL*} *typeProjection*}
[{*NL*} ',']
{*NL*}
'>'

***valueArguments*:**
'(' {*NL*} [*valueArgument* {{*NL*} ',' {*NL*} *valueArgument*} [{*NL*} ',']
{*NL*}] ')'

***valueArgument*:**
[*annotation*]
{*NL*}
[*simpleIdentifier* {*NL*} '=' {*NL*}]
['*']
{*NL*}
expression

***primaryExpression*:**
parenthesizedExpression
| *simpleIdentifier*
| *literalConstant*
| *stringLiteral*

| *callableReference*
| *functionLiteral*
| *objectLiteral*
| *collectionLiteral*
| *thisExpression*
| *superExpression*
| *ifExpression*
| *whenExpression*
| *tryExpression*
| *jumpExpression*

***parenthesizedExpression*:**
'('
{*NL*}
expression
{*NL*}
')'

***collectionLiteral*:**
'[' {*NL*} [*expression* {{*NL*} ',' {*NL*} *expression*} [{*NL*} ','] {*NL*}] ']'

***literalConstant*:**
BooleanLiteral
| *IntegerLiteral*
| *HexLiteral*
| *BinLiteral*
| *CharacterLiteral*
| *RealLiteral*
| 'null'
| *LongLiteral*
| *UnsignedLiteral*

***stringLiteral*:**
lineStringLiteral
| *multiLineStringLiteral*

***lineStringLiteral*:**
'"' {*lineStringContent* | *lineStringExpression*} '"'

***multiLineStringLiteral*:**
'"""' {*multiLineStringContent* | *multiLineStringExpression* | '"'}
TRIPLE_QUOTE_CLOSE

***lineStringContent*:**
LineStrText
| *LineStrEscapedChar*
| *LineStrRef*

***lineStringExpression*:**
'${'

{*NL*}
expression
{*NL*}
'}'

multiLineStringContent:
MultiLineStrText
| '""'
| *MultiLineStrRef*

multiLineStringExpression:
'${'
{*NL*}
expression
{*NL*}
'}'

lambdaLiteral:
'{'
{*NL*}
[[*lambdaParameters*] {*NL*} '->' {*NL*}]
statements
{*NL*}
'}'

lambdaParameters:
lambdaParameter {{*NL*} ',' {*NL*} *lambdaParameter*} [{*NL*} ',']

lambdaParameter:
variableDeclaration
| (*multiVariableDeclaration* [{*NL*} ':' {*NL*} *type*])

anonymousFunction:
'fun'
[{*NL*} *type* {*NL*} '.']
{*NL*}
parametersWithOptionalType
[{*NL*} ':' {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

functionLiteral:
lambdaLiteral
| *anonymousFunction*

objectLiteral:
'object' [{*NL*} ':' {*NL*} *delegationSpecifiers* {*NL*}] [{*NL*} *classBody*]

thisExpression:
'this'

| *THIS_AT*

***superExpression*:**
('super' ['<' {*NL*} *type* {*NL*} '>'] [*AT_NO_WS simpleIdentifier*])
| *SUPER_AT*

***ifExpression*:**
'if'
{*NL*}
'('
{*NL*}
expression
{*NL*}
')'
{*NL*}
(*controlStructureBody* | ([*controlStructureBody*] {*NL*} [';'] {*NL*} 'else'
{*NL*} (*controlStructureBody* | ';')) | ';')

***whenSubject*:**
'(' [{*annotation*} {*NL*} 'val' {*NL*} *variableDeclaration* {*NL*} '=' {*NL*}]
expression ')'

***whenExpression*:**
'when'
{*NL*}
[*whenSubject*]
{*NL*}
'{'
{*NL*}
{*whenEntry* {*NL*}}
{*NL*}
'}'

***whenEntry*:**
(*whenCondition* {{*NL*} ',' {*NL*} *whenCondition*} [{*NL*} ','] {*NL*} '->'
{*NL*} *controlStructureBody* [*semi*])
| ('else' {*NL*} '->' {*NL*} *controlStructureBody* [*semi*])

***whenCondition*:**
expression
| *rangeTest*
| *typeTest*

***rangeTest*:**
inOperator {*NL*} *expression*

***typeTest*:**
isOperator {*NL*} *type*

***tryExpression*:**

'try' {*NL*} *block* ((({*NL*} *catchBlock* {{*NL*} *catchBlock*}) [{*NL*} *finallyBlock*]) | ({*NL*} *finallyBlock*))

***catchBlock*:**

'catch'
{*NL*}
'('
{*annotation*}
simpleIdentifier
':'
type
[{*NL*} ',']
')'
{*NL*}
block

***finallyBlock*:**

'finally' {*NL*} *block*

***jumpExpression*:**

('throw' {*NL*} *expression*)
| (('return' | *RETURN_AT*) [*expression*])
| 'continue'
| *CONTINUE_AT*
| 'break'
| *BREAK_AT*

***callableReference*:**

[*receiverType*] '::' {*NL*} (*simpleIdentifier* | 'class')

***assignmentAndOperator*:**

'+='
| '-='
| '*='
| '/='
| '%='

***equalityOperator*:**

'!='
| '!=='
| '=='
| '==='

***comparisonOperator*:**

'<'
| '>'
| '<='
| '>='

***inOperator*:**
'in'
| *NOT_IN*

***isOperator*:**
'is'
| *NOT_IS*

***additiveOperator*:**
'+'
| '-'

***multiplicativeOperator*:**
'*'
| '/'
| '%'

***asOperator*:**
'as'
| 'as?'

***prefixUnaryOperator*:**
'++'
| '--'
| '-'
| '+'
| *excl*

***postfixUnaryOperator*:**
'++'
| '--'
| ('!' *excl*)

***excl*:**
'!'
| *EXCL_WS*

***memberAccessOperator*:**
({*NL*} '.')
| ({*NL*} *safeNav*)
| '::'

***safeNav*:**
QUEST_NO_WS '.'

***modifiers*:**
annotation | *modifier* {*annotation* | *modifier*}

***parameterModifiers*:**
annotation | *parameterModifier* {*annotation* | *parameterModifier*}

***modifier*:**
(*classModifier* | *memberModifier* | *visibilityModifier* | *functionModifier* | *propertyModifier* | *inheritanceModifier* | *parameterModifier* | *platformModifier*) {*NL*}

***typeModifiers*:**
typeModifier {*typeModifier*}

***typeModifier*:**
annotation
| ('suspend' {*NL*})

***classModifier*:**
'enum'
| 'sealed'
| 'annotation'
| 'data'
| 'inner'
| 'value'

***memberModifier*:**
'override'
| 'lateinit'

***visibilityModifier*:**
'public'
| 'private'
| 'internal'
| 'protected'

***varianceModifier*:**
'in'
| 'out'

***typeParameterModifiers*:**
typeParameterModifier {*typeParameterModifier*}

***typeParameterModifier*:**
(*reificationModifier* {*NL*})
| (*varianceModifier* {*NL*})
| *annotation*

***functionModifier*:**
'tailrec'
| 'operator'
| 'infix'
| 'inline'
| 'external'
| 'suspend'

propertyModifier:
    'const'

inheritanceModifier:
    'abstract'
    | 'final'
    | 'open'

parameterModifier:
    'vararg'
    | 'noinline'
    | 'crossinline'

reificationModifier:
    'reified'

platformModifier:
    'expect'
    | 'actual'

annotation:
    (*singleAnnotation* | *multiAnnotation*) {*NL*}

singleAnnotation:
    ((*annotationUseSiteTarget* {*NL*}) | *AT_NO_WS* | *AT_PRE_WS*) *unescapedAnnotation*

multiAnnotation:
    ((*annotationUseSiteTarget* {*NL*}) | *AT_NO_WS* | *AT_PRE_WS*) '[' (*unescapedAnnotation* {*unescapedAnnotation*}) ']'

annotationUseSiteTarget:
    (*AT_NO_WS* | *AT_PRE_WS*) ('field' | 'property' | 'get' | 'set' | 'receiver' | 'param' | 'setparam' | 'delegate') {*NL*} ':'

unescapedAnnotation:
    constructorInvocation
    | *userType*

simpleIdentifier:
    Identifier
    | 'abstract'
    | 'annotation'
    | 'by'
    | 'catch'
    | 'companion'
    | 'constructor'
    | 'crossinline'
    | 'data'
    | 'dynamic'
    | 'enum'

```
| 'external'
| 'final'
| 'finally'
| 'get'
| 'import'
| 'infix'
| 'init'
| 'inline'
| 'inner'
| 'internal'
| 'lateinit'
| 'noinline'
| 'open'
| 'operator'
| 'out'
| 'override'
| 'private'
| 'protected'
| 'public'
| 'reified'
| 'sealed'
| 'tailrec'
| 'set'
| 'vararg'
| 'where'
| 'field'
| 'property'
| 'receiver'
| 'param'
| 'setparam'
| 'delegate'
| 'file'
| 'expect'
| 'actual'
| 'const'
| 'suspend'
| 'value'
```

***identifier*:**
simpleIdentifier {{*NL*} `'.'` *simpleIdentifier*}

1.3 Documentation comments

Kotlin supports special comment syntax for code documentation purposes,

called KDoc. The syntax is based on Markdown and Javadoc. Documentation comments start with `/**` and end with `*/` and allows external tools to generate documentation based on the comment contents.

Chapter 2

Type system

Glossary

T Type (with unknown nullability)

$T!!$ Non-nullable type

$T?$ Nullable type

$\{T\}$ Universe of all possible types

$\{T!!\}$
: Universe of non-nullable types

$\{T?\}$
: Universe of nullable types

Well-formed type
: A properly constructed type w.r.t. Kotlin type system

Γ Type context

$A <: B$
: A is a subtype of B

$A \not<:\not> B$
: A and B are not related w.r.t. subtyping

Type constructor
: An abstract type with one or more type parameters, which must be instantiated before use

Parameterized type
: A concrete type, which is the result of type constructor instantiation

Type parameter
: Formal type parameter of a type constructor

Type argument
: Actual type argument in a parameterized type

$T[A_1, \ldots, A_n]$
: The result of type constructor T instantiation with type arguments A_i

$T[\sigma]$ The result of type constructor $T(F_1, \ldots, F_n)$ instantiation with the assumed substitution $\sigma : F_1 = A_1, \ldots, F_n = A_n$

σT The result of type substitution in type T w.r.t. substitution σ

$K_T(F, A)$
: Captured type from the type capturing of type parameter F and type argument A in parameterized type T

$T\langle K_1, \ldots, K_n \rangle$
: The result of type capturing for parameterized type T with *captured* types K_i

$T\langle \tau \rangle$
: The result of type capturing for parameterized type $T(F_1, \ldots, F_n)$ with *captured* substitution $\tau : F_1 = K_1, \ldots, F_n = K_n$

$A \,\&\, B$
: Intersection type of A and B

$A \mid B$
: Union type of A and B

GLB
: Greatest lower bound

LUB
: Least upper bound

Introduction

Similarly to most other programming languages, Kotlin operates on data in the form of *values* or *objects*, which have *types* — descriptions of what is the expected behaviour and possible values for their datum. An empty value is represented by a special `null` object; most operations with it result in runtime errors or exceptions.

Kotlin has a type system with the following main properties.

- Hybrid static, gradual and flow type checking;
- Null safety;
- No unsafe implicit conversions;
- Unified top and bottom types;
- Nominal subtyping with bounded parametric polymorphism and mixed-site variance.

Type safety (consistency between compile and runtime types) is verified *statically*, at compile time, for the majority of Kotlin types. However, for better interoperability with platform-dependent code Kotlin also support a variant of *gradual types* in the form of flexible types. Even more so, in some cases the compile-time type of a value may *change* depending on the control- and data-flow of the program; a feature usually known as *flow typing*, represented in Kotlin as smart casts.

Null safety is enforced by having two type universes: *nullable* (with nullable types $T?$) and *non-nullable* (with non-nullable types $T!!$). A value of any non-nullable type cannot contain `null`, meaning all operations within the non-nullable type universe are safe w.r.t. empty values, i.e., should never result in a runtime error caused by `null`.

Implicit conversions between types in Kotlin are limited to safe upcasts w.r.t. subtyping, meaning all other (unsafe) conversions must be explicit, done via either a conversion function or an explicit cast. However, Kotlin also supports smart casts — a special kind of implicit conversions which are safe w.r.t. program control- and data-flow, which are covered in more detail here.

The unified supertype type for all types in Kotlin is `kotlin.Any?`, a nullable version of `kotlin.Any`. The unified subtype type for all types in Kotlin is `kotlin.Nothing`.

Kotlin uses nominal subtyping, meaning subtyping relation is defined when a type is declared, with bounded parametric polymorphism, implemented as generics via parameterized types. Subtyping between these parameterized types is defined through mixed-site variance.

2.1 Type kinds

For the purposes of this section, we establish the following type kinds — different flavours of types which exist in the Kotlin type system.

- Built-in types
- Classifier types
- Type parameters
- Function types
- Array types
- Flexible types
- Nullable types
- Intersection types
- Union types

We distinguish between *concrete* and *abstract* types. Concrete types are types which are assignable to values. Abstract types need to be instantiated as concrete types before they can be used as types for values.

> Note: for brevity, we omit specifying that a type is concrete. All types not described as abstract are implicitly concrete.

We further distinguish *concrete* types between *class* and *interface* types; as Kotlin is a language with single inheritance, sometimes it is important to discriminate between these kinds of types. Any given concrete type may be either a class or an interface type, but never both.

We also distinguish between *denotable* and *non-denotable* types. The former are types which are expressible in Kotlin and can be written by the end-user. The latter are special types which are *not* expressible in Kotlin and are used by the compiler in type inference, smart casts, etc.

2.1.1 Built-in types

Kotlin type system uses the following built-in types, which have special semantics and representation (or lack thereof).

`kotlin.Any`

`kotlin.Any` is the unified supertype ($\top$) for $\{T!!\}$, i.e., all non-nullable types are subtypes of `kotlin.Any`, either explicitly, implicitly, or by subtyping relation.

> Note: additional details about `kotlin.Any` are available here.

`kotlin.Nothing`

`kotlin.Nothing` is the unified subtype ($\bot$) for $\{T\}$, i.e., `kotlin.Nothing` is a subtype of all well-formed Kotlin types, including user-defined ones. This makes it an uninhabited type (as it is impossible for anything to be, for example, a function and an integer at the same time), meaning instances of this type can never exist at runtime; subsequently, there is no way to create an instance of `kotlin.Nothing` in Kotlin.

> Note: additional details about `kotlin.Nothing` are available here.

`kotlin.Function`

`kotlin.Function`(R) is the unified supertype of all function types. It is parameterized over function return type `R`.

Built-in integer types

Kotlin supports the following signed integer types.

- `kotlin.Int`
- `kotlin.Short`
- `kotlin.Byte`
- `kotlin.Long`

Besides their use as types, integer types are important w.r.t. integer literal types.

> Note: additional details about built-in integer types are available here.

Array types

Kotlin arrays are represented as a parameterized type `kotlin.Array`(T), where T is the type of the stored elements, which supports `get`/`set` operations. The `kotlin.Array`(T) type follows the rules of regular type constructors and parameterized types w.r.t. subtyping.

> Note: unlike Java, arrays in Kotlin are declared as invariant. To use them in a co- or contravariant way, one should use use-site variance.

In addition to the general `kotlin.Array`(T) type, Kotlin also has the following specialized array types:

- `DoubleArray` (for `kotlin.Array`(`kotlin.Double`))
- `FloatArray` (for `kotlin.Array`(`kotlin.Float`))
- `LongArray` (for `kotlin.Array`(`kotlin.Long`))
- `IntArray` (for `kotlin.Array`(`kotlin.Int`))
- `ShortArray` (for `kotlin.Array`(`kotlin.Short`))
- `ByteArray` (for `kotlin.Array`(`kotlin.Byte`))
- `CharArray` (for `kotlin.Array`(`kotlin.Char`))
- `BooleanArray` (for `kotlin.Array`(`kotlin.Boolean`))

These array types structurally match the corresponding `kotlin.Array`(T) type; i.e., `IntArray` has the same methods and properties as `kotlin.Array`(`kotlin.Int`). However, they are **not** related by subtyping; meaning one cannot pass a `BooleanArray` argument to a function expecting an `kotlin.Array`(`kotlin.Boolean`).

> Note: the presence of such specialized types allows the compiler to perform additional array-related optimizations.

> Note: specialized and non-specialized array types match modulo their iterator types, which are also specialized; `Iterator<Int>` is specialized to `IntIterator`.

Array type specialization ATS(A) is a transformation of a generic `kotlin.Array`(T) type to a corresponding specialized version, which works as follows.

- if `kotlin.Array`(T) has a specialized version `TArray`, ATS(`kotlin.Array`(T)) = `TArray`
- if `kotlin.Array`(T) does not have a specialized version, ATS(`kotlin.Array`(T)) = `kotlin.Array`(T)

ATS takes an important part in how variable length parameters are handled.

> Note: additional details about built-in array types are available here.

2.1.2 Classifier types

Classifier types represent regular types which are declared as classes, interfaces or objects. As Kotlin supports parametric polymorphism, there are two variants of classifier types: simple and parameterized.

Simple classifier types

A simple classifier type

$$T : S_1, \ldots, S_m$$

consists of

- type name T
- (optional) list of supertypes $S_1, \ldots, S_m$

To represent a well-formed simple classifier type, $T : S_1, \ldots, S_m$ should satisfy the following conditions.

- T is a valid type name
- $\forall i \in [1, m] : S_i$ must be concrete, non-nullable, well-formed type
- the transitive closure $\mathbb{S}^*(T)$ of the set of type supertypes $\mathbb{S}(T : S_1, \ldots, S_m) = \{S_1, \ldots, S_m\} \cup \mathbb{S}(S_1) \cup \ldots \cup \mathbb{S}(S_m)$ is *consistent*, i.e., does not contain two parameterized types with different type arguments.

  Example:

  ```
  // A well-formed type with no supertypes
  interface Base

  // A well-formed type with a single supertype Base
  interface Derived : Base

  // An ill-formed type,
  // as nullable type cannot be a supertype
  interface Invalid : Base?
  ```

  Note: for the purpose of different type system examples, we assume the presence of the following well-formed concrete types:

  - class `String`
  - interface `Number`
  - class `Int` <: `Number`
  - class `Double` <: `Number`

Note: `Number` is actually a built-in abstract class; we use it as an interface for illustrative purposes.

Parameterized classifier types

A classifier type constructor

$$T(F_1, \ldots, F_n) : S_1, \ldots, S_m$$

describes an abstract type and consists of

- type name T
- type parameters $F_1, \ldots, F_n$
- (optional) list of supertypes $S_1, \ldots, S_m$

To represent a well-formed type constructor, $T(F_1, \ldots, F_n) : S_1, \ldots, S_m$ should satisfy the following conditions.

- T is a valid type name
- $\forall i \in [1, n] : F_i$ must be well-formed type parameter
- $\forall j \in [1, m] : S_j$ must be concrete, non-nullable, well-formed type

To instantiate a type constructor, one provides it with type arguments, creating a concrete parameterized classifier type

$$T[A_1, \ldots, A_n]$$

which consists of

- type constructor T
- type arguments $A_1, \ldots, A_n$

To represent a well-formed parameterized type, $T[A_1, \ldots, A_n]$ should satisfy the following conditions.

- T is a well-formed type constructor with n type parameters
- $\forall i \in [1, n] : A_i$ must be well-formed concrete type
- $\forall i \in [1, n]$: variance of A_i does not contradict variance of F_i
- $\forall i \in [1, n] : A_i <: \tau U_i$, where U_i is the upper bound for F_i and captured substitution $\tau : F_1 = K_1, \ldots, F_n = K_n$ manipulates captured types.
- the transitive closure $\mathbb{S}^*(T)$ of the set of type supertypes $\mathbb{S}(T\langle\tau\rangle : \tau S_1, \ldots, \tau S_m) = \{\tau S_1, \ldots, \tau S_m\} \cup \mathbb{S}(\tau S_1) \cup \ldots \cup \mathbb{S}(\tau S_m)$ is *consistent*, i.e., does not contain two parameterized types with different type arguments.

  Example:

```
// A well-formed type constructor with no supertypes
// A and B are unbounded type parameters
interface Generic<A, B>

// A well-formed type constructor
//   with a single parameterized supertype
// Int and String are well-formed concrete types
interface ConcreteDerived<P, Q> : Generic<Int, String>

// A well-formed type constructor
//   with a single parameterized supertype
// P and Q are type parameters of GenericDerived,
//   used as type arguments of Generic
interface GenericDerived<P, Q> : Generic<P, Q>

// An ill-formed type constructor,
//   as abstract type Generic
//   cannot be used as a supertype
interface Invalid<P> : Generic


// A well-formed type constructor with no supertypes
// out A is a projected type parameter
interface Out<out A>


// A well-formed type constructor with no supertypes
// S : Number is a bounded type parameter
// (S <: Number)
interface NumberWrapper<S : Number>

// A well-formed type constructor
//   with a single parameterized supertype
// NumberWrapper<Int> is well-formed,
//   as Int <: Number
interface IntWrapper : NumberWrapper<Int>

// An ill-formed type constructor,
//   as NumberWrapper<String> is an ill-formed parameterized type
//   (String not(<:>) Number)
interface InvalidWrapper : NumberWrapper<String>
```

2.1.3 Type parameters

Type parameters are a special kind of types, which are introduced by type

constructors. They are considered well-formed concrete types only in the type context of their declaring type constructor.

When creating a parameterized type from a type constructor, its type parameters with their respective type arguments go through capturing and create *captured* types, which follow special rules described in more detail below.

Type parameters may be either unbounded or bounded. By default, a type parameter F is unbounded, which is the same as saying it is a bounded type parameter of the form $F <: \texttt{kotlin.Any?}$.

A bounded type parameter additionally specifies upper type bounds for the type parameter and is defined as $F <: B_1, \ldots, B_n$, where B_i is an i-th upper bound on type parameter F.

To represent a well-formed bounded type parameter of type constructor T, $F <: B_1, \ldots, B_n$ should satisfy either of the following sets of conditions.

- Bounded type parameter with regular bounds:
  - F is a type parameter of type constructor T
  - $\forall i \in [1, n] : B_i$ must be concrete, non-type-parameter, well-formed type
  - No more than one of B_i may be a class type

  Note: the last condition is a nod to the single inheritance nature of Kotlin: any type may be a subtype of no more than one class type. For any two class types, either these types are in a subtyping relation (and you should use the more specific type in the bounded type parameter), or they are unrelated (and the bounded type parameter is empty).

  Actual support for multiple class type bounds would be needed only in very rare cases, such as the following example.

```
interface Foo
interface Bar

open class A<T>
class B<T> : A<T>

class C<T> where T : A<out Foo>, T : B<out Bar>
// A convoluted way of saying T <: B<out Foo & Bar>,
// which contains a non-denotable intersection type
```

- Bounded type parameter with type parameter bound:
  - F is a type parameter of type constructor T
  - $i = 1$ (i.e., there is a single upper bound)
  - B_1 must be well-formed type parameter

From the definition, it follows $F <: B_1, \ldots, B_n$ can be represented as $F <: U$ where $U = B_1 \,\&\, \ldots \,\&\, B_n$ (aka intersection type).

Function type parameters

Function type parameters are a flavor of type parameters, which are used in function declarations to create parameterized functions. They are considered well-formed concrete types only in the type context of their declaring function.

> Note: one may view such parameterized functions as a kind of function type constructors.

Function type parameters work similarly to regular type parameters, however, they do not support specifying mixed-site variance.

Mixed-site variance

To implement subtyping between parameterized types, Kotlin uses *mixed-site variance* — a combination of declaration- and use-site variance, which is easier to understand and reason about, compared to wildcards from Java. Mixed-site variance means you can specify, whether you want your parameterized type to be co-, contra- or invariant on some type parameter, both in type parameter (declaration-site) and type argument (use-site).

> Info: *variance* is a way of describing how subtyping works for *variant* parameterized types. With declaration-site variance, for two non-equivalent types $A <: B$, subtyping between `T<A>` and `T<B>` depends on the variance of type parameter F for some type constructor T.
>
> - if F is covariant (`out` F), `T<A>` $<:$ `T<B>`
> - if F is contravariant(`in` F), `T<A>` $:>$ `T<B>`
> - if F is invariant (default), `T<A>` $\not<:\not>$ `T<B>`
>
> Use-site variance allows the user to change the type variance of an *invariant* type parameter by specifying it on the corresponding type argument. `out` A means covariant type argument, `in` A means contravariant type argument; for two non-equivalent types $A <: B$ and an invariant type parameter F of some type constructor T, subtyping for use-site variance has the following rules.
>
> - `T<out A>` $<:$ `T<out B>`
> - `T<in A>` $:>$ `T<in B>`
> - `T<A>` $<:$ `T<out A>`
> - `T<A>` $<:$ `T<in A>`
>
> Important: by the transitivity of the subtyping operator these rules imply that the following also holds:
>
> - `T<A>` $<:$ `T<out B>`
> - `T<in A>` $:>$ `T<B>`

> Note: Kotlin does not support specifying both co- and contravariance at the same time, i.e., it is impossible to have `T<out A in B>` neither on declaration- nor on use-site.
>
> Note: informally, covariant type parameter `out` A of type constructor T means "T is a producer of As and gets them out"; contravariant type parameter `in` A of type constructor T means "T is a consumer of As and takes them in".

For further discussion about mixed-site variance and its practical applications, we readdress you to subtyping.

Declaration-site variance

A type parameter F may be invariant, covariant or contravariant.

By default, all type parameters are invariant.

To specify a covariant type parameter, it is marked as `out` F. To specify a contravariant type parameter, it is marked as `in` F.

The variance information is used by subtyping and for checking allowed operations on values of co- and contravariant type parameters.

> Important: declaration-site variance can be used only when declaring types, e.g., function type parameters cannot be variant.
>
> Example:

```
// A type constructor with an invariant type parameter
interface Invariant<A>
// A type constructor with a covariant type parameter
interface Out<out A>
// A type constructor with a contravariant type parameter
interface In<in A>

fun testInvariant() {
    var invInt: Invariant<Int> = ...
    var invNumber: Invariant<Number> = ...

    if (random) invInt = invNumber // ERROR
    else invNumber = invInt // ERROR

    // Invariant type parameters do not create subtyping
}

fun testOut() {
    var outInt: Out<Int> = ...
    var outNumber: Out<Number> = ...
```

```
    if (random) outInt = outNumber // ERROR
    else outNumber = outInt // OK

    // Covariant type parameters create "same-way" subtyping
    //   Int <: Number => Out<Int> <: Out<Number>
    // (more specific type Out<Int> can be assigned
    //  to a less specific type Out<Number>)
}

fun testIn() {
    var inInt: In<Int> = ...
    var inNumber: In<Number> = ...

    if (random) inInt = inNumber // OK
    else inNumber = inInt // ERROR

    // Contravariant type parameters create "opposite-way" subtyping
    //   Int <: Number => In<Int> :> In<Number>
    // (more specific type In<Number> can be assigned
    //  to a less specific type In<Int>)
}
```

Use-site variance

Kotlin also supports use-site variance, by specifying the variance for type arguments. Similarly to type parameters, one can have type arguments being co-, contra- or invariant.

> Important: use-site variance cannot be used when declaring a supertype top-level type argument.

By default, all type arguments are invariant.

To specify a covariant type argument, it is marked as `out` A. To specify a contravariant type argument, it is marked as `in` A.

Kotlin prohibits contradictory combinations of declaration- and use-site variance as follows.

- It is a compile-time error to use a covariant type argument in a contravariant type parameter
- It is a compile-time error to use a contravariant type argument in a covariant type parameter

In case one cannot specify any well-formed type argument, but still needs to use a parameterized type in a type-safe way, they may use *bivariant* type

argument $\star$, which is roughly equivalent to a combination of `out kotlin.Any?` and `in kotlin.Nothing` (for further details, see subtyping).

> Note: informally, $T[\star]$ means "I can give out something very generic (`kotlin.Any?`) and cannot take in anything".

Example:

```
// A type constructor with an invariant type parameter
interface Inv<A>

fun test() {
    var invInt: Inv<Int> = ...
    var invNumber: Inv<Number> = ...
    var outInt: Inv<out Int> = ...
    var outNumber: Inv<out Number> = ...
    var inInt: Inv<in Int> = ...
    var inNumber: Inv<in Number> = ...

    when (random) {
        1 -> {
            inInt = invInt    // OK
            // T<in Int> :> T<Int>

            inInt = invNumber // OK
            // T<in Int> :> T<in Number> :> T<Number>
        }
        2 -> {
            outNumber = invInt    // OK
            // T<out Number> :> T<out Int> :> T<Int>

            outNumber = invNumber // OK
            // T<out Number> :> T<Number>
        }
        3 -> {
            invInt = inInt  // ERROR
            invInt = outInt // ERROR
            // It is invalid to assign less specific type
            // to a more specific one
            //   T<Int> <: T<in Int>
            //   T<Int> <: T<out Int>
        }
        4 -> {
            inInt = outInt    // ERROR
            inInt = outNumber // ERROR
            // types with co- and contravariant type parameters
            // are not connected by subtyping
```

```
                //   T<in Int> not(<:>) T<out Int>
            }
        }
    }
```

2.1.4 Type capturing

Type capturing (similarly to Java capture conversion) is used when instantiating type constructors; it creates *abstract captured* types based on the type information of both type parameters and arguments, which present a unified view on the resulting types and simplifies further reasoning.

The reasoning behind type capturing is closely related to variant parameterized types being a form of *bounded existential types*; e.g., `A<out T>` may be loosely considered as the following existential type: $\exists X : X <: T.A\langle X\rangle$. Informally, a bounded existential type describes a *set* of possible types, which satisfy its bound constraints. Before such a type can be used, it needs to be *opened* (or *unpacked*): existentially quantified type variables are lifted to fresh type variables with corresponding bounds. We call these type variables *captured* types.

For a given type constructor $T(F_1, \ldots, F_n) : S_1, \ldots, S_m$, its instance $T[\sigma] = T\langle\tau\rangle$ uses the following rules to create captured type K_i from the type parameter F_i and type argument A_i, at least one of which should have specified variance to create a captured type. In case both type parameter and type argument are invariant, their captured type is *equivalent* to A_i.

> Important: type capturing is **not** recursive.
>
> Note: **All** applicable rules are used to create the resulting constraint set.

- For a covariant type parameter `out` F_i, if A_i is an ill-formed type or a contravariant type argument, K_i is an ill-formed type. Otherwise, $K_i <: A_i$.
- For a contravariant type parameter `in` F_i, if A_i is an ill-formed type or a covariant type argument, K_i is an ill-formed type. Otherwise, $K_i :> A_i$.
- For a bounded type parameter $F_i <: U_i \equiv B_1 \,\&\, \ldots \,\&\, B_m$, if $\neg(A_i <: \tau U_i)$, K_i is an ill-formed type. Otherwise, $K_i <: \tau U_i$.

  > Note: captured substitution $\tau : F_1 = K_1, \ldots, F_n = K_n$ manipulates captured types.

- For a covariant type argument `out` A_i, if F_i is a contravariant type parameter, K_i is an ill-formed type. Otherwise, $K_i <: A_i$.
- For a contravariant type argument `in` A_i, if F_i is a covariant type parameter, K_i is an ill-formed type. Otherwise, $K_i :> A_i$.
- For a bivariant type argument $\star$, `kotlin.Nothing` $<: K_i <:$ `kotlin.Any?`.

- Otherwise, $K_i \equiv A_i$.

By construction, every captured type K has the following form:

$$\{L_1 <: K, \ldots, L_p <: K, K <: U_1, \ldots, K <: U_q\}$$

which can be represented as

$$L <: K <: U$$

where $L = L_1 \mid \ldots \mid L_p$ and $U = U_1 \,\&\, \ldots \,\&\, U_q$.

> Note: for implementation reasons the compiler may approximate L and/or U; for example, in the current implementation L is always approximated to be a single type.
>
> Note: as every captured type corresponds to a fresh type variable, two different captured types K_i and K_j which describe the same set of possible types (i.e., their constraint sets are equal) are *not* considered equal. However, in some cases type inference may approximate a captured type K to a concrete type $K^{\approx}$; in our case, it would be that $K_i^{\approx} \equiv K_j^{\approx}$.
>
> Examples: also show the use of type containment to establish subtyping.

```
interface Inv<T>
interface Out<out T>
interface In<in T>

interface Root<T>

interface A
interface B : A
interface C : B

fun <T> mk(): T = TODO()

interface Bounded<T : A> : Root<T>

fun test01() {

    val bounded: Bounded<in B> = mk()

    // Bounded<in B> <: Bounded<KB> where B <: KB <: A
```

```
    //   (from type capturing)
    // Bounded<KB> <: Root<KB>
    //   (from supertype relation)

    val test: Root<in C> = bounded

    // ?- Bounded<in B> <: Root<in C>
    //
    // Root<KB> <: Root<in C> where B <: KB <: A
    //   (from above facts)
    // KB ≤ in C
    //   (from subtyping for parameterized types)
    // KB ≤ in KC where C <: KC <: C
    //   (from type containment rules)
    // KB :> KC
    //   (from type containment rules)
    // (A :> KB :> B) :> (C :> KC :> C)
    //   (from subtyping for captured types)
    // B :> C
    //   (from supertype relation)
    // True

}

interface Foo<T> : Root<Out<T>>

fun test02() {

    val foo: Foo<out B> = mk()

    // Foo<out B> <: Foo<KB> where KB <: B
    //   (from type capturing)
    // Foo<KB> <: Root<Out<KB>>
    //   (from supertype relation)

    val test: Root<out Out<B>> = foo

    // ?- Foo<out B> <: Root<out Out<B>>
    //
    // Root<Out<KB>> <: Root<out Out<B>> where KB <: B
    //   (from above facts)
    // Out<KB> ≤ out Out<B>
    //   (from subtyping for parameterized types)
    // Out<KB> <: Out<B>
    //   (from type containment rules)
    // Out<out KB> <: Out<out B>
```

```
    //   (from declaration-site variance)
    // out KB ⪯ out B
    //   (from subtyping for parameterized types)
    // out KB ⪯ out KB' where B <: KB' <: B
    //   (from type containment rules)
    // KB <: KB'
    //   (from type containment rules)
    // (KB :< B) <: (B <: KB' <: B)
    //   (from subtyping for captured types)
    // B <: B
    //   (from subtyping definition)
    // True

}

interface Bar<T> : Root<Inv<T>>

fun test03() {

    val bar: Bar<out B> = mk()

    // Bar<out B> <: Bar<KB> where KB <: B
    //   (from type capturing)
    // Bar<KB> <: Root<Inv<KB>>
    //   (from supertype relation)

    val test: Root<out Inv<B>> = bar

    // ?- Bar<out B> <: Root<out Inv<B>>
    //
    // Root<Inv<KB>> <: Root<out Inv<B>> where KB <: B
    //   (from above facts)
    // Inv<KB> ⪯ out Inv<B>
    //   (from subtyping for parameterized types)
    // Inv<KB> <: Inv<B>
    //   (from type containment rules)
    // KB ⪯ B
    //   (from subtyping for parameterized types)
    // KB ⪯ KB' where B <: KB' <: B
    //   (from type containment rules)
    // KB =:= KB'
    //   (from type containment rules)
    // (Nothing <: KB :< B) =:= (B <: KB' <: B)
    //   (from subtyping for captured types)
    // False
```

```
}

interface Recursive<T : Recursive<T>>

fun <T : Recursive<T>> probe(e: Recursive<T>): T = mk()

fun test04() {
    val rec: Recursive<*> = mk()

    probe(rec)

    // ?- Recursive<*> <: Recursive<T>
    //
    // Recursive<KS> <: Recursive<KT>
    //     where Nothing <: KS <: Recursive<KS>
    //           Nothing <: KT <: Recursive<KT>
    //   (from type capturing)
    // KS ≼ KT
    //   (from subtyping for parameterized types)
    // KS =:= KT
    //   (from type containment rules)
    // True
}
```

2.1.5 Type containment

Type containment operator $\preceq$ is used to decide, whether a type A is contained in another type B denoted $A \preceq B$, for the purposes of establishing type argument subtyping.

Let A, B be concrete, well-defined non-type-parameter types, K_A, K_B be captured types.

> Important: type parameters $F_i <: U_i$ are handled as if they have been converted to well-formed captured types K_i : `kotlin.Nothing` $<: K_i <: U_i$.

$\preceq$ is defined as follows.

- $A \preceq B$ if $A \equiv B$
- $A \preceq$ `out` B if $A <: B$
- $A \preceq$ `in` B if $A :> B$
- `out` $A \preceq$ `out` B if $A <: B$
- `in` $A \preceq$ `in` B if $A :> B$

Rules for captured types follow the same structure.

- $K_A \preceq K_B$ if $K_A \equiv K_B$
- $K_A \preceq \texttt{out}\ K_B$ if $K_A <: K_B$
- $K_A \preceq \texttt{in}\ K_B$ if $K_A :> K_B$
- $\texttt{out}\ K_A \preceq \texttt{out}\ K_B$ if $K_A <: K_B$
- $\texttt{in}\ K_A \preceq \texttt{in}\ K_B$ if $K_A :> K_B$

In case we need to establish type containment between regular type A and captured type K_B, A is considered as if it is a captured type $K_A : A <: K_A <: A$.

2.1.6 Function types

Kotlin has first-order functions; e.g., it supports function types, which describe the argument and return types of its corresponding function.

A function type FT

$$\mathrm{FT}(A_1, \ldots, A_n) \to R$$

consists of

- argument types A_i
- return type R

and may be considered the following instantiation of a special type constructor FunctionN($\texttt{in}\ P_1, \ldots, \texttt{in}\ P_n, \texttt{out}\ R$) (please note the variance of type parameters)

$$\mathrm{FT}(A_1, \ldots, A_n) \to R \equiv \mathrm{FunctionN}[A_1, \ldots, A_n, R]$$

These FunctionN types follow the rules of regular type constructors and parameterized types w.r.t. subtyping.

A function type with receiver FTR

$$\mathrm{FTR}(\mathrm{RT}, A_1, \ldots, A_n) \to R$$

consists of

- receiver type RT
- argument types A_i

- return type R

From the type system's point of view, it is equivalent to the following function type

$$\mathrm{FTR}(\mathrm{RT}, A_1, \ldots, A_n) \to R \equiv \mathrm{FT}(\mathrm{RT}, A_1, \ldots, A_n) \to R$$

i.e., receiver is considered as yet another argument of its function type.

> Note: this means that, for example, these two types are equivalent w.r.t. type system
>
> - `Int.(Int) -> String`
> - `(Int, Int) -> String`
>
> However, these two types are **not** equivalent w.r.t. overload resolution, as it distinguishes between functions with and without receiver.

Furthermore, all function types FunctionN are subtypes of a general argument-agnostic type `kotlin.Function` for the purpose of unification; this subtyping relation is also used in overload resolution.

> Note: a compiler implementation may consider a function type FunctionN to have additional supertypes, if it is necessary.
>
> Example:

```
// A function of type Function1<Number, Number>
//   or (Number) -> Number
fun foo(i: Number): Number = ...

// A valid assignment w.r.t. function type variance
// Function1<in Int, out Any> :> Function1<in Number, out Number>
val fooRef: (Int) -> Any = ::foo

// A function with receiver of type Function1<Number, Number>
//   or Number.() -> Number
fun Number.bar(): Number = ...

// A valid assignment w.r.t. function type variance
// Receiver is just yet another function argument
// Function1<in Int, out Any> :> Function1<in Number, out Number>
val barRef: (Int) -> Any = Number::bar
```

Suspending function types

Kotlin supports structured concurrency in the form of coroutines via suspending functions.

For the purposes of type system, a suspending function has a *suspending* function type **suspend** $\mathrm{FT}(A_1, \ldots, A_n) \to R$, which is **unrelated by subtyping** to any non-suspending function type. This is important for overload resolution and type inference, as it directly influences the types of function values and the applicability of different functions w.r.t. overloading.

Most function values have either non-suspending or suspending function type based on their declarations. However, as lambda literals do not have any explicitly declared function type, they are considered as possibly being both non-suspending and suspending function type, with the final selection done during type inference.

Example:

```
fun foo(i: Int): String = TODO()

fun bar() {
    val fooRef: (Int) -> String = ::foo
    val fooLambda: (Int) -> String = { it.toString() }
    val suspendFooLambda: suspend (Int) -> String = { it.toString() }

    // Error: as suspending and non-suspending
    //   function types are unrelated
    // val error: suspend (Int) -> String = ::foo
    // val error: suspend (Int) -> String = fooLambda
    // val error: (Int) -> String = suspendFooLambda
}
```

2.1.7 Flexible types

Kotlin, being a multi-platform language, needs to support transparent interoperability with platform-dependent code. However, this presents a problem in that some platforms may not support null safety the way Kotlin does. To deal with this, Kotlin supports *gradual typing* in the form of flexible types.

A flexible type represents a range of possible types between type L (lower bound) and type U (upper bound), written as $(L..U)$. One should note flexible types are *non-denotable*, i.e., one cannot explicitly declare a variable with flexible type, these types are created by the type system when needed.

To represent a well-formed flexible type, $(L..U)$ should satisfy the following conditions.

- L and U are well-formed concrete types
- $L <: U$
- L and U are **not** flexible types (but may contain other flexible types as some of their type arguments)

As the name suggests, flexible types are flexible — a value of type $(L..U)$ can be used in any context, where one of the possible types between L and U is needed (for more details, see subtyping rules for flexible types). However, the actual runtime type T will be a specific type satisfying $\exists S : T <: S \wedge L <: S <: U$, thus making the substitution possibly unsafe, which is why Kotlin generates dynamic assertions, when it is impossible to prove statically the safety of flexible type use.

Dynamic type

Kotlin includes a special `dynamic` type, which in many contexts can be viewed as a flexible type (`kotlin.Nothing..kotlin.Any?`). By definition, this type represents *any* possible Kotlin type, and may be used to support interoperability with dynamically typed libraries, platforms or languages.

However, as a platform may assign special meaning to the values of `dynamic` type, it may be handled differently from the regular flexible type. These differences are to be explained in the corresponding platform-dependent sections of this specification.

Platform types

The main use cases for flexible types are *platform types* — types which the Kotlin compiler uses, when interoperating with code written for another platform (e.g., Java). In this case all types on the interoperability boundary are subject to *flexibilization* — the process of converting a platform-specific type to a Kotlin-compatible flexible type.

For further details on how *flexibilization* is done, see the corresponding JVM section.

> Important: platform types should not be confused with *multi-platform projects* — another Kotlin feature targeted at supporting platform interop.

2.1.8 Nullable types

Kotlin supports null safety by having two type universes — nullable and non-nullable. All classifier type declarations, built-in or user-defined, create non-nullable types, i.e., types which cannot hold `null` value at runtime.

To specify a nullable version of type T, one needs to use $T?$ as a type. Redundant nullability specifiers are ignored: $T?? \equiv T?$.

> Note: informally, question mark means "$T?$ may hold values of type T or value `null`"

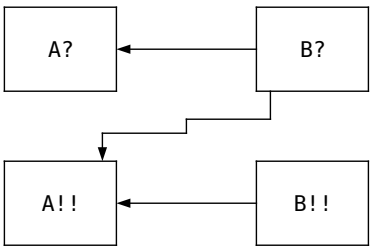
To represent a well-formed nullable type, $T?$ should satisfy the following conditions.

- T is a well-formed concrete type

  Note: if an operation is safe regardless of absence or presence of `null`, e.g., assignment of one nullable value to another, it can be used as-is for nullable types. For operations on $T?$ which may violate null safety, e.g., access to a property, one has the following null-safe options:

  1. Use safe operations
     - safe call
  2. Downcast from $T?$ to $T!!$
     - unsafe cast
     - type check combined with smart casts
     - null check combined with smart casts
     - not-null assertion operator
  3. Supply a default value to use if `null` is present
     - elvis operator

Nullability lozenge



Nullability lozenge represents valid possible subtyping relations between two nullable or non-nullable types in different combinations of their *versions*. For type T, we call $T!!$ its non-nullable version, $T?$ its nullable version.

> Note: trivial subtyping relation $A!! <: A?$ is not represented in the nullability lozenge.

Nullability lozenge may also help in establishing subtyping between two types by following its structure.

Regular (non-type-variable) types are mapped to nullability lozenge *vertices*, as for them A corresponds to $A!!$, and $A?$ corresponds to $A?$. Following the lozenge structure, for regular types A and B, as soon as we have established any valid subtyping between two versions of A and B, it implies subtyping between all other valid w.r.t. nullability lozenge combinations of versions of types A and B.

Type variable types (e.g., captured types or type parameters) are mapped to either nullability lozenge *edges* or *vertices*, as for them T corresponds to either T!! or T?, and T? corresponds to T?. Following the lozenge structure, for type variable type T (i.e., either non-nullable or nullable version) we need to consider valid subtyping for both versions T!! and T? w.r.t. nullability lozenge.

> Example: if we have `kotlin.Int`? $<:$ T?, we also have `kotlin.Int`!! $<:$ T? and `kotlin.Int`!! $<:$ T!!, meaning we can establish `kotlin.Int`!! $<: T \equiv$ `kotlin.Int` $<: T$.
>
> Example: if we have T? $<:$ `kotlin.Int`?, we also have T!! $<:$ `kotlin.Int`? and T!! $<:$ `kotlin.Int`!!, however, we can establish only T $<:$ `kotlin.Int`?, as T $<:$ `kotlin.Int` would need T? $<:$ `kotlin.Int`!! which is forbidden by the nullability lozenge.

2.1.9 Intersection types

Intersection types are special *non-denotable* types used to express the fact that a value belongs to *all* of *several* types at the same time.

Intersection type of two types A and B is denoted $A \,\&\, B$ and is equivalent to the greatest lower bound of its components $\text{GLB}(A, B)$. Thus, the normalization procedure for GLB may be used to *normalize* an intersection type.

> Note: this means intersection types are commutative and associative (following the GLB properties); e.g., $A \,\&\, B$ is the same type as $B \,\&\, A$, and $A \,\&\, (B \,\&\, C)$ is the same type as $A \,\&\, B \,\&\, C$.
>
> Note: for presentation purposes, we will henceforth order intersection type operands lexicographically based on their notation.

When needed, the compiler may *approximate* an intersection type to a *denotable concrete* type using type approximation.

One of the main uses of intersection types are smart casts.

2.1.10 Integer literal types

An integer literal type containing types $T_1, \ldots, T_N$, denoted $\text{ILT}(T_1, \ldots, T_N)$ is a special *non-denotable* type designed for integer literals. Each type $T_1, \ldots, T_N$ must be one of the built-in integer types.

Integer literal types are the types of integer literals and have special handling w.r.t. subtyping.

2.1.11 Union types

> Important: Kotlin does **not** have union types in its type system. However, they make reasoning about several type system features easier. Therefore, we decided to include a brief intro to the union types here.

Union types are special *non-denotable* types used to express the fact that a value belongs to *one* of *several* possible types.

Union type of two types A and B is denoted $A \mid B$ and is equivalent to the least upper bound of its components $\text{LUB}(A, B)$. Thus, the normalization procedure for LUB may be used to *normalize* a union type.

Moreover, as union types are *not* used in Kotlin, the compiler always *decays* a union type to a *non-union* type using type decaying.

2.2 Type contexts and scopes

The way types and scopes interoperate is very similar to how values and scopes work; this includes visibility, accessing types via qualified names or imports. This means, in many cases, type contexts are equivalent to the corresponding scopes. However, there are several important differences, which we outline below.

2.2.1 Inner and nested type contexts

Type parameters are well-formed types in the type context (scope) of their declaring type constructor, including inner type declarations. However, type context for a *nested* type declaration ND of a parent type declaration PD does **not** include the type parameters of PD.

> Note: nested type declarations cannot capture parent type parameters, as they simply create a regular type available under a nested path.

Example:

```
class Parent<T> {
    class Nested(val i: Int)

    // Can use type parameter T as a type
    // in an inner class
    inner class Inner(val t: T)

    // Cannot use type parameter T as a type
    // in a nested class
```

```
        class Error(val t: T)
    }

    fun main() {
        val nested = Parent.Nested(42)

        val inner = Parent<String>().Inner("42")
    }
```

2.3 Subtyping

Kotlin uses the classic notion of *subtyping* as *substitutability* — if S is a subtype of T (denoted as $S <: T$), values of type S can be safely used where values of type T are expected. The subtyping relation $<:$ is:

- reflexive ($A <: A$)
- *rigidly* transitive ($A <: B \land B <: C \Rightarrow A <: C$ for non-flexible types A, B and C)

Two types A and B are *equivalent* ($A \equiv B$), iff $A <: B \land B <: A$. Due to the presence of flexible types, this relation is also only *rigidly* transitive, e.g., holds only for non-flexible types (see here for more details).

2.3.1 Subtyping rules

Subtyping for non-nullable, concrete types uses the following rules.

- $\forall T : \texttt{kotlin.Nothing} <: T <: \texttt{kotlin.Any}$
- For any simple classifier type $T : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : T <: S_i$
- For any parameterized type $\widehat{T} = T\langle\tau\rangle : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : \widehat{T} <: \tau S_i$
- For any two parameterized types $\widehat{T} = T\langle\tau\rangle$ and $\widehat{T'} = T\langle\tau'\rangle$ with captured type arguments K_i and K'_i it is true that $\widehat{T} <: \widehat{T'}$ if $\forall i \in [1, n] : K_i \preceq K'_i$

Subtyping for captured types uses the following rules.

- $\forall K : \texttt{kotlin.Nothing} <: K <: \texttt{kotlin.Any?}$
- For any two captured types $L <: K <: U$ and $L' <: K' <: U'$, it is true that $K <: K'$ if $U <: L'$

Subtyping for nullable types is checked separately and uses a special set of rules which are described here.

2.3.2 Subtyping for flexible types

Flexible types (being flexible) follow a simple subtyping relation with other rigid (i.e., non-flexible) types. Let T, A, B, L, U be rigid types.

- $L <: T \Rightarrow (L..U) <: T$
- $T <: U \Rightarrow T <: (L..U)$

This captures the notion of flexible type $(L..U)$ as something which may be used in place of any type in between L and U. If we are to extend this idea to subtyping between *two* flexible types, we get the following definition.

- $L <: B \Rightarrow (L..U) <: (A..B)$

This is the most extensive definition possible, which, unfortunately, makes the type equivalence relation non-transitive. Let A, B be two *different* types, for which $A <: B$. The following relations hold:

- $A <: (A..B) \wedge (A..B) <: A \Rightarrow A \equiv (A..B)$
- $B <: (A..B) \wedge (A..B) <: B \Rightarrow B \equiv (A..B)$

However, $A \not\equiv B$.

2.3.3 Subtyping for intersection types

Intersection types introduce several new rules for subtyping. Let A, B, C, D be non-nullable types.

- $A \,\&\, B <: A$
- $A \,\&\, B <: B$
- $A <: C \wedge B <: D \Rightarrow A \,\&\, B <: C \,\&\, D$

Moreover, any type T with supertypes $S_1, \ldots, S_N$ is also a subtype of $S_1 \,\&\, \ldots \,\&\, S_N$.

2.3.4 Subtyping for integer literal types

All integer literal type are equivalent w.r.t. subtyping, meaning that for any sets $T_1, \ldots, T_K$ and $U_1, \ldots, U_N$ of built-in integer types:

- $\text{ILT}(T_1, \ldots, T_K) <: \text{ILT}(U_1, \ldots, U_N)$
- $\text{ILT}(U_1, \ldots, U_N) <: \text{ILT}(T_1, \ldots, T_K)$
- $\forall T_i \in \{T_1, \ldots, T_K\} : \text{ILT}(T_1, \ldots, T_K) <: T_i$
- $\forall T_i \in \{T_1, \ldots, T_K\} : T_i <: \text{ILT}(T_1, \ldots, T_K)$

  Note: the last two rules mean $\text{ILT}(T_1, \ldots, T_K)$ can be considered as an intersection type $T_1 \,\&\, \ldots \,\&\, T_K$ or as a union type $T_1 \mid \ldots \mid T_K$, depending on the context. Viewing ILT as intersection type allows us to use integer literals where built-in integer types are expected.

Making ILT behave as union type is needed to support cases when they appear in contravariant position.

Example:

```
interface In<in T>

fun <T> T.asIn(): In<T> = ...

fun <S> select(a: S, b: In<S>): S = ...

fun iltAsIntersection() {
    val a: Int = 42 // ILT(Byte, Short, Int, Long) <: Int

    fun foo(a: Short) {}

    foo(1377) // ILT(Short, Int, Long) <: Short
}

fun iltAsUnion() {
    val a: Short = 42

    select(a, 1337.asIn())
        // For argument a:
        //   Short <: S
        // For argument b:
        //   In<ILT(Short, Int, Long)> <: In<S> =>
        //     S <: ILT(Short, Int, Long)
        // Solution: S =:= Short
}
```

2.3.5 Subtyping for nullable types

Subtyping for two possibly nullable types A and B is defined via *two* relations, both of which must hold.

1. Regular subtyping $<:$ for types A and B using the nullability lozenge
2. Subtyping by nullability $\overset{\text{null}}{<:}$

Subtyping by nullability $\overset{\text{null}}{<:}$ for two possibly nullable types A and B uses the following rules.

1. $A!! \overset{\text{null}}{<:} B$
2. $A \overset{\text{null}}{<:} B$ if $\exists T!! : A <: T!!$
3. $A \overset{\text{null}}{<:} B?$

4. $A \overset{\text{null}}{<:} B$ if $\nexists T!! : B <: T!!$
5. $A? \overset{\text{null}}{\not<:} B$

Informally: these rules represent the following idea derived from the nullability lozenge.

> $A \overset{\text{null}}{\not<:} B$ if B is definitely non-nullable and A may be nullable or B may be non-nullable and A is definitely nullable.

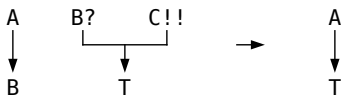
Note: these rules follow the structure of the nullability lozenge and check the absence of nullability violation $A? \overset{\text{null}}{<:} B!!$ via underapproximating it using the *supertype* relation (as we cannot enumerate the *subtype* relation for B).

Example:

```
class Foo<A, B : A?> {
    val b: B = mk()
    val bQ: B? = mk()

    // For this assignment to be well-formed,
    //   B must be a subtype of A
    // Subtyping by nullability holds per rule 4
    // Regular subtyping does not hold,
    //   as B <: A? is not enough to show B <: A
    //   (we are missing B!! <: A!!)
    val ab: A = b // ERROR

    // For this assignment to be well-formed,
    //   B? must be a subtype of A
    // Subtyping by nullability does not hold per rule 5
    val abQ: A = bQ // ERROR

    // For this assignment to be well-formed,
    //   B must be a subtype of A?
    // Subtyping by nullability holds per rule 3
    // Regular subtyping does hold,
    //   as B <: A? is enough to show B <: A?
    val aQb: A? = b // OK

    // For this assignment to be well-formed,
    //   B? must be a subtype of A?
    // Subtyping by nullability holds per rule 3
    // Regular subtyping does hold,
    //   as B <: A? is enough to show B? <: A?
    //   (taking the upper edge of the nullability lozenge)
```

```
    val aQbQ: A? = bQ // OK
}

class Bar<A, B : A> {
    val b: B = mk()
    val bQ: B? = mk()

    // For this assignment to be well-formed,
    //   B must be a subtype of A
    // Subtyping by nullability holds per rule 4
    // Regular subtyping does hold,
    //   as B <: A is enough to show B <: A
    val ab: A = b // OK

    // For this assignment to be well-formed,
    //   B? must be a subtype of A
    // Subtyping by nullability does not hold per rule 5
    val abQ: A = bQ // ERROR

    // For this assignment to be well-formed,
    //   B must be a subtype of A?
    // Subtyping by nullability holds per rule 3
    // Regular subtyping does hold,
    //   as B <: A is enough to show B <: A?
    //   (taking the upper triangle of the nullability lozenge)
    val aQb: A? = b // OK

    // For this assignment to be well-formed,
    //   B? must be a subtype of A?
    // Subtyping by nullability holds per rule 3
    // Regular subtyping does hold,
    //   as B <: A is enough to show B? <: A?
    //   (taking the upper edge of the nullability lozenge)
    val aQbQ: A? = bQ // OK
}
```

Example:



This example shows a situation, when the subtyping by nullability relation from $T <: C!!$ is used to prove $T <: A$.

2.4 Upper and lower bounds

A type U is an *upper bound* of types A and B if $A <: U$ and $B <: U$. A type L is a *lower bound* of types A and B if $L <: A$ and $L <: B$.

> Note: as the type system of Kotlin is bounded by definition (the upper bound of all types is `kotlin.Any?`, and the lower bound of all types is `kotlin.Nothing`), any two types have at least one lower bound and at least one upper bound.

2.4.1 Least upper bound

The *least upper bound* $\mathrm{LUB}(A, B)$ of types A and B is an upper bound U of A and B such that there is no other upper bound of these types which is less by subtyping relation than U.

> Note: LUB is commutative, i.e., $\mathrm{LUB}(A, B) = \mathrm{LUB}(B, A)$. This property is used in the subsequent description, e.g., other properties of LUB are defined only for a specific order of the arguments. Definitions following from commutativity of LUB are implied.

$\mathrm{LUB}(A, B)$ has the following properties, which may be used to *normalize* it. This normalization procedure, if finite, creates a *canonical* representation of LUB.

> Important: A and B are considered to be non-flexible, unless specified otherwise.

- $\mathrm{LUB}(A, A) = A$
- if $A <: B$, $\mathrm{LUB}(A, B) = B$
- if A is nullable, $\mathrm{LUB}(A, B) = \mathrm{LUB}(A!!, B!!)?$
- if $A = T\langle K_{A,1}, \ldots, K_{A,n}\rangle$ and $B = T\langle K_{B,1}, \ldots, K_{B,n}\rangle$, $\mathrm{LUB}(A, B) = T\langle \phi(\eta(K_{A,1}), \eta(K_{B,1})), \ldots, \phi(\eta(K_{A,n}), \eta(K_{B,n}))\rangle$, where $\eta(T)$ and $\phi(X, Y)$ are defined as follows:

$$\eta(K : L <: K <: U) = \{\texttt{out}\ U, \texttt{in}\ L\}$$

> Informally: in many cases, one may view $\eta(T)$ as follows.

$$\begin{aligned}
\eta(\texttt{inv}\ X) &= \{\texttt{out}\ X, \texttt{in}\ X\} \\
\eta(\texttt{out}\ X) &= \{\texttt{out}\ X, \texttt{in}\ \texttt{kotlin.Nothing}\} \\
\eta(\texttt{in}\ X) &= \{\texttt{out}\ \texttt{kotlin.Any?}, \texttt{in}\ X\} \\
\eta(\star) &= \{\texttt{out}\ \texttt{kotlin.Any?}, \texttt{in}\ \texttt{kotlin.Nothing}\}
\end{aligned}$$

$$\phi(\{\texttt{out}\ X_{out}, \texttt{in}\ X_{in}\}, \{\texttt{out}\ Y_{out}, \texttt{in}\ Y_{in}\}) = \\ \eta^{-1}(\{\texttt{out}\ \mathrm{LUB}(X_{out}, Y_{out}), \texttt{in}\ \mathrm{GLB}(X_{in}, Y_{in})\})$$

- if $A = (L_A..U_A)$ and $B = (L_B..U_B)$, $\mathrm{LUB}(A, B) = (\mathrm{LUB}(L_A, L_B)..\,\mathrm{LUB}(U_A, U_B))$
- if $A = (L_A..U_A)$ and B is not flexible, $\mathrm{LUB}(A, B) = (\mathrm{LUB}(L_A, B)..\,\mathrm{LUB}(U_A, B))$

  Important: in some cases, the least upper bound is handled as described here, from the point of view of type constraint system.

In the presence of recursively defined parameterized types, the algorithm given above is not guaranteed to terminate as there may not exist a finite representation of LUB for particular two types. The detection and handling of such situations (compile-time error or leaving the type in some kind of denormalized state) is implementation-defined.

In some situations, it is needed to construct the least upper bound for more than two types, in which case the least upper bound operator $\mathrm{LUB}(T_1, T_2, \ldots, T_N)$ is defined as $\mathrm{LUB}(T_1, \mathrm{LUB}(T_2, \ldots, T_N))$.

2.4.2 Greatest lower bound

The *greatest lower bound* $\mathrm{GLB}(A, B)$ of types A and B is a lower bound L of A and B such that there is no other lower bound of these types which is greater by subtyping relation than L.

> Note: GLB is commutative, i.e., $\mathrm{GLB}(A, B) = \mathrm{GLB}(B, A)$. This property is used in the subsequent description, e.g., other properties of GLB are defined only for a specific order of the arguments. Definitions following from commutativity of GLB are implied.

$\mathrm{GLB}(A, B)$ has the following properties, which may be used to *normalize* it. This normalization procedure, if finite, creates a *canonical* representation of GLB.

> Important: A and B are considered to be non-flexible, unless specified otherwise.

- $\mathrm{GLB}(A, A) = A$
- if $A <: B$, $\mathrm{GLB}(A, B) = A$
- if A is non-nullable, $\mathrm{GLB}(A, B) = \mathrm{GLB}(A!!, B!!)$
- if $A = T\langle K_{A,1}, \ldots, K_{A,n}\rangle$ and $B = T\langle K_{B,1}, \ldots, K_{B,n}\rangle$, $\mathrm{GLB}(A, B) = T\langle\phi(\eta(K_{A,1}), \eta(K_{B,1})), \ldots, \phi(\eta(K_{A,n}), \eta(K_{B,n}))\rangle$, where $\eta(T)$ and $\phi(X, Y)$ are defined as follows:

$$\eta(K : L <: K <: U) = \{\texttt{out}\ U, \texttt{in}\ L\}$$

Informally: in many cases, one may view $\eta(T)$ as follows.

$$\begin{aligned}\eta(\texttt{inv}\ X) &= \{\texttt{out}\ X, \texttt{in}\ X\}\\ \eta(\texttt{out}\ X) &= \{\texttt{out}\ X, \texttt{in}\ \texttt{kotlin.Nothing}\}\\ \eta(\texttt{in}\ X) &= \{\texttt{out}\ \texttt{kotlin.Any?}, \texttt{in}\ X\}\\ \eta(\star) &= \{\texttt{out}\ \texttt{kotlin.Any?}, \texttt{in}\ \texttt{kotlin.Nothing}\}\end{aligned}$$

$$\begin{aligned}&\phi(\{\texttt{out}\ X_{out}, \texttt{in}\ X_{in}\}, \{\texttt{out}\ Y_{out}, \texttt{in}\ Y_{in}\}) =\\ &\quad(\eta^{-1} \circ \Omega)(\{\texttt{out}\ \mathrm{GLB}(X_{out}, Y_{out}), \texttt{in}\ \mathrm{LUB}(X_{in}, Y_{in})\})\\ &\Omega(\{\texttt{out}\ A, \texttt{in}\ B\}) =\\ &\quad\begin{cases}\{\texttt{out}\ A, \texttt{in}\ B\} & \text{if } A :> B\\ \{\texttt{out}\ A, \texttt{in}\ \texttt{kotlin.Nothing}\} & \text{if } A <: B \wedge A \not\equiv B\end{cases}\end{aligned}$$

Note: the Ω function preserves type system consistency; $\forall A, B : A <: B \wedge A \not\equiv B$, type $T\langle\{\texttt{out}\ A, \texttt{in}\ B\}\rangle$ is the evidence of type $T\langle X\rangle : X <: A <: B <: X$, which makes the type system inconsistent. To avoid this situation, we overapproximate `in` B with `in kotlin.Nothing` when needed. Further details are available in the "Mixed-site variance" paper.

- if $A = (L_A..U_A)$ and $B = (L_B..U_B)$, $\mathrm{GLB}(A, B) = (\mathrm{GLB}(L_A, L_B)..\,\mathrm{GLB}(U_A, U_B))$
- if $A = (L_A..U_A)$ and B is not flexible, $\mathrm{GLB}(A, B) = (\mathrm{GLB}(L_A, B)..\,\mathrm{GLB}(U_A, B))$

  Important: in some cases, the greatest lower bound is handled as described here, from the point of view of type constraint system.

In the presence of recursively defined parameterized types, the algorithm given above is not guaranteed to terminate as there may not exist a finite representation of GLB for particular two types. The detection and handling of such situations (compile-time error or leaving the type in some kind of denormalized state) is implementation-defined.

In some situations, it is needed to construct the least upper bound for more than two types, in which case the least upper bound operator $\mathrm{GLB}(T_1, T_2, \ldots, T_N)$ is defined as $\mathrm{GLB}(T_1, \mathrm{GLB}(T_2, \ldots, T_N))$.

2.5 Type approximation

As we mentioned before, Kotlin type system has denotable and non-denotable types. In many cases, we need to *approximate* a non-denotable type, which appeared, for example, during type inference, into a denotable type, so that it can be used in the program. This is achieved via *type approximation*, which we describe below.

> Important: at the moment, type approximation is applied only to intersection and union types.

Type approximation function α is defined as follows.

- $\alpha(A\langle\tau_A\rangle \,\&\, B\langle\tau_B\rangle) = (\alpha{\downarrow} \circ \text{GLB})(S\langle\tau_{A\to S}\rangle, S\langle\tau_{B\to S}\rangle)$, where type S is the least single common supertype of A and B, substitution $\tau_{P\to Q}$ is the result of chain applying substitutions from type P to type $Q :> P$, $\alpha{\downarrow}$ is a function which applies type approximation function to the type arguments if needed;
- $\alpha(A\langle\tau_A\rangle \mid B\langle\tau_B\rangle) = \alpha(\delta(A\langle\tau_A\rangle \mid B\langle\tau_B\rangle))$, where δ is the type decaying function.

> Note: when we talk about the least **single** common supertype of A and B, we mean exactly that: if they have several unrelated common supertypes (e.g., several common superinterfaces), we continue going up the supertypes, until we find a single common supertype or reach `kotlin.Any?`.

2.6 Type decaying

All union types are subject to *type decaying*, when they are converted to a specific intersection type, representable within Kotlin type system.

> Important: at the moment, type decaying is applied only to union types. Note: type decaying is comparable to how *least upper bound* computation works in Java.

Type decaying function δ is defined as follows.

- $\delta(A\langle\tau_A\rangle \mid B\langle\tau_B\rangle) = \&_{S\in\mathbb{S}(A,B)}(\delta{\downarrow} \circ \text{LUB})(S\langle\tau_{A\to S}\rangle, S\langle\tau_{B\to S}\rangle)$, where substitution $\tau_{P\to Q}$ is the result of chain applying substitutions from type P to type $Q :> P$, $\delta{\downarrow}$ is a function which applies type decaying function to the type arguments if needed, $\mathbb{S}(A, B)$ is a set of most specific common supertypes of A and B.

> Note: a set of most specific common supertypes $\mathbb{S}(A, B)$ is a reduction of a set of all common supertypes $\mathbb{U}(A, B)$, which excludes all types $T \in \mathbb{U}$ such that $\exists V \in \mathbb{U} : V \neq T \wedge V <: T$.

References

1. Ross Tate. “Mixed-site variance.” FOOL, 2013.
2. Ross Tate, Alan Leung, and Sorin Lerner. “Taming wildcards in Java’s type system.” PLDI, 2011.

Chapter 3

Built-in types and their semantics

Kotlin has several built-in classifier types, which are important for the rest of this document. Some information about these types is given in the type system section, here we extend it with additional non-type-system-relevant details.

> Note: these types may have regular declarations in the standard library, but they also introduce semantics not representable via Kotlin source code.

In this section we describe these types and their semantics.

> Note: this section is not meant to be a detailed description of all types available in the standard library, for that please refer to the standard library documentation.

3.1 `kotlin.Any`

Besides being the unified supertype of all non-nullable types, `kotlin.Any` must also provide the following methods.

- `public open operator fun equals(other: Any?): Boolean`

  Returns `true` iff a value is equal to some other value. Implementations of `equals` must satisfy the properties of reflexivity (`x.equals(x)` is always true), symmetry (`x.equals(y) == y.equals(x)`), transitivity (if `x.equals(y)` is true and `y.equals(z)` is true, `x.equals(z)` is also true) and consistency (`x.equals(y)` should not change its result between multiple invocations). A non-null value also must never be considered equal to `null`, i.e., `x.equals(null)` must be `false`.

- `public open fun hashCode(): Int`

  Returns the hash code for a value. Implementations of `hashCode` must satisfy the following property: if two values are equals w.r.t. `equals`, `hashCode` must consistently produce the same result.

- `public open fun toString(): String`

  Returns a string representation of a value.

3.2 `kotlin.Nothing`

`kotlin.Nothing` is an uninhabited type, which means the evaluation of an expression with `kotlin.Nothing` type can never complete normally. Therefore, it is used to mark special situations, such as

- non-terminating expressions
- exceptional control flow
- control flow transfer

Further details about how `kotlin.Nothing` should be handled are available here and here.

3.3 `kotlin.Unit`

`kotlin.Unit` is a unit type, i.e., a type with only one value `kotlin.Unit`; all values of type `kotlin.Unit` should reference the same underlying `kotlin.Unit` object. It is somewhat similar in purpose to `void` return type in other programming languages in that it signifies an *absense of a value* (i.e. the returned type for a function returning nothing), but is different in that there is, in fact, a single value of this type.

3.4 `kotlin.Boolean`

`kotlin.Boolean` is the boolean logic type of Kotlin, representing a value which may be either `true` or `false`. It is the type of boolean literals as well as the type returned or expected by some built-in Kotlin operators.

3.5 Built-in integer types

Kotlin has several built-in classifier types, which represent signed integer numbers

of different bit size. These types are important w.r.t. type system and integer literals. Every built-in integer type `I` is a subtype of `kotlin.Comparable<I>`.

The signed integer types are the following.

- `kotlin.Int`
- `kotlin.Short`
- `kotlin.Byte`
- `kotlin.Long`

  Note: Kotlin does not have a built-in arbitrary-precision integer type.

  Note: Kotlin does not have any built-in unsigned integer types.

These types may or may not have different runtime representations, depending on the used platform and/or implementation. Consult the specific platform reference for further details.

`kotlin.Int` is the type of integer numbers that is required to be able to hold the values at least in the range from -2^{31} to $2^{31}-1$. If an arithmetic operation on `kotlin.Int` results in arithmetic overflow, the result is unspecified.

`kotlin.Short` is the type of integer numbers that is required to be able to hold the values at least in the range from -2^{15} to $2^{15}-1$. If an arithmetic operation on `kotlin.Short` results in arithmetic overflow, the result is unspecified.

`kotlin.Byte` is the type of integer numbers that is required to be able to hold the values at least in the range from -2^{7} to $2^{7}-1$. If an arithmetic operation on `kotlin.Byte` results in arithmetic overflow, the result is unspecified.

`kotlin.Long` is the type of integer numbers that is required to be able to hold the values at least in the range from -2^{63} to $2^{63}-1$. If an arithmetic operation on `kotlin.Long` results in arithmetic overflow, the result is unspecified.

> Note: by "arithmetic overflow" we assume both positive and negative integer overflows.
>
> Important: a platform implementation may specify behaviour for an arithmetic overflow.

3.5.1 Integer type widening

In overload resolution, we actually have a priority between built-in integer types which is very similar to a subtyping relation between these types; however, this priority is important only w.r.t. overload resolution and does not entail any actual subtyping between built-in integer types.

In order to introduce this priority we describe a type transformation called *widening* of integer types. Widen(T) for a built-in integer type T is defined as follows:

- Widen(`kotlin.Int`) = `kotlin.Int` & `kotlin.Short` & `kotlin.Byte` & `kotlin.Long`
- Widen(`kotlin.Short`) = `kotlin.Short` & `kotlin.Byte`
- Widen(T) = T for any other T

Informally: Widen means, for the purposes of overload resolution, `kotlin.Int` is preferred over any other built-in integer type and `kotlin.Short` is preferred to `kotlin.Byte`. Using Widen, we can reduce this priority to subtyping: T is more preferred than U if Widen(T) <: Widen(U); this scheme allows to handle built-in integer types transparently when selecting the most specific overload candidate.

For example, consider the following two functions:

```
fun foo(value: Int) = 1
fun foo(value: Short) = 2

...

foo(2)
```

As the integer literal 2 has a type that is applicable for both versions of `foo` (see Overload resolution section for details) and the types `kotlin.Int` and `kotlin.Short` are not related w.r.t. subtyping, it would not be possible to select a more specific candidate out of the two. However, if we consider Widen(`kotlin.Int`) and Widen(`kotlin.Short`) respectively as the types of `value`, first candidate becomes more specific than the second, because Widen(`kotlin.Int`) <: Widen(`kotlin.Short`).

3.6 Built-in floating point arithmetic types

There are two built-in classifier types which represent floating-point numbers: `kotlin.Float` and `kotlin.Double`. These types may or may not have different runtime representations, depending on the used platform and/or implementation. Consult the specific platform reference for further details.

`kotlin.Float` is the type of floating-point number that is able to contain all the numbers as a IEEE 754 single-precision binary floating number with the same precision. `kotlin.Float` is a subtype of `kotlin.Comparable<kotlin.Float>`.

`kotlin.Double` is the type of floating-point number that is able to contain all the numbers as a IEEE 754 double-precision binary floating number with the same precision. `kotlin.Double` is a subtype of `kotlin.Comparable<kotlin.Double>`.

Platform implementations may give additional information on how these types are represented on a particular platform.

3.7 kotlin.Char

`kotlin.Char` is the built-in classifier type which represents a single Unicode symbol in UCS-2 character encoding. It is the type of character literals.

> Important: a platform implementation may *extend* the supported character encodings, e.g., to UTF-16.

3.8 kotlin.String

`kotlin.String` is the built-in classifier type which represents a sequence of Unicode symbol in UCS-2 character encoding. It is the type of the result of string interpolation.

> Important: a platform implementation may *extend* the supported character encodings, e.g., to UTF-16.

3.9 kotlin.Enum

`kotlin.Enum<T>` is the built-in parameterized classifier type which is used as a superclass for all enum classes: every enum class `E` is implicitly a subtype of `kotlin.Enum<E>`.

`kotlin.Enum<T>` has the following characteristics.

- `kotlin.Enum<T>` is a subtype of `kotlin.Comparable<T>`

`kotlin.Enum<T>` provides the following properties.

- `public final val name: String`

  Contains the name of this enumeration constant, exactly as declared in its declaration.

- `public final val ordinal: Int`

  Contains the ordinal of this enumeration constant, i.e., its position in the declaration, starting from zero.

`kotlin.Enum<T>` provides the following member functions.

- `public override final fun compareTo(other: T): Int`

The implementation of `kotlin.Comparable`. The result of `a.compareTo(b)` for enum class instances `a` and `b` is equivalent to `a.ordinal.compareTo(b.ordinal)`.

- `public override final fun equals(other: Any?): Boolean`
- `public override final fun hashCode(): Int`

  These member functions are defined to their default behaviour: only the same entry of an enum class is equal to itself and no other object. Hash implementation is required to be consistent, but unspecified.

  > Note: the presence of these final member functions ensures the semantics of equality and comparison for the enumeration objects, as they cannot be overridden by the user.

- `protected final fun clone(): Any`

  Throws an unspecified exception.

  > Note: the `clone()` implementation throws an exception, as enumeration objects cannot be copied and on some platforms `clone` function serves for copying.

3.10 Built-in array types

`kotlin.Array<T>` is the built-in parameterized classifier type which is used to represent an indexed fixed-size collection of elements of type `T`.

It is final (i.e., cannot be inherited from) and has the following public constructor.

- `public inline constructor(size: Int, init: (Int) -> T)`

  Creates a new array with the specified size, where each element is calculated by calling the specified `init` function with the corresponding element's index. The function `init` is called for each array element sequentially starting from the first one. This constructor is special in two ways: first, it is `inline` and inline constructors are not generally allowed in Kotlin. Second, it is required for the parameter `T` to be instantiated with a runtime-available type.

`kotlin.Array<T>` provides the following methods and properties.

- `public operator fun get(index: Int): T`

  Returns the array element at the specified index. If the [index] is out of bounds of this array, throws an `IndexOutOfBoundsException`.

- `public operator fun set(index: Int, value: T): Unit`

Sets the array element at the specified index to the specified value. If the [index] is out of bounds of this array, throws an `IndexOutOfBoundsException`.

- `public val size: Int`

  Returns the array size.

- `public operator fun iterator(): Iterator<T>`

  Creates an iterator for iterating over the elements of the array.

3.10.1 Specialized array types

In addition to the general `kotlin.Array<T>` type, Kotlin also has the following specialized array types:

- `kotlin.DoubleArray` (for `kotlin.Array<kotlin.Double>`)
- `kotlin.FloatArray` (for `kotlin.Array<kotlin.Float>`)
- `kotlin.LongArray` (for `kotlin.Array<kotlin.Long>`)
- `kotlin.IntArray` (for `kotlin.Array<kotlin.Int>`)
- `kotlin.ShortArray` (for `kotlin.Array<kotlin.Short>`)
- `kotlin.ByteArray` (for `kotlin.Array<kotlin.Byte>`)
- `kotlin.CharArray` (for `kotlin.Array<kotlin.Char>`)
- `kotlin.BooleanArray` (for `kotlin.Array<kotlin.Boolean>`)

These array types are similar to the corresponding `kotlin.Array<T>` type; i.e., `kotlin.IntArray` has the same methods and properties as `kotlin.Array<Int>`, with the following changes.

- `public constructor(size: Int)`

  Creates a new array with the specified size, where each element is set to the corresponding built-in type default value.

  Note: default values are platform-specific.

- `public operator fun iterator(): {TYPE}Iterator`

  Creates a specialized iterator for iterating over the elements of the array.

3.11 Iterator types

`kotlin.Iterator<out T>` is the built-in parameterized classifier type which is used to represent a sequence of elements of type `T`, allowing for sequential access to these elements.

It provides the following methods.

- `public operator fun next(): T`

Returns the next element in the sequence.

- `public operator fun hasNext(): Boolean`

  Returns `true` if the sequence has more elements.

3.11.1 Specialized iterator types

Specialized iterator types are iterator types used for specialized array types. They inherit `kotlin.Iterator<out T>` for their type (i.e., `kotlin.CharIterator` inherits `kotlin.Iterator<Char>`) and provide the following methods.

- `public operator fun next{TYPE}(): {TYPE}`

  Returns the next element in the sequence as a specific type.

  > Note: this additional method allows the compiler and/or developer to avoid unneeded platform-specific boxing/unboxing conversions.

3.12 `kotlin.Throwable`

`kotlin.Throwable` is the built-in classifier type that is the base type of all exception types. Any value that is used in a `throw` expression must have a static type that is a subtype of `kotlin.Throwable`. Any type that is used in a `catch` part of the `try` expression must be a subtype of (or equal to) `kotlin.Throwable`.

It provides at least the following properties:

- `public val message: String?`

  An optional message depicting the cause of the throw.

- `public val cause: Throwable?`

  An optional other value of type `kotlin.Throwable` allowing for nested throwables to be constructed.

Other members may exist, please refer to the standard library documentation for details. No subtype of `kotlin.Throwable` is allowed to have type parameters. Declaring such a type is a compile-time error.

3.13 `kotlin.Comparable`

`kotlin.Comparable<in T>` is a built-in parameterized type which represents values that may be compared for total ordering. It provides the following member function:

```
public operator fun compareTo(other: T): Int
```

This function is used to implement comparison operators through overloadable operator convention for standard library classes.

> Note: a type is not required to be a subtype of `kotlin.Comparable` in order to implement total ordering operations

3.14 kotlin.Function

`kotlin.Function<out R>` is the base classifier type of all function types. See the relevant section for details.

3.15 Built-in annotation types

Kotlin has a number of built-in annotation types, which are covered in more detail here.

3.16 Reflection support builtin types

3.16.1 kotlin.reflect.KClass

`kotlin.reflect.KClass<T: Any>` is the class used to represent runtime type information for runtime-available classifier types. It is also used in platform-specific reflection facilities.

This is the type of class literals. This type is required to introduce `equals` and `hashCode` member function implementations (see `kotlin.Any`) that allow for comparison and hashing of runtime type information, e.g., that class literals are equal if they represent the same runtime type and not equal otherwise. Platform definitions, as well as particular implementations, may introduce additional members for this type.

3.16.2 kotlin.reflect.KCallable

`kotlin.reflect.KCallable<out R>` is the class used to represent runtime information for callables (i.e. properties and functions). It is mainly used as base type for other types described in this section. It provides at least the following property:

```
public val name: String
```

This property contains the name of the callable. Other members or base types for this class may be provided by platform and/or implementation.

3.16.3 `kotlin.reflect.KProperty`

`kotlin.reflect.KProperty<out R>` is the class used to represent runtime information for properties. It is the base type of property references. This type is used in property delegation. `kotlin.reflect.KProperty<R>` is a subtype of `kotlin.reflect.KCallable<R>`. Other members or base types for this class may be provided by platform and/or implementation.

3.16.4 `kotlin.reflect.KFunction`

`kotlin.reflect.KFunction<out R>` is the class used to represent runtime information for functions. It is the base type of function references. `kotlin.reflect.KFunction<R>` is a subtype of `kotlin.reflect.KCallable<R>` and `kotlin.Function<R>`. Other members or base types for this class may be provided by platform and/or implementation.

Chapter 4

Declarations

Glossary

Entity
: Distinguishable part of a program

Identifier
: Name of a program entity

Path
: Sequence of identifiers which references a program entity in a given scope

Introduction

Declarations in Kotlin are used to introduce entities (values, types, etc.); most declarations are *named*, i.e. they also assign an identifier to their own entity, however, some declarations may be *anonymous*.

Every declaration is accessible in a particular *scope*, which is dependent both on where the declaration is located and on the declaration itself.

4.1 Classifier declaration

***classDeclaration*:**

[*modifiers*]
('`class`' | (['`fun`' {*NL*}] '`interface`'))
{*NL*}
simpleIdentifier

[{*NL*} *typeParameters*]
[{*NL*} *primaryConstructor*]
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *typeConstraints*]
[({*NL*} *classBody*) | ({*NL*} *enumClassBody*)]

***objectDeclaration*:**
[*modifiers*]
`'object'`
{*NL*}
simpleIdentifier
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

Classifier declarations introduce new types to the program, of the forms described here. There are three kinds of classifier declarations:

- class declarations;
- interface declarations;
- object declarations.

  Important: object literals are similar to object declarations and are considered to be anonymous classifier declarations, despite being expressions.

4.1.1 Class declaration

A simple class declaration consists of the following parts.

- Name c;
- Optional primary constructor declaration $ptor$;
- Optional supertype specifiers $S_1, \ldots, S_s$;
- Optional body b, which may include the following:
  - secondary constructor declarations $stor_1, \ldots, stor_c$;
  - instance initialization blocks $init_1, \ldots, init_i$;
  - property declarations $prop_1, \ldots, prop_p$;
  - function declarations $md_1, \ldots, md_m$;
  - companion object declaration $companionObj$;
  - nested classifier declarations $nested$.

and creates a simple classifier type $c : S_1, \ldots, S_s$.

Supertype specifiers are used to create inheritance relation between the declared type and the specified supertype. You can use classes and interfaces as supertypes, but not objects or inner classes.

> Note: if supertype specifiers are absent, the declared type is considered to be implicitly derived from `kotlin.Any`.

It is allowed to inherit from a single class only, i.e., multiple class inheritance is not supported. Multiple interface inheritance is allowed.

Instance initialization block describes a block of code which should be executed during object creation.

Property and function declarations in the class body introduce their respective entities in this class' scope, meaning they are available only on an entity of the corresponding class.

Companion object declaration `companion object CO { ... }` for class `C` introduces an object, which is available under this class' name or under the path `C.CO`. Companion object name may be omitted, in which case it is considered to be equal to `Companion`.

Nested classifier declarations introduce new classifiers, available under this class' name. Further details are available here.

A parameterized class declaration, in addition to what constitutes a simple class declaration, also has a type parameter list $T_1, \ldots, T_m$ and extends the rules for a simple class declaration w.r.t. this type parameter list. Further details are described here.

Examples:

```
// An open class with no supertypes
//
open class Base

// A class inherited from `Base`
//
// Has a single read-only property `i`
//   declared in its primary constructor
//
class B(val i: Int) : Base()

// An open class with no superclasses
//
// Has a single read-only property `i`
//   declared in its body
//
// Initial value for the property is calculated
//   in the init block
//
open class C(arg: Int) {
    val i: Int

    init {
        i = arg * arg
```

```
    }
}

// A class inherited from `C`
//   Does not have a primary constructor,
//   thus does not need to invoke the supertype constructor
//
// The secondary constructor delegates to the supertype constructor
class D : C {
    constructor(s: String) : super(s.toInt())
}

// An open class inherited from `Base`
//
// Has a companion object with a mutable property `name`
class E : Base() {
    companion object /* Companion */ {
        var name = "I am a companion object of E!"
    }
}
```

Example:

```
class Pair(val a: Int, val b: Int) : Comparable<Pair> {

    fun swap(): Pair = Pair(b, a)

    override fun compareTo(other: Pair): Int {
        val f = a.compareTo(other.a)
        if (f != 0) return f
        return b.compareTo(other.b)
    }

    companion object {
        fun duplet(a: Int) = Pair(a, a)
    }
}
```

Constructor declaration

There are two types of class constructors in Kotlin: primary and secondary.

A primary constructor is a concise way of describing class properties together with constructor parameters, and has the following form

$$ptor : (p_1, \ldots, p_n)$$

where each of p_i may be one of the following:

- regular constructor parameter *name* : *type*;
- read-only property constructor parameter `val` *name* : *type*;
- mutable property constructor parameter `var` *name* : *type*.

Property constructor parameters, together with being regular constructor parameters, also declare class properties of the same name and type.

> Important: if a property constructor parameter with type T is specified as `vararg`, its corresponding class property type is the result of *array type specialization* of type `Array<out` T`>`.

One can consider primary constructor parameters to have the following syntactic expansion.

```
class Foo(i: Int, vararg val d: Double, var s: String) : Super(i, d, s) {}

class Foo(i: Int, vararg d_: Double, s_: String) : Super(i, d_, s_) {
  val d = d_
  var s = s_
}
```

When accessing property constructor parameters inside the class body, one works with their corresponding properties; however, when accessing them in the supertype specifier list (e.g., as an argument to a superclass constructor invocation), we see them as actual parameters, which cannot be changed.

If a class declaration has a primary constructor and also includes a class supertype specifier, that specifier must represent a valid invocation of the supertype constructor.

A secondary constructor describes an alternative way of creating a class instance and has only regular constructor parameters.

If a class has a primary constructor, any secondary constructor must delegate to either the primary constructor or to another secondary constructor via `this(...)`.

If a class does not have a primary constructor, its secondary constructors must delegate to either the superclass constructor via `super(...)` (if the superclass is present in the supertype specifier list) or to another secondary constructor via `this(...)`. If the only superclass is `kotlin.Any`, delegation is optional.

In all cases, it is forbidden if two or more secondary constructors form a delegation loop.

Class constructors (both primary and secondary) may have variable-argument parameters and default parameter values, just as regular functions. Please refer to the function declaration reference for details.

If a class does not have neither primary, nor secondary constructors, it is assumed to implicitly have a default parameterless primary constructor. This also means that, if a class declaration includes a class supertype specifier, that specifier must represent a valid invocation of the supertype constructor.

Examples:

```
open class Base

class POKO : Base() {}

class NotQuitePOKO : Base {
    constructor() : super() {}
}

class Primary(val s: String) : Base() {
    constructor(i: Int) : this(i.toString()) {}

    constructor(d: Double) : this(d.toInt()) {}

    // Error, has primary ctor,
    //   needs to delegate to primary or secondary ctor
    // constructor() : super() {}
}

class Secondary : Base {
    constructor(i: Int) : super() {}

    constructor(s: String) : this(s.toInt()) {}

    // Ok, no primary ctor,
    //   can delegate to `super(...)`
    constructor() : super() {}
}
```

Nested and inner classifiers

If a classifier declaration ND is *nested* in another classifier declaration PD, it creates a nested classifier type — a classifier type available under the path PD . ND. In all other aspects, nested classifiers are equivalent to regular ones.

Inner classes are a special kind of nested classifiers, which introduce types of objects associated (linked) with other (parent) objects. An inner class declaration ID nested in another classifier declaration PD may reference an *object* of type ID associated with it.

This association happens when instantiating an object of type ID, as its con-

structor may be invoked only when a receiver of type PD is available, and this receiver becomes associated with the new instantiated object of type ID.

Inner classes cannot be declared in interface declarations, as interfaces cannot be instantiated.

Inner classes cannot be declared in a statement scope, as such scope does not have an object to associate the inner class with.

Inner classes cannot be declared in object declarations, as object declarations also create a single named value of their type, which makes additional association unnecessary.

> Note: for information on how type parameters of parent and nested / inner classifiers interoperate, we delegate you to the type system section of the specification.
>
> Note: unlike object declarations, in object literals only inner classes are allowed, as types of object literals are anonymous, making their nested classifiers available only through explicit receiver, effectively forcing them to be inner.
>
> Examples:

```
interface Quz {
    interface Bar
    class Nested
    // Error: no parent object to reference,
    //   as interfaces cannot be instantiated
    // inner class Inner
}

class Foo {
    interface Bar
    class Nested
    inner class Inner
}

object Single {
    interface Bar
    class Nested
    // Error: value of type Single is available as-is,
    //   no reason to make an inner class
    // inner class Inner
}

fun foo() {
    // Error: interfaces cannot be local
    // interface Bar
```

```
        class Nested

        // Error: inner classes cannot be local
        // inner class Inner
    }

    fun test() {
        val fooV = Foo()

        Quz.Nested()
        Foo.Nested()
        fooV.Inner()

        Single.Nested()

        val anon = object {
            // Error: cannot reference <anon>.Bar
            // interface Bar
            // Error: cannot reference <anon>.Nested
            // class Nested
            inner class Inner
        }

        anon.Inner()
    }
```

Inheritance delegation

In a classifier (an object or a class) declaration C, any supertype I inheritance may be *delegated to* an arbitrary value v if:

- The supertype I is an interface type;
- v has type T such that $T <: I$.

The inheritance delegation uses a syntax similar to property delegation using the `by` keyword, but is specified in the classifier declaration header and is a very different concept. If inherited using delegation, each method M of I (whether they have a default implementation or not) is delegated to the corresponding method of v as if it was overridden in C with all the parameter values directly passed to the corresponding method in v, unless the body of C itself has a suitable override of M (see the method overriding section).

The particular means on how v is stored inside the classifier object is platform-defined.

Due to the initialization order of a classifier object, the expression used to

construct v can not access any of the classifier object properties or methods excluding the parameters of the primary constructor.

Example:

```
interface I {
    fun foo(value: Int): Double
    val bar: Long
}
interface J : I {
    fun fee(): Int
}

class C(delegatee: I): I by delegatee
```

is expanded to

```
interface I {
    fun foo(value: Int): Double
    val bar: Long
}
interface J : I {
    fun fee(): Int
}

class C(delegatee: I): I {
    val I$delegate = delegate

    override fun foo(value: Int): Double = I$delegate.foo(value)
    override val bar: Long
        get() = I$delegate.bar
}
```

Please note that the expression used as delegate is accessed exactly once when creating the object, e.g. if the delegate expression contains a mutable property access, this mutable property is accessed once during object construction and its subsequent changes do not affect the delegated interface functions. See classifier initialization section for details on the evaluation order of classifier initialization entities.

For example (assuming interface `I` from the previous example is defined):

```
var mut = object: J {...}

class D: I by mut // D delegates I to mutable property
```

is expanded to

```
var mut = object: J {...}
```

```
class D: I {
    val I$delegate = mut // mut is accessed only once

    override fun foo(value: Int): Double = I$delegate.foo(value)
    override val bar: Long
        get() = I$delegate.bar
}

mut = x1
val d1 = D() // d1 methods are delegated to x1
mut = x2
val d2 = D() // d2 methods are delegated to x2
// but d1 methods are still delegated to x1
```

Abstract classes

A class declaration can be marked `abstract`. Such classes *cannot* be instantiated directly; they are used as superclasses for other classes or objects.

Abstract classes may contain one or more abstract members: members without implementation, which should be implemented in a subtype of this abstract class.

4.1.2 Data class declaration

A data class *dataClass* is a special kind of class, which represents a product type constructed from a number of data properties $(dp_1, \ldots, dp_m)$, described in its primary constructor. Non-property constructor parameters are not allowed in the primary constructor of a data class. As such, data classes allow Kotlin to reduce the boilerplate and generate a number of additional data-relevant functions.

- `equals() / hashCode() / toString()` functions compliant with their contracts:
  - `equals(that)` returns true iff:
    - `that` has the same runtime type as `this`;
    - `this.prop == that.prop` returns `true` for every data property `prop`;
  - `hashCode()` returns the same numbers for objects `A` and `B` if they are equal w.r.t. the generated `equals`;
  - `toString` returns a string representations which is guaranteed to include the class name along with all the data properties' string representations.
- A `copy()` function for shallow object copying with the following properties:
  - It has the same number of parameters as the primary constructor with the same names and types;

  - It calls the primary constructor with the corresponding parameters at the corresponding positions;
  - It has defaults for all the parameters defaulting to the value of the corresponding property in `this` object.
- A number of `componentN()` functions for destructuring declaration:
  - For the data property at position N (**starting from 1**), the generated `component`N function has the same type as this property and returns the value of this property;
  - It has an `operator` modifier, allowing it to be used in destructuring declarations;
  - The number of these functions is the same as the number of data properties.

All these functions consider only data properties $\{dp_i\}$; e.g., your data class may include regular property declarations in its body, however, they will *not* be considered in the `equals()` implementation or have a `componentN()` generated for them.

There are several rules as to how these generated functions may be explicified or inherited.

> Note: a generated function is explicified, if its implementation (with matching function signature) is provided explicitly in the body of the data class. A generated function is inherited, if its implementation (with matching function signature) is taken from a supertype of the data class.

The declarations of `equals`, `hashCode` and `toString` may be explicified similarly to how overriding works in normal classes. If a correct explicit implementation is available, no function is generated. Other functions (`copy`, `componentN`) **cannot** be explicified.

The declarations of `equals`, `hashCode` and `toString` may be inherited from the base class, if it provides a `final` version with a matching signature. If a correct inherited implementation is available, no function is generated. Other functions (`copy`, `componentN`) **cannot** be inherited.

In addition, for every generated function, if any of the base types provide an open function with a matching signature, it is automatically overridden by the generated function as if it was generated with an `override` modifier.

> Note: data classes or their supertypes may also have functions which have a matching name and/or signature with one of the generated functions. As expected, these cases result in either override or overload conflicts the same way they would with a normal class declaration, or they create two separate functions which follow the rules of overloading.

Data classes have the following restrictions:

- Data classes are closed and cannot be inherited from;
- Data classes must have a primary constructor with property constructor parameters only, which become data properties for the data class;
- There must be at least one data property in the primary constructor;
- Data properties cannot be specified as `vararg` constructor arguments.

For example, the following data class declaration

```
data class DC(val x: Int, val y: Double)
```

is equivalent to

```
class DC(val x: Int, val y: Double) {
    override fun equals(other: Any?): Boolean {
         if(other !is DC) return false
         return x == other.x && y == other.y
    }

    override fun hashCode(): Int = x.hashCode() + 31 * y.hashCode()

    override fun toString(): String = "DC(x=$x,y=$y)"

    operator fun component1(): Int = x

    operator fun component2(): Double = y

    fun copy(x: Int = this.x, y: Double = this.y): DC = DC(x, y)
}
```

The following data class declaration

```
data class DC(val x: Int) {
    override fun equals(other: Any?) = false
    override fun toString(): String = super.toString()
```

may be equivalent to

```
class DC(val x: Int) {
    override fun equals(other: Any?) = false

    override fun hashCode(): Int = x.hashCode()

    override fun toString(): String = super.toString()

    operator fun component1(): Int = x

    fun copy(x: Int = this.x): DC = DC(x)
}
```

(note how `equals` and `toString` implementations are explicified in the second declaration)

Disclaimer: the implementations of these methods given in this examples are not guaranteed to exactly match the ones generated by kotlin compiler, please refer to the descriptions of these methods above for guarantees

4.1.3 Enum class declaration

Enum class E is a special kind of class with the following properties:

- It has a number of predefined values that are declared in the class itself (*enum entries*);
- No other values of this class can be constructed;
- It implicitly inherits the built-in class `kotlin.Enum<E>` (and cannot have any other base classes);
- It is implicitly final and cannot be inherited from;
- It cannot have type parameters of any kind;
- It has special syntax to accommodate for the properties described above.

Enum class body uses special kind of syntax (see grammar) to declare enum entries in addition to all other declarations inside the class body. Enum entries have their own bodies that may contain their own declarations, similar to object declarations.

Note: an enum class can have zero enum entries. This makes objects of this class impossible to construct.

Every enum entry of class `E` implicitly overrides members of `kotlin.Enum<E>` in the following way:

- `public final val name: String`

  defined to be the same as the name of the entry as declared in code;

- `public final val ordinal: Int`

  defined to be the ordinal of the entry, e.g. the position of this entry in the list of entries, starting with 0;

- `public override final fun compareTo(other: E): Int`

  (a member of `kotlin.Comparable<E>`) defined by default to compare entries by their ordinals, but may be overridden to have different behaviour both in the enum class declaration and in entry declarations;

- `public override fun toString(): String`

(a member of `kotlin.Any`) defined by default to return the entry name, but may be overridden to have different behaviour both in the enum class declaration and in entry declarations.

In addition to these, every enum class type `E` has the following **static** member functions declared implicitly:

- `public final static fun valueOf(value: String): E`

  returning an object corresponding to the entry with the name equal to `value` parameter of the call or throws an exception otherwise;

- `public final static fun values(): kotlin.Array<E>`

  returning an array of all possible enum values in the order they are declared. Every invocation of this function returns a new array to disallow changing its contents.

  Important: `static` is not a valid Kotlin keyword and is only used here for clarity

  Note: these static member functions are handled differently by the overload resolution.

  Note: Kotlin standard library introduces another function to access all enum values for a specific enum class called `kotlin.enumValues<T>`. Please refer to the standard library documentation for details.

  Example:

```
enum class State { LIQUID, SOLID, GAS }

...
State.SOLID.name // "SOLID"
State.SOLID.ordinal // 1
State.GAS > State.LIQUID // true
State.SOLID.toString() // "SOLID"
State.valueOf("SOLID") // State.SOLID
State.valueOf("Foo") // throws exception
State.values() // arrayOf(State.LIQUID, State.SOLID, State.GAS)

...

// enum class can have additional declarations that may be overriden in its values
enum class Direction(val symbol: Char) {
    UP('^') {
        override val opposite: Direction
            get() = DOWN
    },
    DOWN('v') {
        override val opposite: Direction
```

```
            get() = UP
    },
    LEFT('<') {
        override val opposite: Direction
            get() = RIGHT
    },
    RIGHT('>') {
        override val opposite: Direction
            get() = LEFT
    };
    abstract val opposite: Direction
}
```

4.1.4 Annotation class declaration

Annotations class is a special kind of class that is used to declare annotations. Annotation classes have the following properties:

- They cannot have any secondary constructors;
- All the primary constructor parameters must use the property syntax;
- They implicitly implement `kotlin.Annotation` interface (and cannot implement additional interfaces);
- They cannot have any specified base classes;
- They are implicitly closed and cannot be inherited from;
- They may not have any member functions, properties not declared in the primary constructor or any overriding declarations;
- They cannot have companion objects;
- They cannot have nested classes;
- They cannot have type parameters;
- The types of primary constructor parameters are limited to:
  - `kotlin.String`;
  - `kotlin.KClass`;
  - Built-in number types;
  - Other annotation types;
  - Arrays of any other allowed type.

Annotation classes cannot be constructed directly unless passed as arguments to other annotations, but their primary constructors are used when specifying code annotations for other entities.

Examples:

```
// a couple annotation classes
annotation class Super(val x: Int, val f: Float = 3.14f)
annotation class Duper(val supers: Array<Super>)

// the same classes used as annotations
```

```
@Duper(arrayOf(Super(2, 3.1f), Super(3)))
class SuperClass {
    @Super(4)
    val x = 3
}

// annotation class without parameters
annotation class Transmogrifiable

@Transmogrifiable
fun f(): Int = TODO()

// variable argument properties are supported
annotation class WithTypes(vararg val classes: KClass<out Annotation>)

@WithTypes(Super::class, Transmogrifiable::class)
val x = 4
```

4.1.5 Value class declaration

> Note: as of Kotlin 1.5.0, user-defined value classes are an experimental feature. There is, however, a number of value classes in Kotlin standard library.

A class may be declared a **value** class by using `inline` or `value` modifier in its declaration. Value classes must adhere to the following limitations:

- Value classes are closed and cannot be inherited from;
- Value classes cannot be `inner`, `data` or `enum` classes;
- Value classes must have a primary constructor with a single property constructor parameter, which is the data property of the class;
- This property cannot be specified as `vararg` constructor argument;
- This property must be declared `public`;
- This property must be of a runtime-available type;
- They must not override `equals` and `hashCode` member functions of `kotlin.Any`;
- They must not have any base classes besides `kotlin.Any`;
- No other properties of this class may have backing fields.

> Note: `inline` modifier for value classes is supported as a legacy feature for compatibility with Kotlin 1.4 experimental inline classes and will be deprecated in the future.

Value classes implicitly override `equals` and `hashCode` member functions of `kotlin.Any` by delegating them to their only data property. Unless `toString` is overriden by the value class definition, it is also implicitly overriden by delegating to the data property. In addition to these, an value class is allowed

by the implementation to be **inlined** where applicable, so that its data property is operated on instead. This also means that the property may be boxed back to the value class by using its primary constructor at any time if the compiler decides it is the right thing to do.

Due to these restrictions, it is highly discouraged to use value classes with the reference equality operators.

> Note: in the future versions of Kotlin, value classes may be allowed to have more than one data property.

4.1.6 Interface declaration

Interfaces differ from classes in that they cannot be directly instantiated in the program, they are meant as a way of describing a contract which should be satisfied by the interface's subtypes. In other aspects they are similar to classes, therefore we shall specify their declarations by specifying their differences from class declarations.

- An interface can be declared only in a declaration scope;
  - Additionally, an interface cannot be declared in an [object literal];
- An interface cannot have a class as its supertype;
- An interface cannot have a constructor;
- Interface properties cannot have initializers or backing fields;
- Interface properties cannot be delegated;
- An interface cannot have inner classes;
- An interface and all its members are implicitly open;
- All interface member properties and functions are implicitly public;
  - Trying to declare a non-public member property or function in an interface is an compile-time error.

Functional interface declaration

A *functional interface* is an interface with a **single** abstract function and no other abstract properties or functions.

A function interface declaration is marked as `fun interface`. It has the following additional restrictions compared to regular interface declarations.

- A functional interface can have only one abstract member function, which must be non-parameterized;
- A functional interface cannot have any abstract member properties;

A functional interface has an associated function type, which is the same as the function type of its single abstract member function.

> Important: the associated function type of a functional interface is different from the type of said functional interface.

If one needs an object of a functional interface type, they can use the regular ways of implementing an interface, either via an anonymous object declaration or as a complete class. However, as functional interface essentially represents a single function, Kotlin supports the following additional ways of providing a functional interface implementation from function values (expressions with function type).

- If a lambda literal `L` is preceded with a functional interface name `T`, and the type of `L` is a subtype of the associated function type of `T`, this expression creates an instance of `T` with lambda literal `L` used as its abstract member function implementation.

  Example:

  ```
  fun interface FI {
      fun bar(s: Int): Int
  }

  fun foo() {
      val fi = FI { it }
      val fi2 = FI { s: Int -> s + 42 }
      val fi3 = FI { s: Number -> s.toInt() }
  }
  ```

- If an expression `L` is used as an argument of functional type `T` in a function call, and the type of `L` is a subtype of the associated function type of `T`, this argument is considered as an instance of `T` with expression `L` used as its abstract member function implementation.

  Example:

  ```
  fun interface FI {
      fun bar(s: Int): Int
  }

  fun doIt(fi: FI) {}

  fun foo() {
      doIt { it }
      doIt { s: Int -> s + 42 }
      doIt { s: Number -> s.toInt() }

      doIt(fun(s): Int { return s; })

      val l = { s: Number -> s.toInt() }

      doIt(l)
  }
  ```

Informally: this feature is known as "Single Abstract Method" (SAM) conversion.

Note: in Kotlin version 1.3 and earlier, SAM conversion was not available for Kotlin functional interfaces. It was, however, available on Kotlin/JVM for Java functional interfaces.

4.1.7 Object declaration

Object declarations are similar to class declaration in that they introduce a new classifier type, but, unlike class or interface declarations, they also introduce a value of this type in the same declaration. No other values of this type may be declared, making object a single existing value of its type.

Note: This is similar to *singleton pattern* common to object-oriented programming in introducing a type which includes a single global value.

Similarly to interfaces, we shall specify object declarations by highlighting their differences from class declarations.

- An object can only be declared in a declaration scope;
  - Additionally, an object cannot be declared in an [object literal];
- An object type cannot be used as a supertype for other types;
- An object cannot have an explicit primary or secondary constructor;
- An object cannot have a companion object;
- An object cannot have inner classes;
- An object cannot be parameterized, i.e., cannot have type parameters.

Note: an object is assumed to implicitly have a default parameterless primary constructor.

Note: this section is about declaration of *named* objects. Kotlin also has a concept of *anonymous* objects, or object literals, which are similar to their named counterparts, but are expressions rather than declarations and, as such, are described in the corresponding section.

4.1.8 Local class declaration

A class (but not an interface or an object) may be declared *locally* inside a statement scope (namely, inside a function). Such declarations are similar to object literals in that they may capture values available in the scope they are declared in:

```
fun foo() {
    val x = 2
    class Local {
```

```
        val y = x
    }
    Local().y // 2
}
```

Enum classes and annotation classes cannot be declared locally.

4.1.9 Classifier initialization

When creating a class or object instance via one of its constructors *ctor*, it is initialized in a particular order, which we describe here.

A primary *pctor* or secondary constructor *ctor* has a corresponding superclass constructor *sctor* defined as follows.

- For primary constructor *pctor*, a corresponding superclass constructor *sctor* is the one from the supertype specifier list;
- For secondary constructor *ctor*, a corresponding supertype constructor *sctor* is the one ending the constructor delegation chain of *ctor*;
- If an explicit superclass constructor is not available, `Any()` is implicitly used.

When a classifier type is initialized using a particular secondary constructor *ctor* delegated to primary constructor *pctor* which, in turn, is delegated to the corresponding superclass constructor *sctor*, the following happens, in this *initialization order*:

- *pctor* is invoked using the specified parameters, initializing all the properties declared by its property parameters *in the order of appearance in the constructor declaration*;
- The superclass object is initialized as if created by invoking *sctor* with the specified parameters;
- Interface delegation expressions are invoked and the result of each is stored in the object to allow for interface delegation, *in the order of appearance of delegation declarations in the supertype specifier list*;
- Each property initialization code as well as the initialization blocks in the class body are invoked *in the order of appearance in the class body*;
- *ctor* body is invoked using the specified parameters.

  Note: this means that if an `init`-block appears between two property declarations in the class body, its body is invoked between the initialization code of these two properties.

The initialization order stays the same if any of the entities involved are omitted, in which case the corresponding step is also omitted (e.g., if the object is created using the primary constructor, the body of the secondary one is not invoked).

If any step in the initialization order creates a loop, it results in unspecified behaviour.

If any of the properties are accessed before they are initialized w.r.t initialization order (e.g., if a method called in an initialization block accesses a property declared *after* the initialization block), the value of the property is unspecified. It stays unspecified even after the "proper" initialization is performed.

> Note: this can also happen if a property is captured in a lambda expression used in some way during subsequent initialization steps.

Examples:

```
open class Base(val v: Any?) {
    init {
        println("2: $this")
    }
}

class Init(val a: Number /* (1) */) : Base(0xC0FFEE) /* (2) */ {

    init {
        println("3: $this") /* (3) */
    }

    constructor(v: Int) : this(v as Number) {
        println("10: $this") /* (10) */
    }

    val b: String = a.toString() /* (4) */

    init {
        println("5: $this") /* (5) */
    }

    var c: Any? = "b is $b" /* (6) */

    init {
        println("7: $this") /* (7) */
    }

    val d: Double = 42.0 /* (8) */

    init {
        println("9: $this") /* (9) */
    }

    override fun toString(): String {
        return "Init(a=$a, b='$b', c=$c, d=$d)"
    }
```

```
}

fun main() {
    Init(5)
    // 2: Init(a=null, b='null', c=null, d=0.0)
    // 3: Init(a=5, b='null', c=null, d=0.0)
    // 5: Init(a=5, b='5', c=null, d=0.0)
    // 7: Init(a=5, b='5', c=b is 5, d=0.0)
    // 9: Init(a=5, b='5', c=b is 5, d=42.0)
    // 10: Init(a=5, b='5', c=b is 5, d=42.0)

    // Here we can see how the undefined values for
    //   uninitialized properties may leak outside
}
```

4.2 Function declaration

***functionDeclaration*:**
[*modifiers*]
`'fun'`
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} `'.'`]
{*NL*}
simpleIdentifier
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionBody*:**
block
| (`'='` {*NL*} *expression*)

Function declarations assign names to functions — blocks of code which may be called by passing them a number of arguments. Functions have special *function types* which are covered in more detail here.

A simple function declaration consists of four main parts:

- Name f;
- Parameter list $(p_1 : P_1[= v_1], \ldots, p_n : P_n[= v_n])$;
- Return type R;
- Body b.

and has a function type $f : (p_1 : P_1, \ldots, p_n : P_n) \to R$.

Parameter list $(p_1 : P_1[= v_1], \ldots, p_n : P_n[= v_n])$ describes function parameters, i.e. inputs needed to execute the declared function. Each parameter $p_i : P_i = v_i$ introduces p_i as a name of value with type P_i available inside function body b; therefore, parameters are final and cannot be changed inside the function. A function may have zero or more parameters.

A parameter may include a default value v_i, which is used if the corresponding argument is not specified in function invocation; v_i must be an expression which evaluates to type $V <: P_i$.

Return type R, if omitted, is calculated as follows.

- If function body b is present in the expression form and it may be inferred to have a valid type $B : B \not\equiv$ `kotlin.Nothing`, $R \equiv B$.
- If function body b is present in the block form, $R \equiv$ `kotlin.Unit`.

In other cases return type R cannot be omitted and must be specified explicitly.

> As type `kotlin.Nothing` has a special meaning in Kotlin type system, it must be specified explicitly, to avoid spurious `kotlin.Nothing` function return types.

Function body b is optional; if it is omitted, a function declaration creates an *abstract* function, which does not have an implementation. This is allowed only inside an abstract class. If a function body b is present, it should evaluate to type B which should satisfy $B <: R$.

A parameterized function declaration consists of five main parts.

- Name f;
- Type parameter list $T_1, \ldots, T_m$;
- Parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$;
- Return type R;
- Body b.

and extends the rules for a simple function declaration w.r.t. type parameter list. Further details are described here.

4.2.1 Function signature

In some cases we need to establish whether one function declaration *matches* another, e.g., for checking overridability. To do that, we compare *function signatures*, which consist of the following.

- Name f;
- Type parameter list $T_1, \ldots, T_m$ (if present);
- Parameter list $P_1, \ldots, P_n$.

Two function signatures A and B are considered *matching*, if the following is true.

- Name of A is the same as the name of B;
- Formal parameter types of A are pairwise equal to the formal parameter types of B w.r.t. possible type parameter substitutions;
- If the number of type parameters is the same, type parameters of A must be pairwise equivalent to the type parameters of B.

  Important: a platform implementation may change which function signatures are considered matching, depending on the platform's specifics.

4.2.2 Named, positional and default parameters

Kotlin supports *named* parameters out-of-the-box, meaning one can bind an argument to a parameter in function invocation not by its position, but by its name, which is equal to the argument name.

```
fun bar(a: Int, b: Double, s: String): Double = a + b + s.toDouble()

fun main(args: Array<String>) {
    println(bar(b = 42.0, a = 5, s = "13"))
}
```

Note: it is prohibited to bind the same named parameter to an argument several times, such invocations should result in a compile-time error.

All the names of named parameters are resolved at compile-time, meaning that performing a call with a parameter name not used at declaration-site is a compile-time error.

If one wants to mix named and positional arguments, the argument list must conform to the following form: $PoN_1, \ldots, PoN_M, N_1, \ldots, N_Q$, where PoN_i is an i-th argument in either positional or named form, N_j is a named argument irregardless of its position.

Note: in Kotlin version 1.3 and earlier, PoN_i were restricted to positional arguments only.

If one needs to provide a named argument to a variable length parameter, it can be achieved via either regular named argument `arg = arr` or a spread operator expression form `arg = *arr`. In both cases type of `arr` must be a subtype of ATS(`kotlin.Array(out` T`)`) for a variable length parameter of type T.

Note: in Kotlin version 1.3 and earlier, only the spread operator expression form for named variable length arguments was supported.

Kotlin also supports *default* parameters — parameters which have a default value used in function invocation, if the corresponding argument is missing. Note that default parameters cannot be used to provide a value for positional

argument *in the middle* of the positional argument list; allowing this would create an ambiguity of which argument for position i is the correct one: explicit one provided by the developer or implicit one from the default value.

```
fun bar(a: Int = 1, b: Double = 42.0, s: String = "Hello"): Double =
    a + b + s.toDouble()

fun main(args: Array<String>) {
    // Valid call, all default parameters used
    println(bar())
    // Valid call, defaults for `b` and `s` used
    println(bar(2))
    // Valid call, default for `b` used
    println(bar(2, s = "Me"))

    // Invalid call, default for `b` cannot be used
    println(bar(2, "Me"))
}
```

In summary, argument list should have the following form:

- Zero or more arguments in either positional or named form;
- Zero or more named arguments.

Missing arguments are bound to their default values, if they exist.

The evaluation order of argument list is described in Function calls and property access section of this specification.

4.2.3 Variable length parameters

One of the parameters may be designated as being variable length (aka *vararg*). A parameter list $(p_1, \ldots, \text{vararg } p_i : P_i = v_i, \ldots, p_n)$ means a function may be called with any number of arguments in the i-th position. These arguments are represented inside function body b as a value p_i of type, which is the result of *array type specialization* of type `kotlin.Array`(`out` P_i).

> Important: we also consider variable length parameters to have such types for the purposes of type inference and calls with named parameters.

If a variable length parameter is not last in the parameter list, all subsequent arguments in the function invocation should be specified as named arguments.

If a variable length parameter has a default value, it should be an expression which evaluates to a value of type, which is the result of *array type specialization* of type `kotlin.Array`(`out` P_i).

A value of type $Q <:$ ATS(`kotlin.Array`(`out` P_i)) may be *unpacked* to a variable length parameter in function invocation using spread operator; in this case array elements are considered to be separate arguments in the variable length parameter position.

> Note: this means that, for variable length parameters corresponding to specialized array types, unpacking is possible only for these specialized versions; for a variable length parameter of type `Int`, for example, unpacking is valid only for `IntArray`, and not for `Array<Int>`.

A function invocation may include several spread operator expressions corresponding to the vararg parameter. These may also be freely mixed with non-spread-expression arguments.

> Examples

```
fun foo(vararg i: Int) { ... }
fun intArrayOf(vararg i: Int): IntArray = i
...
// i is [1, 2, 3]
foo(1, 2, 3)
// i is [1, 2, 3]
foo(*intArrayOf(1, 2, 3))
// i is [1, 2, 3, 4, 5]
foo(1, 2, *intArrayOf(3, 4), 5)
// i is [1, 2, 3, 4, 5, 6]
foo(*intArrayOf(1, 2, 3), 4, *intArrayOf(5, 6))
```

4.2.4 Extension function declaration

An *extension function declaration* is similar to a standard function declaration, but introduces an additional special function parameter, the *receiver parameter*. This parameter is designated by specifying the receiver type (the type before `.` in function name), which becomes the type of this receiver parameter. This parameter is not named and must always be supplied (either explicitly or implicitly), e.g. it cannot be a variable-argument parameter, have a default value, etc.

Calling such a function is special because the receiver parameter is not supplied as an argument of the call, but as the *receiver* of the call, be it implicit or explicit. This parameter is available inside the scope of the function as the implicit receiver or `this`-expression, while nested scopes may introduce additional receivers that take precedence over this one. See the receiver section for details. This receiver is also available (as usual) in nested scope using labeled `this` syntax using the name of the declared function as the label.

For more information on how a particular receiver for each call is chosen, please refer to the overloading section.

> Note: when declaring extension functions inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested functions

For all other purposes, extension functions are not different from non-extension functions.

Examples:

```
fun Int.foo() { println(this + 1) } // this has type Int

fun main(args: Array<String>) {
    2.foo() // prints "3"
}

class Bar {
    fun foo() { println(this) } // this has type Bar
    fun Int.foo() { println(this) } // this has type Int
}
```

4.2.5 Inlining

A function may be declared `inline` using a special `inline` modifier. This allows the compiler to inline the function at call-site, replacing the call with the body of the function with arguments mapped to corresponding parameters. It is unspecified whether inlining will actually be performed, however.

Declaring a function `inline` has two additional effects:

- It allows type parameters of the function to be declared `reified`, making them runtime-available and allowing usage of specific expressions involving these parameters, such as type checks and class literals. Calling such a function is only allowed in a context where a particular type argument provided for this type parameter is also a runtime-available type.
- Any parameter of this function of a function type is treated as *inlined* parameter unless it has one of two special modifiers: `crossinline` or `noinline`. If a particular argument corresponding to inline parameter is a lambda literal, this lambda literal is considered *inlined* and, in particular, affects the way the return expressions are handled in its body. See the corresponding section for details.

Inlined parameters are not allowed to escape the scope of the function body, meaning that they cannot be stored in variables, returned from the function or captured by other values. They may only be called inside the function body or passed to other functions as inline arguments.

Crossinline parameters may not be stored or returned from the function, but may be captured (for example, by object literals or other noinline lambda literals).

Noinline parameters may be treated as any other values. They may also be passed to other functions as `noinline` or `crossinline` arguments.

Particular platforms may introduce additional restrictions or guarantees for the inlining mechanism.

Examples:

```
fun bar(value: Any?) {...}
inline fun fee(arg: () -> Unit) {...}
inline fun foo(inl: () -> Unit,
               crossinline cinl: () -> Unit,
               noinline noinl: () -> Unit) {
    // all arguments may be called
    inl()
    cinl()
    noinl()
    // all arguments may be passed as inline
    fee(inl)
    fee(cinl)
    fee(noinl)
    // passing to non-inline function
    // is allowed only for noinline parameters
    bar(inl) // not allowed
    bar(cinl) // not allowed
    bar(noinl) // allowed
    // capturing in a lambda expression
    // is allowed for noinline/crossinline parameters
    bar({ inl() }) // not allowed
    bar({ cinl() }) // allowed
    bar({ noinl() }) // allowed
}
```

4.2.6 Tail recursion optimization

A function may be declared *tail-recursive* by using a special `tailrec` modifier. A tail-recursive function that contains a recursive call to itself may be optimized to a non-recursive form by a particular platform in order to avoid problems of recursion such as a possibility of stack overflows possible on some platforms.

In order to be applicable for such an optimization, the function must adhere to tail recursive form: for all paths containing recursive calls the result of the recursive call must also be the result of the function. If a function declaration is marked with the `tailrec` modifier, but is not actually applicable for the optimization, it must produce a compile-time warning.

4.3 Property declaration

***propertyDeclaration*:**
[*modifiers*]
('val' | 'var')
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} '.']
({*NL*} (*multiVariableDeclaration* | *variableDeclaration*))
[{*NL*} *typeConstraints*]
[{*NL*} (('=' {*NL*} *expression*) | *propertyDelegate*)]
[(*NL* {*NL*}) ';']
{*NL*}
(([*getter*] [{*NL*} [*semi*] *setter*]) | ([*setter*] [{*NL*} [*semi*] *getter*]))

Kotlin uses *properties* to represent object-like entities, such as local variables, class fields or top-level values.

Property declarations may create read-only (`val`) or mutable (`var`) entities in their respective scope.

Properties may also have custom getter or setter — special functions which are used to read or write the property value. Getters and setters cannot be called directly, but rather define how the corresponding properties are evaluated when accessed.

4.3.1 Read-only property declaration

A read-only property declaration `val x: T = e` introduces `x` as a name of the result of `e`.

A read-only property declaration may include a custom getter in the form of

```
val x: T = e
    get(): T { ... } // (1)
```

or

```
val x: T = e
    get(): T = ... // (2)
```

in which case `x` is used as a synonym to the getter invocation. All of the right-hand value `e`, the type `T` in both positions, and the getter are optional, however, at least one of them must be specified. More so, if we cannot infer the resulting property type from the type of `e` or from the type of getter in expression form (2), the type `T` must be specified explicitly either as the property type, or as the getter return type. In case both `e` and `T` are specified, the type of `e` must be a subtype of `T` (see subtyping for more details).

The initializer expression `e`, if given, serves as the starting value for the property backing field (see getters and setters section for details) and is evaluated when the property is created. Properties that are not allowed to have backing fields (see getters and setters section for details) are also not allowed to have initializer expressions.

> Note: although a property with an initializer expression looks similar to an assignment, it is different in several key ways: first, a read-only property cannot be assigned, but may have an initializer expression; second, the initializer expression never invokes the property setter, but assigns the property backing field value directly.

4.3.2 Mutable property declaration

A mutable property declaration `var x: T = e` introduces `x` as a name of a mutable variable with type `T` and initial value equals to the result of `e`. The rules regarding the right-hand value `e` and the type `T` match those of a read-only property declaration.

A mutable property declaration may include a custom getter and/or custom setter in the form of

```
var x: T = e
    get(): TG { ... }
    set(value: TS) { ... }
```

in which case `x` is used as a synonym to the getter invocation when read from and to the setter invocation when written to.

4.3.3 Local property declaration

If a property declaration is local, it creates a local entity which follows most of the same rules as the ones for regular property declarations. However, local property declarations cannot have custom getters or setters.

Local property declarations also support *destructuring* declaration in the form of

```
val (a: T, b: U, c: V, ...) = e
```

which is a syntactic sugar for the following expansion

```
val a: T = e.component1()
val b: U = e.component2()
val c: V = e.component3()
...
```

where `componentN()` should be a valid operator function available on the result of `e`. Some of the entries in the destructuring declaration may be replaced

with an *ignore marker* `_`, which signifies that no variable is declared and no `componentN()` function is called.

As with regular property declaration, type specification is optional, in which case the type is inferred from the corresponding `componentN()` function. Destructuring declarations cannot use getters, setters or delegates and must be initialized in-place.

4.3.4 Getters and setters

As mentioned before, a property declaration may include a custom getter and/or custom setter (together called *accessors*) in the form of

```
var x: T = e
    get(): TG { ... }
    set(anyValidArgumentName: TS): RT { ... }
```

These functions have the following requirements

- $TG \equiv T$;
- $TS \equiv T$;
- $RT \equiv$ `kotlin.Unit`;
- Types TG, TS and RT are optional and may be omitted from the declaration;
- Read-only properties may have a custom getter, but not a custom setter;
- Mutable properties may have any combination of a custom getter and a custom setter
- Setter argument may have any valid identifier as argument name.

  Note: Regular coding convention recommends `value` as the name for the setter argument

One can also omit the accessor body, in which case a *default* implementation is used (also known as default accessor).

```
var x: T = e
    get
    set
```

This notation is usually used if you need to change some aspects of an accessor (i.e., its visibility) without changing the default implementation.

Getters and setters allow one to customize how the property is accessed, and may need access to the property's *backing field*, which is responsible for actually

storing the property data. It is accessed via the special `field` property available inside accessor body, which follows these conventions

- For a property declaration of type `T`, `field` has the same type `T`
- `field` is read-only inside getter body
- `field` is mutable inside setter body

However, the backing field is created for a property only in the following cases

- A property has no custom accessors;
- A property has a default accessor;
- A property has a custom accessor, and it uses `field` property;
- A mutable property has a custom getter or setter, but not both.

In all other cases a property has no backing field. Properties without backing fields are not allowed to have initializer expressions.

Read/write access to the property is replaced with getter/setter invocation respectively. Getters and setters allow for some modifiers available for function declarations (for example, they may be declared `inline`, see grammar for details). Properties themselves may also be declared `inline`, meaning that both getter and setter of said property are `inline`.

4.3.5 Delegated property declaration

A delegated read-only property declaration `val x: T by e` introduces `x` as a name for the *delegation* result of property `x` to the entity `e` or to the delegatee of `e` provided by `provideDelegate`. For the former, one may consider these properties as regular properties with a special *delegating* getters:

```
val x: T by e
```

is the same as

```
val x$delegate = e
val x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
```

Here every access to such property (`x` in this case) becomes an overloadable form which is expanded into the following:

```
e.getValue(thisRef, property)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `getValue` is a suitable operator function available on `e`;
- `thisRef` is the receiver object for the property. This argument is `null` for local properties;

- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details).

A delegated mutable property declaration `var x: T by e` introduces `x` as a name of a mutable entity with type `T`, access to which is *delegated* to the entity `e` or to the delegatee of `e` provided by `provideDelegate`. As before, one may view these properties as regular properties with special *delegating* getters and setters:

```
var x: T by e
```

is the same as

```
val x$delegate = e
var x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
    set(value: T) { x$delegate.setValue(thisRef, ::x, value) }
```

Read access is handled the same way as for a delegated read-only property. Any write access to `x` (using, for example, an assignment operator `x = y`) becomes an overloadable form with the following expansion:

```
e.setValue(thisRef, property, y)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `setValue` is a suitable operator function available on `e`;
- `thisRef` is the receiver object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details);
- `y` is the value `x` is assigned to. In case of complex assignments (see the assignment section), as they are all overloadable forms, first the assignment expansion is performed, and after that, the expansion of the delegated property using normal assignment.

The type of a delegated property may be omitted at the declaration site, meaning that it may be inferred from the delegating function itself, as it is with regular getters and setters. If this type is omitted, it is inferred as if it was assigned the value of its expansion. If this inference fails, it is a compile-time error.

If the delegate expression has a suitable operator function called `provideDelegate`, a *provided* delegate is used instead. The provided delegate is accessed using the following expansion:

```
val x: T by e
```

is the same as

```
val x$delegate = e.provideDelegate(thisRef, ::x)
val x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
```

and

```
var x: T by e
```

is the same as

```
val x$delegate = e.provideDelegate(thisRef, ::x)
val x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
    set(value) { x$delegate.setValue(thisRef, ::x, value) }
```

where `provideDelegate` is a suitable operator function available using the receiver `e`, while `getValue` and `setValue` work the same way they do with normal property delegation. As is the case with`setValue` and `getValue`, `thisRef` is a reference to the receiver of the property or `null` for local properties, but there is also a special case: for extension properties `thisRef` supplied to `provideDelegate` is `null`, while `thisRef` provided to `getValue` and `setValue` is the actual receiver. This is due to the fact that, during the creation of the property, no receiver is available.

For both provided and standard delegates, the generated delegate value is placed in the same context as its corresponding property. This means that for a class member property it will be a synthetic member, for a local property it is a local value in the same scope as the property and for top-level (both extension and non-extension) properties it will be a top-level value. This affects this value's lifetime in the same way normal value lifetime works.

Example:

```
operator fun <V, R : V> Map<in String, V>.getValue(
        thisRef: Any?, property: KProperty<*>): R =
    getOrElse(property.name) {
        throw NoSuchElementException()
    } as R

operator fun <V> MutableMap<in String, V>.setValue(
        thisRef: Any?, property: KProperty<*>, newValue: V) =
    set(property.name, newValue)

fun handleConfig(config: MutableMap<String, Any?>) {
    val parent by config       // Any?
    val host: String by config // String
    var port: Int by config    // Int

    // Delegating property accesses to Map.getValue
    // Throwing NSEE as there is no "port" key in the map
```

```
    // println("$parent: going to $host:$port")

    // Delegating property access to Map.setValue
    port = 443
    // Map now contains "port" key

    // Delegating property accesses to Map.getValue
    // Not throwing NSEE as there is "port" key in the map
    println("$parent: going to $host:$port")
}

fun main() {
    handleConfig(mutableMapOf(
        "parent" to "",
        "host" to "https://kotlinlang.org/"
    ))
}
```

Example with `provideDelegate`:

```
operator fun <V> MutableMap<in String, V>.provideDelegate(
        thisRef: Any?,
        property: KProperty<*>): MutableMap<in String, V> =
    if (containsKey(property.name)) this
    else throw NoSuchElementException()

operator fun <V, R : V> Map<in String, V>.getValue(
        thisRef: Any?, property: KProperty<*>): R = ...

operator fun <V> MutableMap<in String, V>.setValue(
        thisRef: Any?, property: KProperty<*>, newValue: V) = ...

fun handleConfig(config: MutableMap<String, Any?>) {
    val parent by config       // Any?
    val host: String by config // String
    var port: Int by config    // Int
    // Throwing NSEE here as `provideDelegate`
    //   checks for "port" key in the map

    ...
}

fun main() {
    handleConfig(mutableMapOf(
        "parent" to "",
        "host" to "https://kotlinlang.org/"
    ))
```

```
}
```

4.3.6 Extension property declaration

An *extension property declaration* is similar to a standard property declaration, but, very much alike an extension function, introduces an additional parameter to the property called *the receiver parameter*. This is different from usual property declarations, that do not have any parameters. There are other differences from standard property declarations:

- Extension properties cannot have initializers;
- Extension properties cannot have backing fields;
- Extension properties cannot have default accessors.

  Note: informally, on can say that extension properties have no state of their own. Only properties that use other objects' storage facilities and/or uses constant data can be extension properties.

Aside from these differences, extension properties are similar to regular properties, but, when accessing such a property one always need to supply a *receiver*, implicit or explicit. Like for regular properties, the type of the receiver must be a subtype of the receiver parameter, and the value that is supplied as the receiver is bound to the receiver parameter. For more information on how a particular receiver for each access is chosen, please refer to the overloading section.

The receiver parameter can be accessed inside getter and setter scopes of the property as the implicit receiver or `this`. It may also be accessed inside nested scopes using labeled `this` syntax using the name of the property declared as the label. For delegated properties, the value passed into the operator functions `getValue` and `setValue` as the receiver is the value of the receiver parameter, rather than the value of the outer classifier. This is also true for local extension properties: while regular local properties are passed `null` as the first argument of these operator functions, local extension properties are passed the value of the receiver argument instead.

Note: when declaring extension properties inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested properties

For all other purposes, extension properties are not different from non-extension properties.

Examples:

```
val Int.foo: Int get() = this + 1

fun main(args: Array<String>) {
    println(2.foo.foo) // prints "4"
}
```

```
class Bar {
    val foo get() = this // returns type Bar
    val Int.foo get() = this // returns type Int
}
```

4.3.7 Property initialization

All non-abstract properties must be definitely initialized before their first use. To guarantee this, Kotlin compiler uses a number of analyses which are described in more detail here.

4.3.8 Constant properties

A property may be declared **constant**, meaning that its value is known during compilation, by using the special `const` modifier. In order to be declared `const`, a property must meet the following requirements:

- Its type is one of the following:
  - One of the the built-in integral types;
  - One of the the built-in floating types;
  - `kotlin.Boolean`;
  - `kotlin.Char`;
  - `kotlin.String`;
- It is declared in the top-level scope or inside an object declaration;
- It has an initializer expression and this initializer expression can be evaluated at compile-time. Integer literals and string interpolation expressions without evaluated expressions, as well as built-in arithmetic/comparison operations and string concatenation operations on those are such expressions, as well as other constant properties, but it is implementation-defined which other expressions qualify for this;
- It does not have getters, setters or delegation specifiers.

  Example:

```
// Correct constant properties
const val answer = 2 * 21
const val msg = "Hello World!"
const val calculated = answer + 45

// Incorrect constant property
const val emptyStringHashCode = "".hashCode()
```

4.3.9 Late-initialized properties

A mutable member property can be declared with a special `lateinit` modifier, effectively turning off the property initialization checks for it. Such a property is called late-initialized and may be used for values that are supposed to be initialized not during object construction, but during some other time (for example, a special initialization function). This means, among other things, that it is the responsibility of the programmer to guarantee that the property is initialized before its usage.

A property may be declared late-initialized if:

- It has no custom getters, setters or delegation;
- It is a member or a top-level property;
- It is mutable;
- It has declared non-nullable type which is also not one of the following types:
  - One of the built-in integer types;
  - One of the built-in floating types;
  - `kotlin.Boolean`;
  - `kotlin.Char`.

4.4 Type alias

***typeAlias*:**
[*modifiers*]
`'typealias'`
{*NL*}
simpleIdentifier
[{*NL*} *typeParameters*]
{*NL*}
`'='`
{*NL*}
type

Type alias introduces an alternative name for the specified type and supports both simple and parameterized types. If type alias is parameterized, its type parameters must be unbounded. Another restriction is that recursive type aliases are forbidden — the type alias name cannot be used in its own right-hand side.

At the moment, Kotlin supports only top-level type aliases. The scope where it is accessible is defined by its *visibility modifiers*.

4.5 Declarations with type parameters

Most declarations may be introduced as *generic*, introducing type parameters that must be explicitly specified or inferred when the corresponding declaration is used. For declarations that introduce new types this mechanism provides the means of introducing a parameterized type. Please refer to the corresponding section for details.

Type parameters may be used as types inside the scope introduced by the declaration. When such a declaration is used, the parameters are substituted by types available inside the scope the declaration is used in.

The following declarations are not allowed to have type parameters:

- Non-extension property declarations;
- Object declarations (including companion object declarations);
- Constructor declarations;
- Getters and setters of property declarations;
- Enum class declarations;
- Annotation class declarations;
- Classifier declarations inheriting from `kotlin.Throwable`.

Type parameters are allowed to specify *subtyping restrictions* on them in the form `T : U`, meaning $T <: U$ where T is a type parameter and U is some other type available in the scope the declaration is declared in. These either are written directly at the parameter placement syntax or using a special `where` syntax. Any number of restrictions is allowed on a single type, however, for a given type parameter `T`, only one restriction `T : U` can have `U` to be another type parameter.

These restrictions are turned into corresponding type constraints when the type parameters are substituted with types and are employed during type inference and overload resolution of any usage of the corresponding declaration. See the corresponding sections for details.

Type parameters do not introduce runtime-available types unless declared `reified`. Only type parameters of inline functions can be declared `reified`.

4.5.1 Type parameter variance

The declaration-site variance of a particular type parameter for a classifier declaration is specified using special keywords `in` (for covariant parameters) and `out` (for contravariant parameters). If the variance is not specified, the parameter is implicitly declared invariant. See the type system section for details.

A type parameter is **used in covariant position** in the following cases:

- It is used as an argument in another generic type and the corresponding parameter in that type is covariant;
- It is the return type of a function;
- It is a type of a property.

A type parameter is **used in contravariant position** in the following cases:

- It is used as an argument in another generic type and the corresponding parameter in that type is contravariant;
- It is a type of an parameter of a function;
- It is a type of a mutable property.

A type parameter is used in an invariant position if it is used as an argument in another generic type and the corresponding parameter in that type is invariant.

A usage of a contravariant type parameter in a covariant or invariant position, as well as usage of a covariant type parameter in a contravariant or invariant position, results in **variance conflict** and a compiler error, unless the containing declaration is private to the type parameter owner (in which case its visibility is restricted, see the visibility section for details). This applies only to member declarations of the corresponding class, extensions are not subject to this limitation.

This restrictions may be lifted in particular cases by annotating the corresponding type parameter usage with a special built-in annotation `kotlin.UnsafeVariance`. By supplying this annotation the author of the code explicitly declares that safety features that variance checks provide are not needed in this particular declarations.

Examples:

```
class Inv<T> {
    fun a(): T {...} // Ok, covariant usage
    fun b(value: T) {...} // Ok, contravariant usage
    fun c(p: Out<T>) {...} // Ok, covariant usage
    fun d(): Out<T> {...} // Ok, covariant usage
    fun e(p: In<T>) {...} // Ok, contravariant usage
    fun f(): In<T> {...} // Ok, contravariant usage
}

class Out<out T> { // T is covariant
    fun a(): T {...} // Ok, covariant usage
    fun b(value: T) {...} // ERROR, contravariant usage
    fun c(p: Inv<T>) {...} // ERROR, invariant usage
    fun d(): Inv<T> {...} // ERROR, invariant usage
}

class In<in T> { // T is contravariant
    fun a(): T {...} // ERROR, covariant usage
```

```
    fun b(value: T) {...} // Ok, contravariant usage
    fun c(p: Inv<T>) {...} // ERROR, invariant usage
    fun d(): Inv<T> {...} // ERROR, invariant usage
}
```

Any of these restrictions may be lifted using `@UnsafeVariance` annotation on the type argument:

```
class Out<out T> { // T is covariant
    fun b(value: @UnsafeVariance T) {...} // Ok
}

class In<in T> { // T is contravariant
    fun a(): @UnsafeVariance T {...} // Ok
}
```

Using `@UnsafeVariance` is inherently unsafe and should be used only when the programmer can guarantee that variance violations would not result in runtime errors. For example, receiving a value in a contravariant position for a covariant class parameter is usually ok if the function involved is guaranteed not to mutate internal state of the class.

For examples on how restrictions are lifted for private visibility (private-to-this), see visibility section

4.5.2 Reified type parameters

Type parameters of inline function declarations (and only those) can be declared `reified` using the corresponding keyword. A reified type parameter is a runtime-available type inside the function scope, see the corresponding section for details. Reified type parameters can only be substituted by other runtime-available types when using such functions.

Example:

```
fun <T> foo(value: Any?) {
    // ERROR, is-operator is only allowed for runtime-available types
    if(value is T) ...
}

inline fun <reified T> foo(value: Any?) {
    if(value is T) ... // Ok
}
```

4.6 Declaration visibility

Each declaration has a visibility property relative to the scope it is declared in. By default, all the declarations are `public`, meaning that they can be accessed from any other scope their outer scope can be accessed from. The only exception to this rule are overriding declarations that by default inherit the visibility from the declaration they override. Declarations may be also marked `public` explicitly.

Declarations marked as `private` can only be accessed from the same scope they are declared in. For example, all `private` top-level declarations in a file may only be accessed by code from the same file.

Some `private` declarations are special in that they have an even more restricted visibility, called "`private` to `this`". These include declarations that are allowed to lift certain variance rules in their types as long as they are never accessed outside `this` object, meaning that they can be accessed using `this` as the receiver, but are not visible on other instances of the same class even in the methods of this class. For example, for a class declaration C with type parameter T it is not allowed to introduce declarations involving T with conflicting variance, unless they are declared `private`. That is, if T is declared as covariant, any declarations with a type using T in a contravariant position (including properties with type T itself if they are mutable) and if T is declared as contravariant, any declarations with a type using T in a covariant position (including properties with type T itself) are forbidden, unless they are declared using `private` visibility, in which case they are instead treated as "`private` to `this`".

Example:

```
class Foo<out T>(val t: T) { // T is a covariant parameter
    // not allowed, T is in contravariant position
    public fun set1(t: T) {}
    // allowed, set2 is private-to-this
    private fun set2(t: T) {}
    private fun bar(other: Foo<T>) {
        // allowed, set2 is called on this
        this.set2(t)
        // not allowed, set2 is called on other
        other.set2(t)
    }
}
```

Note: the above does not account for `@UnsafeVariance` annotation that lifts any variance restrictions on type parameters

Declarations marked as `internal` may only be accessed from the same module, treated as `public` from inside the module and as `private` from outside the module.

Declarations in classifier declaration scope can also be declared `protected`, meaning that they can only be accessed from the same classifier type as well as any types inheriting from this type regardless of the scope they are declared in.

There is a partial order of *weakness* between different visibility modifiers:

- `protected` and `internal` are weaker than `private`;
- `public` is weaker than `protected` and `internal`.

  Note: there is a certain restriction regarding `inline` functions that have a different visibility from entities they access. In particular, an `inline` function cannot access entities with a stronger visibility (i.e. `public inline` function accessing a `private` property). There is one exception to this: a `public inline` function can access `internal` entities which are marked with a special builtin annotation `@PublishedApi`.

  Example:

```
class Foo<T>(internal val t: T) {
    // not allowed, t is internal, getValue is public
    inline fun getValue(): T = t
}
class Bar<T>(@PublishedApi internal val t: T) {
    // allowed through @PublishedApi
    inline fun getValue(): T = t
}
```

Chapter 5

Inheritance

Kotlin is an object-oriented language with its object model based on **inheritance**.

5.1 Classifier type inheritance

Classifier types may be inherited from each other: the type inherited *from* is called the *base type*, while the type which inherits the base type is called the *derived type*. The following limitations are imposed on the possible inheritance structure.

A class or object type is allowed to inherit from only one class type (called its **direct superclass**) and multiple interface types. As specified in the declaration section, if the superclass of a class or object type is not specified, it is assumed to be `kotlin.Any`. This means, among other things, that every class or object type always has a direct superclass.

A class is called **closed** and cannot be inherited from if it is not explicitly declared as either `open` or `abstract`.

> Note: classes are neither `open` nor `abstract` by default.

A `data class`, `enum class` or `annotation class` cannot be declared `open` or `abstract`, i.e., are always closed and cannot be inherited from. Declaring a class `sealed` also implicitly declares it `abstract`.

An interface type may be inherited from any number of other interface types (and only interface types), if the resulting type is well-formed.

Object types cannot be inherited from.

Inheritance is the primary mechanism of introducing subtyping relations between user-defined types in Kotlin. When a classifier type A is declared with base types

$B_1, \ldots, B_m$, it introduces subtyping relations $A <: B_1, \ldots, A <: B_m$, which are then used in overload resolution and type inference mechanisms.

5.1.1 Abstract classes

A class declared `abstract` cannot be instantiated, i.e., an object of this class cannot be created directly. Abstract classes are implicitly `open` and their primary purpose is to be inherited from. Only abstract classes allow for `abstract` property and function declarations in their scope.

5.1.2 Sealed classes and interfaces

A class or interface (but not a functional interface) may be declared `sealed`, making it special from the inheritance point-of-view.

- A `sealed` class is implicitly `abstract` (and these two modifiers are exclusive);
- A `sealed` class or interface can only be inherited from by types declared in the same package and in the same module, and which have a fully-qualified name (meaning local and anonymous types cannon be inherited from `sealed` types);
- `Sealed` classes and interfaces allow for exhaustiveness checking of when expressions for values of such types. Any sealed type `S` is associated with its *direct non-sealed subtypes*: a set of non-sealed types, which are either direct subtypes of `S` or transitive subtypes of `S` via some number of other *sealed* types. These direct non-sealed subtypes form the boundary for exhaustiveness checks.

5.1.3 Inheritance from built-in types

Built-in types follow the same rules as user-defined types do. Most of them are closed class types and cannot be inherited from. Function types are treated as interfaces and can be inherited from as such.

5.2 Overriding

A callable declaration (that is, a property or member function declaration) inside a classifier declaration is said to be *overridable* if:

- Its visibility (and the visibility of its getter and setter, if present) is not `private`;

- It is declared as `open`, `abstract` or `override` (interface methods and properties are implicitly `abstract` if they don't have a body or `open` if they do).

It is illegal for a declaration to be both `private` and either `open`, `abstract` or `override`, such declarations should result in a compile-time error.

A callable declaration D inside a classifier declaration *subsumes* a name-matching declaration B of the base classifier type if the following are true.

- B and D are declarations of the same kind (property declarations or function declarations);
- Function signature of D (if any) matches function signature of B (if any).

If the declaration B of the base classifier type is overridable, the declaration D of the derived classifier type subsumes it, and D has an `override` modifier, D is *overriding* the base declaration B.

A function declaration D which overrides function declaration B should satisfy the following conditions.

- Return type of D is a subtype of return type of B;
- Suspendability of D and B must be the same.

A property declaration D which overrides property declaration B should satisfy the following conditions.

- Mutability of D is not stronger than mutability of B (where read-only `val` is stronger than mutable `var`);
- Type of D is a subtype of type of B; except for the case when both D and B are mutable (`var`), then types of D and B must be equivalent.

Otherwise, it is a compile-time error.

If the base declaration is not overridable and/or the overriding declaration does not have an `override` modifier, it is not permitted and should result in a compile-time error.

If the overriding declaration *does not* have its visibility specified, its visibility is implicitly set to be the same as the visibility of the overridden declaration.

If the overriding declaration *does* have its visibility specified, it must not be stronger than the visibility of the overridden declaration.

Examples:

```
open class B {
    protected open fun f() {}
}
class C : B() {
    open override fun f() {}
    // `f` is protected, as its visibility is
    //   inherited from the base declaration
```

```
}
class D : B() {
    public open override fun f() {}
    // this is correct, as public visibility is
    //   weaker that protected visibility
    //   from the base declaration
}

open class P {
    open fun g() {}
}

class Q : P() {
    protected open override fun g() {}
    // this is an error, as protected visibility is
    //   stronger that public visibility
    //   from the base declaration
}
```

Important: platforms may introduce additional cases of both *overridability* and *subsumption* of declarations, as well as limit the overriding mechanism due to implementation limitations.

Note: Kotlin does not have a concept of full hiding (or shadowing) of declarations.

Note: if a declaration binds a new function to the same name as was introduced in the base class, but which does not subsume it, it is neither a compile-time error nor an overriding declaration. In this case these two declarations follow the normal rules of overloading. However, these declarations may still result in a compile-time error as a result of conflicting overload detection.

Chapter 6

Scopes and identifiers

Kotlin program is logically divided into *scopes*.

A scope is a syntactically-delimited region of code which constitutes a context in which entities and their names can be introduced. Scopes can be nested, with entities introduced in outer scopes possibly available in the inner scopes if linked.

The top level of a Kotlin file is also a scope, containing all the scopes within a file.

All the scopes are divided into two kinds: declaration scopes and statement scopes. These two kinds of scopes differ in how the identifiers may refer to the entities defined in the scopes.

Declaration scopes include:

- The project modules;
- The project packages;
- The top level scopes of non-script Kotlin files;
- The bodies of classifier declarations;
- The bodies of object literals.

Statement scopes include:

- The top level scopes of script Kotlin files;
- Scopes produced by control structure bodies of different expressions;
- The bodies of function declarations;
- The bodies of function literals;
- The bodies of getters and setters of properties;
- The bodies of constructors;
- The bodies of instance initialization blocks in classifier declarations.

All declarations in a particular scope introduce new *bindings* of identifiers in this scope to their respective entities in the program. These entities may be types or

values, where values refer to objects, functions or properties (which may also be delegated). Top-level scopes additionally allow to introduce new bindings using `import` directives from other top-level scopes.

In most situations, it is not allowed to bind several values to the same identifier in the same scope, but it is allowed to bind a value to an identifier already available in the scope through linked scopes or imports.

An exception to this rule are function declarations, which are matched by signatures and allow defining several functions with the same name in the same scope. When calling functions, a process called overload resolution allows for differentiating between such functions. Overload resolution also applies to properties if they are used as functions through `invoke`-convention, but it does not allow defining several properties with the same name in the same scope.

> Note: platforms may introduce additional restrictions on which identifiers may be declared together in the same or linked scopes.

The main difference between declaration scopes and statement scopes is that names in the statement scope are bound in the order of appearance. It is not allowed to access a value through an identifier in code which (syntactically) precedes the binding itself. On the contrary, in declaration scopes it is fully allowed, although initialization cycles may occur leading to unspecified behaviour.

> Note: Kotlin compiler may attempt to detect and report such initialization cycles as compile-time warnings or errors.

It also means that statement scopes nested inside declaration scopes may access values declared afterwards in parent declaration scopes, but any values declared inside a statement scope can be accessed only after their declaration point.

Examples:

- In declaration scope:

```
// x refers to the property defined below
//   even if there is another property
//   called x in outer scope or imported
fun foo() = x + 2
val x = 3
```

- In statement scope:

```
// x refers to another property
//   defined in outer scope or imported
//   or is a compile-time error
fun foo() = x + 2
val x = 3
```

6.1 Linked scopes

Scopes `A` and `B` in a Kotlin program may be *downwards-linked* (`A ~> B`), meaning identifiers from `A` can be used in `B` without the need for additional qualification. If scopes `A` and `B` are downwards-linked, scopes `B` and `A` are considered *upwards-linked* (`B <~ A`).

> Note: link relation is transitive, unless specified otherwise.

Scopes are downwards-linked (DLD) or upwards-linked (ULD) as follows:

- A statement scope is DLD to any directly nested scope;
- An object declaration scope is DLD to any nested scopes;
- An object declaration scope is non-transitively ULD to the companion object scopes of its superclasses;
- An object declaration scope is non-transitively ULD to the companion object scopes of its parent classifier superclasses;
- An object declaration scope is ULD to the companion object declaration scope of its parent classifier;
- A companion object declaration scope is DLD to any nested scopes;
- A companion object declaration scope is non-transitively ULD to the companion object scopes of its superclasses;
- A companion object declaration scope is non-transitively ULD to the companion object scopes of its parent classifier superclasses;
- A companion object declaration scope is ULD to the companion object declaration scope of the *parent* of its parent classifier;
- A classifier or nested class declaration scope is DLD to any nested statement scopes;
- A classifier or nested class declaration scope is ULD to its companion object declaration scope;
- An inner class declaration scope is DLD to any nested statement scopes;
- An inner class declaration scope is ULD to the classifier declaration scope of its parent classifier.

> Important: linked scopes **do not** cover cases when identifiers from supertypes are used in subtypes, as this is covered by the inheritance rules.

6.2 Identifiers and paths

Kotlin program operates with different *entities*, such as classes, interfaces, values, etc. An entity can be referenced using its *path*: a sequence of identifiers which references this entity in a given scope.

Kotlin supports two kinds of paths.

- Simple paths `P`, which consist of a single identifier
- Qualified paths `P.m`, which consist of a path `P` and a member identifier `m`

Besides identifiers which are introduced by the developer (e.g., via declaring classes or introducing variables), there are several predefined identifiers with special semantics.

- `this` – an identifier which references the default receiver available in the current scope, further details are available here
- `this@label` – an identifier which references the default receiver available in the selected scope, further details are available here
- `super<Klazz>` – an identifier which references the supertype `Klazz` available in the current scope, further details are available here
- `super<Klazz>@label` – an identifier which references the supertype `Klazz` available in the selected scope, further details are available here

6.3 Labels

Labels are special syntactic marks which allow one to reference certain code fragments or elements. Lambda expressions) and loop statements are allowed to be labeled, with label identifier associated with the corresponding entity.

> Note: in Kotlin version 1.3 and earlier, labels were allowed to be placed on any expression or statement.

Labels can be *redeclared*, meaning that the same label identifier may be reused with different parts of code (or even on the same expression/loop) several times. Labels are *scoped*, meaning that they are only available in the scope they were declared in.

Labels are used by certain expressions, such as `break`, `continue` and `return`, to specify exactly what entity the expression corresponds to. Please refer to the corresponding sections for details.

When resolving labels (determining which label an expression refers to), the *closest* label with the matching identifier is chosen, i.e., a label in an innermost scope syntactically closest to the point of its use.

Chapter 7

Statements

***statements*:**
[*statement* {*semis statement*}] [*semis*]

***statement*:**
{*label* | *annotation*} (*declaration* | *assignment* | *loopStatement* | *expression*)

Kotlin does not explicitly distinguish between statements, expressions and declarations, i.e., expressions and declarations can be used in statement positions. This section focuses only on those statements that are *not* expressions or declarations. For information on those parts of Kotlin, please refer to the Expressions and Declarations sections of the specification.

> Example: Kotlin supports using conditionals both as expressions and as statements. As their use as expressions is more general, detailed information about conditionals is available in the Expressions section of the specification.

7.1 Assignments

***assignment*:**
((*directlyAssignableExpression* `'='`) | (*assignableExpression assignmentAndOperator*)) {*NL*} *expression*

***assignmentAndOperator*:**
`'+='`
| `'-='`
| `'*='`
| `'/='`
| `'%='`

An *assignment* is a statement that writes a new value to some program entity, denoted by its left-hand side. Both left-hand and right-hand sides of an assignment must be expressions, more so, there are several restrictions for the expression on the left-hand side.

For an expression to be *assignable*, i.e. be allowed to occur on the left-hand side of an assignment, it **must** be one of the following:

- An identifier referring to a mutable property;
- A navigation expression referring to a mutable property. If this navigation operator is the safe navigation operator, this introduces a special case of *safe assignment*;
- An indexing expression.

  Note: Kotlin assignments **are not** expressions and cannot be used as such.

7.1.1 Simple assignments

A *simple assignment* is an assignment which uses the assign operator `=`. If the left-hand side of an assignment refers to a mutable property, a value of that property is changed when an assignment is evaluated, using the following rules (applied in order).

- If a property has a setter (including delegated properties), it is called using the right-hand side expression as its argument;
- Otherwise, if a property is a mutable property, its value is changed to the evaluation result of the right-hand side expression.

If the left-hand side of an assignment is an indexing expression, the whole statement is treated as an overloaded operator with the following expansion:

$A[B_1, B_2, B_3, \ldots, B_N] = C$ is the same as calling A`.set`$(B_1, B_2, B_3, \ldots, B_N, C)$ where `set` is a suitable operator function.

7.1.2 Operator assignments

An *operator assignment* is a combined-form assignment which involves one of the following operators: `+=`, `-=`, `*=`, `/=`, `%=`. All of these operators are overloadable operator functions with the following expansions (applied in order):

- A `+=` B is exactly the same as one of the following:
  - A`.plusAssign(`B`)` if a suitable `plusAssign` operator function exists and is available;
  - A `=` A`.plus(`B`)` if a suitable `plus` operator function exists and is available.
- A `-=` B is exactly the same as one of the following:

  - A`.minusAssign(`B`)` if a suitable `minusAssign` operator function exists and is available;
  - A `=` A`.minus(`B`)` if a suitable `minus` operator function exists and is available.
- A `*=` B is exactly the same as one of the following:
  - A`.timesAssign(`B`)` if a suitable `timesAssign` operator function exists and is available;
  - A `=` A`.times(`B`)` if a suitable `times` operator function exists and is available.
- A `/=` B is exactly the same as one of the following:
  - A`.divAssign(`B`)` if a suitable `divAssign` operator function exists and is available;
  - A `=` A`.div(`B`)` if a suitable `div` operator function exists and is available;
- A `%=` B is exactly the same as one of the following:
  - A`.remAssign(`B`)` if a suitable `remAssign` operator function exists and is available;
  - A `=` A`.rem(`B`)` if a suitable `rem` operator function exists and is available.

  Note: before Kotlin version 1.3, there were additional overloadable functions for `%` called `mod`/`modAssign`

After the expansion, the resulting function call expression or simple assignment is processed according to their corresponding rules, and overload resolution and type checking are performed. If both expansion variants result in correctly resolved and inferred code, this should be reported as an operator overloading ambiguity. If only one of the expansion variants can be resolved correctly, this variant is picked as the correct one. If neither of variants result in correct code, the operator calls must be reported as unresolved.

> Example: consider the following compound operator statement: `x[y] += z`. The corresponding expansion variants are `x.get(y).plusAssign(z)` and `x.set(x.get(y).plus(z))` according to expansion rules for corresponding operators. If, for example, the call to `set` in the second variant results in resolution or inference error, the whole corresponding expansion is deemed unresolved and the first variant is picked if applicable.
>
> Note: although for most real-world use cases operators `++` and `--` are similar to operator assignments, in Kotlin they are expressions and are described in the corresponding section of this specification.

7.1.3 Safe assignments

If the left-hand side of an assignment involves a safe-navigation operator, it is treated as a special case of *safe assignment*. Safe assignments are expanded

similar to safe navigation operator expressions:

- `a?.c` is exactly the same as

  ```
  when(val $tmp = a) {
      null -> null
      else -> { $tmp.c }
  }
  ```

  For any right-hand combinations of operators present in `c`, which are expanded further, as usual.

  Example: The assignment

  ```
  x?.y[0] = z
  ```

  is expanded to

  ```
  when(val $tmp = x) {
      null -> null
      else -> { $tmp.y[0] = z }
  }
  ```

  which, according to expansion rules for indexing assignments is, in turn, expanded to

  ```
  when(val $tmp = x) {
      null -> null
      else -> { $tmp.y.set(0, z) }
  }
  ```

7.2 Loop statements

Loop statements describe an evaluation of a certain number of statements repeatedly until a *loop exit condition* applies.

***loopStatement*:**
forStatement
| *whileStatement*
| *doWhileStatement*

Loops are closely related to the semantics of jump expressions, as these expressions, namely `break` and `continue`, are only allowed in a body of a loop. Please refer to the corresponding sections for details.

7.2.1 While-loop statements

***whileStatement*:**
`'while'`

{*NL*}
'('
expression
')'
{*NL*}
(*controlStructureBody* | ';')

A *while-loop statement* is similar to an `if` expression in that it also has a condition expression and a body consisting of zero or more statements. While-loop statement evaluating its body repeatedly for as long as its condition expression evaluates to true or a jump expression is evaluated to finish the loop.

> Note: this also means that the condition expression is evaluated before every evaluation of the body, including the first one.

The while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

7.2.2 Do-while-loop statements

***doWhileStatement*:**

'do'
{*NL*}
[*controlStructureBody*]
{*NL*}
'while'
{*NL*}
'('
expression
')'

A *do-while-loop statement*, similarly to a while-loop statement, also describes a loop, with the following differences. First, it has a different syntax. Second, it evaluates the loop condition expression **after** evaluating the loop body.

> Note: this also means that the body is always evaluated at least once.

The do-while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

7.2.3 For-loop statements

***forStatement*:**

'for'
{*NL*}
'('
{*annotation*}
(*variableDeclaration* | *multiVariableDeclaration*)

'in'
expression
')'
{*NL*}
[*controlStructureBody*]

Note: unlike most other languages, Kotlin does not have a free-form condition-based for loops. The only form of a for-loop available in Kotlin is the "foreach" loop, which iterates over lists, arrays and other data structures.

A *for-loop statement* is a special kind of loop statement used to iterate over some data structure viewed as an iterable collection of elements. A for-loop statement consists of a loop body, a **container expression** and an **iteration variable declaration**.

The for-loop is actually an overloadable syntax form with the following expansion:

`for(VarDecl in C) Body` is the same as

```
when(val $iterator = C.iterator()) {
    else -> while ($iterator.hasNext()) {
                val VarDecl = __iterator.next()
                <... all the statements from Body>
            }
}
```

where `iterator`, `hasNext`, `next` are all suitable operator functions available in the current scope. `VarDecl` here may be a variable name or a set of variable names as per destructuring variable declarations.

Note: the expansion is hygienic, i.e., the generated iterator variable never clashes with any other variable in the program and cannot be accessed outside the expansion.

7.3 Code blocks

***block*:**
'{'
{*NL*}
statements
{*NL*}
'}'

***statements*:**
[*statement* {*semis statement*}] [*semis*]

A *code block* is a sequence of zero or more statements between curly braces separated by newlines or/and semicolons. Evaluating a code block means evaluating all its statements in the order they appear inside of it.

> Note: Kotlin does **not** support code blocks as statements; a curly-braces code block in a statement position is a lambda literal.

A *last expression* of a code block is the last statement in it (if any) if and only if this statement is also an expression. A code block is said to contain no last expression if it does not contain any statements or its last statement is not an expression (e.g., it is an assignment, a loop or a declaration).

> Informally: you may consider the case of a missing last expression as if a synthetic last expression with no runtime semantics and type `kotlin.Unit` is introduced in its place.

A *control structure body* is either a single statement or a code block. A *last expression* of a control structure body CSB is either the last expression of a code block (if CSB is a code block) or the single expression itself (if CSB is an expression). If a control structure body is not a code block or an expression, it has no last expression.

> Note: this is equivalent to wrapping the single expression in a new synthetic code block.

In some contexts, a control structure body is expected to have a value and/or a type. The value of a control structure body is:

- the value of its last expression if it exists;
- the singleton `kotlin.Unit` object otherwise.

The type of a control structure body is the type of its value.

7.3.1 Coercion to `kotlin.Unit`

When we expect the type of a control structure body to be `kotlin.Unit`, we relax the type checking requirements for its type by *coercing* it to `kotlin.Unit`. Specifically, we *ignore* the type mismatch between `kotlin.Unit` and the control structure body type.

> Examples:

```
fun foo() {
    val a /* : () -> Unit */ = {
        if (true) 42
        // CSB with no last expression
        // Type is defined to be `kotlin.Unit`
    }

    val b: () -> Unit = {
```

```
            if (true) 42 else -42
            // CSB with last expression of type `kotlin.Int`
            // Type is expected to be `kotlin.Unit`
            // Coercion to kotlin.Unit applied
        }
    }
```

Chapter 8

Expressions

Glossary

CSB
: Control structure body

Introduction

Expressions (together with statements) are one of the main building blocks of any program, as they represent ways to *compute* program values or *control* the program execution flow.

In Kotlin, an expression may be *used as a statement* or *used as an expression* depending on the context. As all expressions are valid statements, standalone expressions may be used as single statements or inside code blocks.

An expression is used as an expression, if it is encountered in any position where a statement is not allowed, for example, as an operand to an operator or as an immediate argument for a function call. An expression is used as a statement if it is encountered in any position where a statement is allowed.

Some expressions are allowed to be used as statements, only if certain restrictions are met; this may affect the semantics, the compile-time type information or/and the safety of these expressions.

8.1 Constant literals

Constant literals are expressions which describe constant values. Every constant

literal is defined to have a single standard library type, whichever it is defined to be on current platform. All constant literals are evaluated immediately.

8.1.1 Boolean literals

BooleanLiteral
`'true'` | `'false'`

Keywords `true` and `false` denote boolean literals of the same values. These are strong keywords which cannot be used as identifiers unless escaped. Values `true` and `false` always have the type `kotlin.Boolean`.

8.1.2 Integer literals

***IntegerLiteral*:**
DecDigitNoZero {*DecDigitOrSeparator*} *DecDigit*
| *DecDigit*

HexLiteral
`'0'` (`'x'` | `'X'`) *HexDigit* {*HexDigitOrSeparator*} *HexDigit*
| `'0'` (`'x'` | `'X'`) *HexDigit*

BinLiteral
`'0'` (`'b'` | `'B'`) *BinDigit* {*BinDigitOrSeparator*} *BinDigit*
| `'0'` (`'b'` | `'B'`) *BinDigit*

***DecDigitNoZero*:**
`'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`

***DecDigitOrSeparator*:**
DecDigit | `'_'`

***HexDigitOrSeparator*:**
HexDigit | `'_'`

BinDigitOrSeparator
BinDigit | `'_'`

***DecDigits*:**
DecDigit {*DecDigitOrSeparator*} *DecDigit*
| *DecDigit*

Decimal integer literals

A sequence of decimal digit symbols (`0` though `9`) is a decimal integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

Note: unlike other languages, Kotlin does not support octal literals. Even more so, any decimal literal starting with digit `0` and containing more than 1 digit is not a valid decimal literal.

Hexadecimal integer literals

A sequence of hexadecimal digit symbols (`0` through `9`, `a` through `f`, `A` through `F`) prefixed by `0x` or `0X` is a hexadecimal integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

Binary integer literals

A sequence of binary digit symbols (`0` or `1`) prefixed by `0b` or `0B` is a binary integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

8.1.3 The types for integer literals

Any of the decimal, hexadecimal or binary literals may be suffixed by the long literal mark (symbol `L`). An integer literal with the long literal mark has type `kotlin.Long`. A literal without the mark has a special integer literal type dependent on the value of the literal:

- If the value is greater than maximum `kotlin.Long` value (see built-in integer types), it is an illegal integer literal and should be a compile-time error;
- Otherwise, if the value is greater than maximum `kotlin.Int` value (see built-in integer types), it has type `kotlin.Long`;
- Otherwise, it has an integer literal type containing all the built-in integer types guaranteed to be able to represent this value.

  Example: integer literal `0x01` has value 1 and therefore has type ILT(`kotlin.Byte`, `kotlin.Short`, `kotlin.Int`, `kotlin.Long`). Integer literal `70000` has value 70000, which is not representable using types `kotlin.Byte` and `kotlin.Short` and therefore has type ILT(`kotlin.Int`, `kotlin.Long`).

8.1.4 Real literals

***RealLiteral*:**
FloatLiteral | *DoubleLiteral*

***FloatLiteral*:**
DoubleLiteral (`'f'` | `'F'`)
| *DecDigits* (`'f'` | `'F'`)

***DoubleLiteral*:**
[*DecDigits*] `'.'` *DecDigits* [*DoubleExponent*]
| [*DecDigits*] [*DoubleExponent*]

A *real literal* consists of the following parts: the whole-number part, the decimal point (ASCII period character `.`), the fraction part and the exponent. Unlike other languages, Kotlin real literals may only be expressed in decimal numbers. A real literal may also be followed by a type suffix (`f` or `F`).

The exponent is an exponent mark (`e` or `E`) followed by an optionally signed decimal integer (a sequence of decimal digits).

The whole-number part and the exponent part may be omitted. The fraction part may be omitted only together with the decimal point, if the whole-number part and either the exponent part or the type suffix are present. Unlike other languages, Kotlin does not support omitting the fraction part, but leaving the decimal point in.

The digits of the whole-number part or the fraction part or the exponent may be optionally separated by underscores, but an underscore may not be placed between, before, or after these parts. It also may not be placed before or after the exponent mark symbol.

A real literal without the type suffix has type `kotlin.Double`, a real literal with the type suffix has type `kotlin.Float`.

> Note: this means there is no special suffix associated with type `kotlin.Double`.

8.1.5 Character literals

CharacterLiteral
`'''` (*EscapeSeq* | <any character excluding CR, LF, `'''` or `'\'`>) `'''`

EscapeSeq
UniCharacterLiteral | *EscapedIdentifier*

UniCharacterLiteral
`'\' 'u'` *HexDigit HexDigit HexDigit HexDigit*

EscapedIdentifier
`'\'` (`'t'` | `'b'` | `'r'` | `'n'` | `'''` | `'"'` | `'\'` | `'$'`)

A *character literal* defines a constant holding a Unicode character value. A simply-formed character literal is any symbol between two single quotation marks (ASCII single quotation character `'`), excluding newline symbols (*CR* and

LF), the single quotation mark itself and the escaping mark (ASCII backslash character \).

All character literals have type `kotlin.Char`.

Escaped characters

A character literal may also contain an escaped symbol of two kinds: a simple escaped symbol or a Unicode codepoint. Simple escaped symbols include:

- `\t` — the Unicode TAB symbol (U+0009);
- `\b` — the Unicode BACKSPACE symbol (U+0008);
- `\r` — *CR*;
- `\n` — *LF*;
- `\'` — the Unicode apostrophe symbol (U+0027);
- `\"` — the Unicode double quotation symbol (U+0028);
- `\\` — the Unicode backslash symbol (U+005C);
- `\$` — the Unicode DOLLAR sign (U+0024).

A Unicode codepoint escaped symbol is the symbol `\u` followed by exactly four hexadecimal digits. It represents the Unicode symbol with the codepoint equal to the number represented by these four digits.

> Note: this means Unicode codepoint escaped symbols support only Unicode symbols in range from U+0000 to U+FFFF.

8.1.6 String literals

Kotlin supports string interpolation which supersedes traditional string literals. For further details, please refer to the corresponding section.

8.1.7 Null literal

The keyword `null` denotes the **null reference**, which represents an absence of a value and is a valid value only for nullable types. Null reference has type `kotlin.Nothing?` and is, by definition, the only value of this type.

8.2 String interpolation expressions

***stringLiteral*:**
lineStringLiteral
| *multiLineStringLiteral*

***lineStringLiteral*:**
    '"' {*lineStringContent* | *lineStringExpression*} '"'

***multiLineStringLiteral*:**
    '"""' {*multiLineStringContent* | *multiLineStringExpression* | '"'}
    TRIPLE_QUOTE_CLOSE

***lineStringContent*:**
    LineStrText
    | *LineStrEscapedChar*
    | *LineStrRef*

***lineStringExpression*:**
    '${'
    {*NL*}
    expression
    {*NL*}
    '}'

***multiLineStringContent*:**
    MultiLineStrText
    | '"'
    | *MultiLineStrRef*

***multiLineStringExpression*:**
    '${'
    {*NL*}
    expression
    {*NL*}
    '}'

String interpolation expressions replace the traditional string literals and supersede them. A string interpolation expression consists of one or more fragments of two different kinds: string content fragments (raw pieces of string content inside the quoted literal) and *interpolated expression fragments*, specified by a special syntax using the `$` symbol.

Interpolated expressions support two different forms.

- `$id`, where `id` is a simple path available in the current scope;
- `${e}`, where `e` is a valid Kotlin expression.

  Note: the first form requires `id` to be a simple path; if you want to reference a qualified path (e.g., `foo.bar`), you should use the second form as `${foo.bar}`.

In either case, the interpolated value is evaluated and converted into `kotlin.String` by a process defined below. The resulting value of a string interpolation expression is the concatenation of all fragments in the expression.

An interpolated value v is converted to `kotlin.String` according to the following convention:

- If it is equal to the null reference, the result is `"null"`;
- Otherwise, the result is v`.toString()` where `toString` is the `kotlin.Any` member function (no overloading resolution is performed to choose this function in this context).

There are two kinds of string interpolation expressions: line interpolation expressions and multiline (or raw) interpolation expressions. The difference is that some symbols (namely, newline symbols) are not allowed to be used inside line interpolation expressions and they need to be escaped in the same way they are escaped in character literals. On the other hand, multiline interpolation expressions allow such symbols inside them, but do not allow single character escaping of any kind.

> Note: among other things, this means that escaping of the `$` symbol is impossible in multiline strings. If you need an escaped `$` symbol, use an interpolated expression `"${'$'}"` instead.

String interpolation expression always has type `kotlin.String`.

8.3 Try-expressions

***tryExpression*:**
`'try'` {*NL*} *block* ((({*NL*} *catchBlock* {{*NL*} *catchBlock*}) [{*NL*} *finallyBlock*]) | ({*NL*} *finallyBlock*))

***catchBlock*:**
`'catch'`
{*NL*}
`'('`
{*annotation*}
simpleIdentifier
`':'`
type
[{*NL*} `','`]
`')'`
{*NL*}
block

***finallyBlock*:**
`'finally'` {*NL*} *block*

A *try-expression* is an expression starting with the keyword `try`. It consists of a code block (*try body*) and one or more of the following kinds of blocks: zero or more *catch blocks* and an optional *finally block*. A *catch block* starts with the soft

keyword `catch` with a single *exception parameter*, which is followed by a code block. A *finally block* starts with the soft keyword `finally`, which is followed by a code block. A valid try-expression must have at least one catch or finally block.

The try-expression evaluation evaluates its body; if any statement in the try body throws an exception (of type E), this exception, rather than being immediately propagated up the call stack, is checked for a matching catch block. If a catch block of this try-expression has an exception parameter of type $T :> E$, this catch block is evaluated immediately after the exception is thrown and the exception itself is passed inside the catch block as the corresponding parameter. If there are several catch blocks which match the exception type, the first one is picked.

For an in-detail explanation on how exceptions and catch-blocks work, please refer to the Exceptions section. For a low-level explanation, please refer to the platform-specific parts of this document.

If there is a finally block, it is evaluated after the evaluation of all previous try-expression blocks, meaning:

- If no exception is thrown during the evaluation of the try body, no catch blocks are executed, the finally block is evaluated after the try body, and the program execution continues as normal.
- If an exception was thrown, and one of the catch blocks matched its type, the finally block is evaluated after the evaluation of the matching catch block.
- If an exception was thrown, but no catch block matched its type, the finally block is evaluated before propagating the exception up the call stack.

The value of the try-expression is the same as the value of the last expression of the try body (if no exception was thrown) or the value of the last expression of the matching catch block (if an exception was thrown and matched). All other situations mean that an exception is going to be propagated up the call stack, and the value of the try-expression is undefined.

> Note: as described, the finally block (if present) is always executed, but has no effect on the value of the try-expression.

The type of the try-expression is the least upper bound of the types of the last expressions of the try body and the last expressions of all the catch blocks.

> Note: these rules mean the try-expression always may be used as an expression, as it always has a corresponding result value.

8.4 Conditional expressions

***ifExpression*:**

'if'
{*NL*}
'('
{*NL*}
expression
{*NL*}
')'
{*NL*}
(*controlStructureBody* | ([*controlStructureBody*] {*NL*} [';'] {*NL*} 'else'
{*NL*} (*controlStructureBody* | ';')) | ';')

Conditional expressions use a boolean value of one expression (*condition*) to decide which of the two control structure bodies (*branches*) should be evaluated. If the condition evaluates to `true`, the first branch (the *true branch*) is evaluated if it is present, otherwise the second branch (the *false branch*) is evaluated if it is present.

> Note: this means the following branchless conditional expression, despite being of almost no practical use, is valid in Kotlin
>
> ```
> if (condition) else;
> ```

The value of the resulting expression is the same as the value of the chosen branch.

The type of the resulting expression is the least upper bound of the types of two branches, if both branches are present. If either of the branches are omitted, the resulting conditional expression has type `kotlin.Unit` and may be used only as a statement.

> Example:
>
> ```
> // x has type kotlin.Int and value 1
> val x = if (true) 1 else 2
> // illegal, as if expression without false branch
> //   cannot be used as an expression
> val y = if (true) 1
> ```

The type of the condition expression must be a subtype of `kotlin.Boolean`, otherwise it is a compile-time error.

> Note: when used as expressions, conditional expressions are special w.r.t. operator precedence: they have the highest priority (the same as for all primary expressions) when placed on the right side of any binary expression, but when placed on the left side, they have the lowest priority. For details, see Kotlin grammar.
>
> Example:
>
> ```
> x = if (true) 1 else 2
> ```
>
> is the same as

```
x = (if (true) 1 else 2)
```

At the same time

```
if (true) x = 1 else x = 2
```

is the same as

```
if (true) (x = 1) else (x = 2)
```

8.5 When expressions

***whenExpression*:**
'when'
{*NL*}
[*whenSubject*]
{*NL*}
'{'
{*NL*}
{*whenEntry* {*NL*}}
{*NL*}
'}'

***whenEntry*:**
(*whenCondition* {{*NL*} ',' {*NL*} *whenCondition*} [{*NL*} ','] {*NL*} '->'
{*NL*} *controlStructureBody* [*semi*])
| ('else' {*NL*} '->' {*NL*} *controlStructureBody* [*semi*])

***whenCondition*:**
expression
| *rangeTest*
| *typeTest*

***rangeTest*:**
inOperator {*NL*} *expression*

***typeTest*:**
isOperator {*NL*} *type*

When expression is similar to a conditional expression in that it allows one of several different control structure bodies (*cases*) to be evaluated, depending on some boolean conditions. The key difference is that a when expressions may include *several* different conditions with their corresponding control structure bodies. When expression has two different forms: with bound value and without it.

When expression without bound value (the form where the expression enclosed in parentheses after the `when` keyword is absent) evaluates one of the different CSBs based on its condition from the *when entry*. Each when entry

consists of a boolean *condition* (or a special `else` condition) and its corresponding CSB. When entries are checked and evaluated in their order of appearance. If the condition evaluates to `true`, the corresponding CSB is evaluated and the value of when expression is the same as the value of the CSB. All remaining conditions and expressions are not evaluated.

The `else` condition is a special condition which evaluates to `true` if none of the branches above it evaluated to `true`. The `else` condition **must** also be in the last when entry of when expression, otherwise it is a compile-time error.

> Note: informally, you can always replace the `else` condition with an always-`true` condition (e.g., boolean literal `true`) with no changes to the result of when expression.

When expression with bound value (the form where the expression enclosed in parentheses after the `when` keyword is present) is similar to the form without bound value, but uses a different syntax and semantics for conditions. In fact, it supports four different condition forms:

- *Type test condition*: type checking operator followed by a type (`is T` or `!is T`). The resulting condition is a type check expression of the form `boundValue is T` or `boundValue !is T`.
- *Contains test condition*: containment operator followed by an expression (`in Expr` or `!in Expr`). The resulting condition is a containment check expression of the form `boundValue in Expr` or `boundValue !in Expr`.
- *Any other applicable expression* (`Expr`) The resulting condition is an equality check of the form `boundValue == Expr`.
- The `else` condition, which is a special condition which evaluates to `true` if none of the branches above it evaluated to `true`. The `else` condition **must** also be in the last when entry of when expression, otherwise it is a compile-time error.

  > Note: the rule for "any other expression" means that if a when expression with bound value contains a boolean condition, this condition is **checked for equality** with the bound value, instead of being used directly for when entry selection.

  > Note: in Kotlin version 1.3 and earlier, simple (unlabeled) `break` and `continue` expressions were disallowed in when expressions.

The type of the resulting when expression is the least upper bound of the types of all its entries. If when expression is not exhaustive, it has type `kotlin.Unit` and may be used only as a statement.

> Examples:

```
val a = 42
val b = -1

when {
```

```
  a == b -> {}
  a != b -> {}
}

// Error, as it is a non-exhaustive when expression
val c = when {
  a == b -> {}
  a != b -> {}
}

val d = when {
  a == b -> {}
  a != b -> {}
  else -> {}
}

when {
  a == b || a != b -> {}
  42 > 0 -> {}
}

val a = 42
val b = -1

val l = (1..10).toList()

when (a) {
  is Int, !is Int -> {}
  in l, !in l -> {}
}

// Error, as it is a non-exhaustive when expression
val c = when (a) {
  is Int, !is Int -> {}
  in l, !in l -> {}
}

val d = when (a) {
  is Int, !is Int -> {}
  in l, !in l -> {}
  else -> {}
}
```

When with bound value also allows for an inline property declaration of the form `when (val V = E) { ... }` inside the parentheses. This declares a new property (see declaration sections for details) alongside the usual mechanics of the *when-expression*. The scope of this property is limited to the `when` expression,

including both conditions and control structure bodies of the expression. As its form is limited to a simple "assignment-like" declaration with an initializer, this property does not allow getters, setters, delegation or destructuring. It is also required to be immutable. Conceptually, it is very similar to declaring such a property before the when-expression and using it as subject, but with a difference in scoping of this property described above.

Example:

```
when(val a = b + c) {
    !is Foo -> a + 1
    else -> b
}

val y = a // illegal, a is not visible here anymore
```

8.5.1 Exhaustive when expressions

A when expression is called ***exhaustive*** if at least one of the following is true:

- It has an `else` entry;
- It has a bound value and at least one of the following is true:
  - The bound expression is of type `kotlin.Boolean` and the conditions contain both:
    - A constant expression evaluating to `true`;
    - A constant expression evaluating to `false`;

      Important: here the term "constant expression" refers to any expression constructed of constant literals, string interpolation over constant expressions and an implementation-defined set of functions that may always be evaluated at compile-time
  - The bound expression is of a `sealed` class or interface and all of its *direct non-sealed subtypes* $T_1, \ldots, T_n$ are covered in this expression. A subtype T_i is considered covered if when expression contains one of the following:
    - a type test condition isT_i;
    - a type test condition isS_i (where S_i is sealed and $T_i <: S_i$);
    - a type test condition $!isS_j$ (where S_j is sealed and $\exists j \neq i : T_j <: S_j$.

    Additionally, an enum subtype E_i is considered covered also if all its enumerated values are checked for equality using constant expression;
  - The bound expression is of an `enum class` type and all its enumerated values are checked for equality using constant expression;
  - The bound expression is of a nullable type T? and one of the cases above is met for its non-nullable counterpart T together with another condition which checks the bound value for equality with `null`.

For object types, the type test condition may be replaced with equality check with the object value.

> Note: if one were to override `equals` for an object type incorrectly (i.e., so that an object is not equal to itself), it would break the exhaustiveness check. It is unspecified whether this situation leads to an exception or an undefined value for this `when` expression.

```
sealed class Base
class Derived1: Base()
object Derived2: Base()

val b: Base = ...

val c = when(b) {
    is Derived1 -> ...
    Derived2 -> ...
    // no else needed here
}
```

```
sealed interface I1
sealed interface I2
sealed interface I3

class D1 : I1, I2
class D2 : I1, I3

sealed class D3 : I1, I3

fun foo() {
    val b: I1 = mk()

    val c = when(a) {
        !is I3 -> {} // covers D1
        is D2 -> {} // covers D2
        // D3 is sealed and does not take part
        // in the exhaustiveness check
    }
}
```

> Informally: an exhaustive when expression is guaranteed to evaluate one of its CSBs regardless of the specific when conditions.

8.6 Logical disjunction expressions

***disjunction*:**

conjunction {{*NL*} `'||'` {*NL*} *conjunction*}

Operator symbol `||` performs logical disjunction over two values of type `kotlin.Boolean`. This operator is **lazy**, meaning that it does not evaluate the right hand side argument unless the left hand side argument evaluated to `false`.

Both operands of a logical disjunction expression must have a type which is a subtype of `kotlin.Boolean`, otherwise it is a compile-time error. The type of logical disjunction expression is `kotlin.Boolean`.

8.7 Logical conjunction expressions

***conjunction*:**
equality {{*NL*} `'&&'` {*NL*} *equality*}

Operator symbol `&&` performs logical conjunction over two values of type `kotlin.Boolean`. This operator is **lazy**, meaning that it does not evaluate the right hand side argument unless the left hand side argument evaluated to `true`.

Both operands of a logical conjunction expression must have a type which is a subtype of `kotlin.Boolean`, otherwise it is a compile-time error. The type of logical disjunction expression is `kotlin.Boolean`.

8.8 Equality expressions

***equality*:**
comparison {*equalityOperator* {*NL*} *comparison*}

***equalityOperator*:**
`'!='`
| `'!=='`
| `'=='`
| `'==='`

Equality expressions are binary expressions involving equality operators. There are two kinds of equality operators: *reference equality operators* and *value equality operators*.

8.8.1 Reference equality expressions

Reference equality expressions are binary expressions which use reference equality operators: `===` and `!==`. These expressions check if two values are equal (`===`) or non-equal (`!==`) *by reference*: two values are equal by reference if and only if

they represent the same runtime value. In particular, this means that two values acquired by the same constructor call are equal by reference, while two values created by two different constructor calls are not equal by reference. A value created by any constructor call is never equal by reference to a null reference.

There is an exception to these rules: values of value classes are not guaranteed to be reference equal even if they are created by the same constructor invocation as said constructor invocation is explicitly allowed to be inlined by the compiler. It is thus highly discouraged to compare inline classes by reference.

For special values created without explicit constructor calls, notably, constant literals and constant expressions composed of those literals, and for values of inline classes, the following holds:

- If these values are non-equal by value, they are also non-equal by reference;
- Any instance of the null reference `null` is equal by reference to any other instance of the null reference;
- Otherwise, equality by reference is implementation-defined and should not be used as a means of comparing such values.

Reference equality expressions always have type `kotlin.Boolean`.

Kotlin checks the applicability of reference equality operators at compile-time and may reject certain combinations of types for `A` and `B`. Specifically, it uses the following basic principle.

> If type of `A` and type of `B` are definitely distinct and not related by subtyping, `A === B` is an invalid expression and should result in a compile-time error.
>
> Informally: this principle means "no two objects of different types can be equal by reference".

8.8.2 Value equality expressions

Value equality expressions are binary expressions which use value equality operators: `==` and `!=`. These operators are overloadable, but are different from other overloadable operators and have the following expansion:

- `A == B` is exactly the same as `(A as? Any)?.equals(B) ?: (B === null)` where `equals` is the method of `kotlin.Any`;
- `A != B` is exactly the same as `!((A as? Any)?.equals(B) ?: (B === null))` where `equals` is the method of `kotlin.Any`.

Value equality expressions always have type `kotlin.Boolean` as does the `equals` method in `kotlin.Any`.

Kotlin checks the applicability of value equality operators at compile-time and may reject certain combinations of types for `A` and `B`. Specifically, it uses the following basic principle.

If type of `A` and type of `B` are definitely distinct and not related by subtyping, `A == B` is an invalid expression and should result in a compile-time error.

Informally: this principle means "no two objects unrelated by subtyping can ever be considered equal by `==`".

8.9 Comparison expressions

comparison:
genericCallLikeComparison {*comparisonOperator* {*NL*} *genericCallLikeComparison*}

comparisonOperator:
```
'<'
| '>'
| '<='
| '>='
```

Comparison expressions are binary expressions which use the comparison operators: `<`, `>`, `<=` and `>=`. These operators are overloadable with the following expansion:

- `A < B` is exactly the same as `integerLess(A.compareTo(B), 0)`
- `A > B` is exactly the same as `integerLess(0, A.compareTo(B))`
- `A <= B` is exactly the same as `!integerLess(0, A.compareTo(B))`
- `A >= B` is exactly the same as `!integerLess(A.compareTo(B), 0)`

where `compareTo` is a valid operator function available in the current scope and `integerLess` is a special intrinsic function unavailable in user-side Kotlin which performs integer "less-than" comparison of two integer numbers.

The `compareTo` operator function must have return type `kotlin.Int`, otherwise such declaration is a compile-time error.

All comparison expressions always have type `kotlin.Boolean`.

8.10 Type-checking and containment-checking expressions

infixOperation:
elvisExpression {(*inOperator* {*NL*} *elvisExpression*) | (*isOperator* {*NL*} *type*)}

inOperator:
    `'in'`
    | *NOT_IN*

isOperator:
    `'is'`
    | *NOT_IS*

8.10.1 Type-checking expressions

A type-checking expression uses a type-checking operator `is` or `!is` and has an expression E as a left-hand side operand and a type name T as a right-hand side operand. A type-checking expression checks whether the runtime type of E is a subtype of T for `is` operator, or not a subtype of T for `!is` operator.

The type T must be runtime-available, otherwise it is a compile-time error.

If the type T is not a parameterized type, it must be runtime-available, otherwise it is a compile-time error. If T is a parameterized type, the bare type argument inference is performed for the compile-time known type of E and the type constructor TC of T. After that, given the result arguments of this bare type inference $A_0, A_1 \dots A_N$, T must suffice the constraint $T!! <: TC[A_0, A_1 \dots A_N]$, checking each of its argument for conformance with the type of E.

Example:

```
interface Foo<A, B>
class Fee<T, U>: Foo<U, T>

fun f(foo: Foo<String, Int>) {
    // valid: you can specify parameters
    // as long as they correspond to base type
    if(foo is Fee<Int, String>) { ... }
    // invalid: Fee<String, Int> is not a subtype
    // of Foo<String, Int>
    if(foo is Fee<String, Int>) { ... }
    // valid: may be specified partially
    if(foo is Fee<Int, *>) { ... }
```

T may also be specified without arguments by using the *bare type syntax* in which case the same process of bare type argument inference is performed, with the difference being that the resulting arguments $A_0, A_1 \dots A_N$ are used as arguments for T directly. If any of these arguments are inferred to be star-projections, this is a compile-time error.

Example:

```
interface Foo<A, B>
class Fee<T, U>: Foo<U, T>
```

```
fun f(foo: Foo<String, Int>) {
    // valid: same as foo is Fee<Int, String>
    if(foo is Fee) { ... }
```

Type-checking expression always has type `kotlin.Boolean`.

> Note: the expression `null is T?` for any type `T` always evaluates to `true`, as the type of the left-hand side (`null`) is `kotlin.Nothing?`, which is a subtype of any nullable type `T?`.
>
> Note: type-checking expressions may create smart casts, for further details, refer to the corresponding section.

8.10.2 Containment-checking expressions

A *containment-checking expression* is a binary expression which uses a containment operator `in` or `!in`. These operators are overloadable with the following expansion:

- `A in B` is exactly the same as `B.contains(A)`;
- `A !in B` is exactly the same as `!(B.contains(A))`.

where `contains` is a valid operator function available in the current scope.

> Note: this means that, contrary to the order of appearance in the code, the right-hand side expression of a containment-checking expression is evaluated before its left-hand side expression

The `contains` function must have a return type `kotlin.Boolean`, otherwise it is a compile-time error. Containment-checking expressions always have type `kotlin.Boolean`.

8.11 Elvis operator expressions

***elvisExpression*:**
infixFunctionCall {{*NL*} *elvis* {*NL*} *infixFunctionCall*}

An *elvis operator expression* is a binary expression which uses an elvis operator (`?:`). It checks whether the left-hand side expression is reference equal to `null`, and, if it is, evaluates and return the right-hand side expression.

This operator is **lazy**, meaning that if the left-hand side expression is not reference equal to `null`, the right-hand side expression is not evaluated.

The type of elvis operator expression is the least upper bound of the non-nullable variant of the type of the left-hand side expression and the type of the right-hand side expression.

8.12 Range expressions

***rangeExpression*:**
additiveExpression {'..' {*NL*} *additiveExpression*}

A *range expression* is a binary expression which uses a range operator `..` It is an overloadable operator with the following expansion:

- `A..B` is exactly the same as `A.rangeTo(B)`

where `rangeTo` is a valid operator function available in the current scope.

The return type of this function is not restricted. A range expression has the same type as the return type of the corresponding `rangeTo` overload variant.

8.13 Additive expressions

***additiveExpression*:**
multiplicativeExpression {*additiveOperator* {*NL*} *multiplicativeExpression*}

***additiveOperator*:**
'+'
| '-'

An *additive expression* is a binary expression which uses an addition (`+`) or subtraction (`-`) operators. These are overloadable operators with the following expansions:

- `A + B` is exactly the same as `A.plus(B)`
- `A - B` is exactly the same as `A.minus(B)`

where `plus` or `minus` is a valid operator function available in the current scope.

The return type of these functions is not restricted. An additive expression has the same type as the return type of the corresponding operator function overload variant.

8.14 Multiplicative expressions

***multiplicativeExpression*:**
asExpression {*multiplicativeOperator* {*NL*} *asExpression*}

***multiplicativeOperator*:**
'*'
| '/'
| '%'

A *multiplicative expression* is a binary expression which uses a multiplication (`*`), division (`/`) or remainder (`%`) operators. These are overloadable operators with the following expansions:

- `A * B` is exactly the same as `A.times(B)`
- `A / B` is exactly the same as `A.div(B)`
- `A % B` is exactly the same as `A.rem(B)`

where `times`, `div`, `rem` is a valid operator function available in the current scope.

> Note: in Kotlin version 1.3 and earlier, there was an additional overloadable operator for `%` called `mod`, which has been removed in Kotlin 1.4.

The return type of these functions is not restricted. A multiplicative expression has the same type as the return type of the corresponding operator function overload variant.

8.15 Cast expressions

***asExpression*:**
prefixUnaryExpression {{*NL*} *asOperator* {*NL*} *type*}

***asOperator*:**
`'as'`
| `'as?'`

A *cast expression* is a binary expression which uses cast operators `as` or `as?` and has the form `E as/as? T`, where E is an expression and T is a type name.

An **`as` cast expression** `E as T` is called *an unchecked cast* expression. This expression perform a runtime check whether the runtime type of E is a subtype of T and throws an exception otherwise. If type T is a runtime-available type without generic parameters, then this exception is thrown immediately when evaluating the cast expression, otherwise it is implementation-defined whether an exception is thrown at this point.

An unchecked cast expression result always has the same type as the type T specified in the expression.

An **`as?` cast expression** `E as? T` is called *a checked cast* expression. This expression is similar to the unchecked cast expression in that it also does a runtime type check, but does not throw an exception if the types do not match, it returns `null` instead. If type T is not a runtime-available type, then the check is not performed and `null` is never returned, leading to potential runtime errors later in the program execution. This situation should be reported as a compile-time warning.

If type T is a runtime-available type **with** generic parameters, type parameters are **not** checked w.r.t. subtyping. This is another potentially erroneous situation, which should be reported as a compile-time warning.

Similarly to type checking expressions, some type arguments may be excluded from this check if they are known from the supertype of E and, if all type arguments of T can be inferred, they may be omitted altogether using the *bare type syntax*. See type checking section for explanation.

The checked cast expression result has the type which is the nullable variant of the type T specified in the expression.

> Note: cast expressions may create smart casts, for further details, refer to the corresponding section.

8.16 Prefix expressions

***prefixUnaryExpression*:**
{*unaryPrefix*} *postfixUnaryExpression*

***unaryPrefix*:**
annotation
| *label*
| (*prefixUnaryOperator* {*NL*})

***prefixUnaryOperator*:**
`'++'`
| `'--'`
| `'-'`
| `'+'`
| *excl*

8.16.1 Annotated expressions

Any expression in Kotlin may be prefixed with any number of annotations. These do not change the value of the expression and can be used by external tools and for implementing platform-dependent features. See annotations chapter of this document for further information and examples of annotations.

8.16.2 Prefix increment expressions

A *prefix increment* expression is an expression which uses the prefix form of operator `++`. It is an overloadable operator with the following expansion:

- `++A` is exactly the same as `when(val $tmp = A.inc()) { else -> A = $tmp; $tmp }` where `inc` is a valid operator function available in the current scope.

  Informally: `++A` assigns the result of `A.inc()` to `A` and also returns it as the result.

For a prefix increment expression `++A` expression `A` must be an assignable expression. Otherwise, it is a compile-time error.

As the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

A prefix increment expression has the same type as the return type of the corresponding `inc` overload variant.

8.16.3 Prefix decrement expressions

A *prefix decrement* expression is an expression which uses the prefix form of operator `--`. It is an overloadable operator with the following expansion:

- `--A` is exactly the same as `when(val $tmp = A.dec()) { else -> A = $tmp; $tmp }` where `dec` is a valid operator function available in the current scope.

  Informally: `--A` assigns the result of `A.dec()` to `A` and also returns it as the result.

For a prefix decrement expression `--A` expression `A` must be an assignable expression. Otherwise, it is a compile-time error.

As the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

A prefix decrement expression has the same type as the return type of the corresponding `dec` overload variant.

8.16.4 Unary minus expressions

An *unary minus* expression is an expression which uses the prefix form of operator `-`. It is an overloadable operator with the following expansion:

- `-A` is exactly the same as `A.unaryMinus()` where `unaryMinus` is a valid operator function available in the current scope.

No additional restrictions apply.

8.16.5 Unary plus expressions

An *unary plus* expression is an expression which uses the prefix form of operator `+`. It is an overloadable operator with the following expansion:

- `+A` is exactly the same as `A.unaryPlus()` where `unaryPlus` is a valid operator function available in the current scope.

No additional restrictions apply.

8.16.6 Logical not expressions

A *logical not* expression is an expression which uses the prefix operator `!`. It is an overloadable operator with the following expansion:

- `!A` is exactly the same as `A.not()` where `not` is a valid operator function available in the current scope.

No additional restrictions apply.

8.17 Postfix operator expressions

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***postfixUnaryOperator*:**
`'++'`
| `'--'`
| (`'!'` *excl*)

8.17.1 Postfix increment expressions

A *postfix increment* expression is an expression which uses the postfix form of operator `++`. It is an overloadable operator with the following expansion:

- `A++` is exactly the same as `when(val $tmp = A) { else -> A = $tmp.inc(); $tmp }` where `inc` is a valid operator function available in the current scope.

Informally: `A++` stores the value of A to a temporary variable, assigns the result of `A.inc()` to `A` and then returns the temporary variable as the result.

For a postfix increment expression `A++` expression `A` must be assignable expressions. Otherwise, it is a compile-time error.

As the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

A postfix increment expression has the same type as its operand expression (for our examples, the type of `A`).

8.17.2 Postfix decrement expressions

A *postfix decrement* expression is an expression which uses the postfix form of operator `--`. It is an overloadable operator with the following expansion:

- `A--` is exactly the same as `when(val $tmp = A) { else -> A = $tmp.dec(); $tmp }` where `dec` is a valid operator function available in the current scope.

  Informally: `A--` stores the value of A to a temporary variable, assigns the result of `A.dec()` to `A` and then returns the temporary variable as the result.

For a postfix decrement expression `A--` expression `A` must be assignable expressions. Otherwise, it is a compile-time error.

As the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

A postfix decrement expression has the same type as its operand expression (for our examples, the type of `A`).

8.18 Not-null assertion expressions

A *not-null assertion expression* is a postfix expression which uses an operator `!!`. For an expression `e!!`, if the type of `e` is nullable, a not-null assertion expression checks whether the evaluation result of `e` is equal to `null` and, if it is, throws a runtime exception. If the evaluation result of `e` is not equal to `null`, the result of `e!!` is the evaluation result of `e`.

If the type of `e` is non-nullable, not-null assertion expression `e!!` has no effect.

The type of non-null assertion expression is the non-nullable variant of the type of `e`.

Note: this type may be non-denotable in Kotlin and, as such, may be approximated in some situations with the help of type inference.

8.19 Indexing expressions

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***indexingSuffix*:**
`'['`
{*NL*}
expression
{{*NL*} `','` {*NL*} *expression*}
[{*NL*} `','`]
{*NL*}
`']'`

An *indexing expression* is a suffix expression which uses one or more subexpressions as *indices* between square brackets (`[` and `]`).

It is an overloadable operator with the following expansion:

- `A[I_0,I_1,...,I_N]` is exactly the same as `A.get(I_0,I_1,...,I_N)`, where `get` is a valid operator function available in the current scope.

An indexing expression has the same type as the corresponding `get` expression.

Indexing expressions are assignable, for a corresponding assignment form, see here.

8.20 Call and property access expressions

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*

| *indexingSuffix*
| *navigationSuffix*

***navigationSuffix*:**
memberAccessOperator {*NL*} (*simpleIdentifier* | *parenthesizedExpression* | `'class'`)

***callSuffix*:**
[*typeArguments*] (([*valueArguments*] *annotatedLambda*) | *valueArguments*)

***annotatedLambda*:**
{*annotation*} [*label*] {*NL*} *lambdaLiteral*

***valueArguments*:**
`'('` {*NL*} [*valueArgument* {{*NL*} `','` {*NL*} *valueArgument*} [{*NL*} `','`] {*NL*}] `')'`

***typeArguments*:**
`'<'`
{*NL*}
typeProjection
{{*NL*} `','` {*NL*} *typeProjection*}
[{*NL*} `','`]
{*NL*}
`'>'`

***typeProjection*:**
([*typeProjectionModifiers*] *type*)
| `'*'`

***typeProjectionModifiers*:**
typeProjectionModifier {*typeProjectionModifier*}

***memberAccessOperator*:**
({*NL*} `'.'`)
| ({*NL*} *safeNav*)
| `'::'`

8.20.1 Navigation operators

Expressions which use the navigation binary operators (`.`, `?.` or `::`) are syntactically similar, but, in fact, may have very different semantics.

`a.c` may have one of the following semantics when used as an expression:

- A fully-qualified type, property or object name. The left side of `.` must be a value available in the current scope, while the right side corresponds to a declaration in the scope of that value.

  Note: qualification uses operator `.` only.

- A property access. Here `a` is a value available in the current scope and `c` is a property name.

  Note: the navigation operator `.` is closely related to the concept of paths.

If followed by the call suffix (arguments in parentheses), `a.c()` may have one of the following semantics when used as an expression:

- A function call; here `a` is a value available in the current scope and `c` is a function name;
- A property access with `invoke`-convention; here `a` is a value available in the current scope and `c` is a property name.

  These expressions follow the overloading rules.

`a::c` may have one of the following semantics when used as an expression:

- A class literal expression if, instead of an identifier, `c` is the keyword `class`;
- A property reference. Here `a` may be either a value available in the current scope or a type name, and `c` is a property name.
- A function reference. Here `a` may be either a value available in the current scope or a type name, and `c` is a function name.

`a?.c` is a *safe navigation* operator, which has the following expansion:

- `a?.c` is exactly the same as

  ```
  when (val $tmp = a) {
      null -> null
      else -> { $tmp.c }
  }
  ```

  for any right-hand combinations of operators present in `c`, which are expanded further, as usual.

The type of `a?.c` is the nullable variant of the type of `a.c`.

Note: safe navigation expression may also include the call suffix as `a?.c()`and is expanded in a similar fashion.

8.20.2 Callable references

Callable references are a special kind of expressions used to refer to callables (properties and functions) without actually calling/accessing them. They are not to be confused with class literals which use similar syntax, but with the keyword `class` instead of an identifier.

A callable reference `A::c` where `A` is a type name and `c` is a name of a callable available for type `A` is a *callable reference* for type `A`. A callable reference `e::c` where `e` is an expression of type `E` and `c` is a name of a callable available for

type `E` is a *callable reference* for expression `e`. The exact callable selected when using this syntax is based on overload resolution much like when accessing the value of a property using the `.` navigation operator. However, in some cases there are important differences which we cover in the corresponding paragraphs.

Depending on the meaning of the left-hand and right-hand sides of a callable reference `lhs::rhs`, the value of the whole expression is defined as follows.

- If `lhs` is a type, but not a value (an example of a type which can also be used as a value is an object type), while `rhs` is resolved to refer to a property of `lhs`, `lhs::rhs` is a *type-property* reference;
- If `lhs` is a type, but not a value (an example of a type which can also be used as a value is an object type), while `rhs` is resolved to refer to a function available on `rhs`, `lhs::rhs` is a *type-function* reference;
- If `lhs` is a value, while `rhs` is resolved to refer to a property of `lhs`, `lhs::rhs` is a *value-property* reference;
- If `lhs` is a value, while `rhs` is resolved to refer to a function available on `rhs`, `lhs::rhs` is a *value-function* reference.

  Important: callable references to callables which are a member and an extension (that is, an extension to one type declared as a member of a classifier) are forbidden

  Examples:

```
class A {
    val a: Int = 42

    fun a(): String = "TODO()"

    companion object {
        val bProp: Int = 42

        fun bFun(): String = "TODO()"
    }
}

object O {
    val a: Int = 42

    fun a(): String = "TODO()"
}

fun main() {
    // Error: ambiguity between two possible callables
    // val errorAmbiguity = A::a

    // Error: cannot reference companion object implicitly
```

```
        // val errorCompanion = A::bFun

        val aTypePropRef: (A) -> Int = A::a

        val aTypeFunRef: (A) -> String = A::a

        val aValPropRef: () -> Int = A()::a

        val aValFunRef: () -> String = A()::a

        // Error: object type behave as values
        // val oTypePropRef: (O) -> Int = O::a

        // Error: object types behave as values
        // val oTypeFunRef: (O) -> String = O::a

        val oValPropRef: () -> Int = O::a

        val oValFunRef: () -> String = O::a
    }
```

The types of these expressions are implementation-defined, but the following constraints must hold:

- The type of any property reference is a subtype of `kotlin.reflect.KProperty<T>`, where the type parameter `T` is fixed to the type of the property;
- The type of any function reference is a subtype of `kotlin.reflect.KFunction<T>`, where the type parameter `T` is fixed to the return type of the function;
- The type of any callable reference is a subtype of function type which allows the corresponding callable to be accessed/called accordingly.
  - For a type-callable reference `lhs::rhs`, it is a function type `(O, Arg0 ... ArgN) -> R`, where `O` is a receiver type (type of `lhs`), `Arg0, ... , ArgN` are either empty (for a property reference) or the types of function formal parameters (for a function reference), and `R` is the result type of the callable;
  - For a value-callable reference `lhs::rhs`, it is a function type `(Arg0 ... ArgN) -> R`, where `Arg0, ... , ArgN` are either empty (for a property reference) or the types of function formal parameters (for a function reference), and `R` is the result type of the callable. The receiver of such callable reference is bound to `lhs`.

Being of a function type also means callable references are valid callables themselves, with an appropriate operator `invoke` overload, which allows using call syntax to evaluate such callable with the suitable arguments.

> Informally: one may say that any callable reference is essentially the same as a lambda literal with the corresponding number of arguments, delegating to the callable being referenced.

Please note that the above holds for *resolved* callable references, where it is known what entity a particular reference references. In the general case, however, it is unknown as the overload resolution must be performed first. Please refer to the corresponding section for details.

8.20.3 Class literals

A class literal is similar in syntax to a callable reference, with the difference being that it uses the keyword `class`. Similar to callable references, there are two forms of class literals: type and value class literals.

> Note: class literals are one of the few cases where a parameterized type may (and actually **must**) be used without its type parameters.

All class literals `lhs::class` are of type `kotlin.KClass<T>` and produce a platform-defined object associated with type `T`, which, in turn, is either the `lhs` type or the runtime type of the `lhs` value. In both cases, `T` must be a runtime-available non-nullable type. As the runtime type of any expression cannot be known at compile time, the compile-time type of a class literal is `kotlin.KClass<U>` where $T <: U$ and `U` is the compile-time type of `lhs`.

A class literal can be used to access platform-specific capabilities of the runtime type information available on the current platform, either directly or through reflection facilities.

8.20.4 Function calls and property access

Function call expression is an expression used to invoke functions. Property access expression is an expression used to access properties.

There are two kinds of both: with and without explicit receiver (the left-hand side of the `.` operator). For details on how a particular candidate and receiver for a particular call / property access is chosen, please refer to the Overload resolution section.

> Important: in some cases function calls are syntactically indistinguishable from property accesses with `invoke`-convention call suffix.

From this point on in this section we well refer to both as function calls. As described in the function declaration section, function calls receive arguments of several different kinds:

- Explicit receiver argument, used in calls with explicit receivers;
- Normal arguments, provided directly inside the parentheses part of the call;
- Named arguments in the form `identifier = value`, where `identifier` is a parameter name used at declaration-site of the function;

- Variable length arguments, provided the same way as normal arguments;
- A trailing lambda literal argument, specified outside the parentheses (see lambda literal section for details).

In addition to these, a function declaration may specify a number of default parameters, which allow one to omit specifying them at call-site, in which case their default value is used during the evaluation.

The evaluation of a function call begins with the evaluation of its explicit receiver, if it is present. Function arguments are then evaluated **in the order of their appearance in the function call** left-to-right, with no consideration on how the parameters of the function were specified during function declaration. This means that, even if the order at declaration-site was different, arguments at call-site are evaluated in the order they are given. Default arguments not specified in the call are all evaluated **after** all provided arguments, in the order of their appearance in function declaration. Afterwards, the function itself is invoked.

> Note: this means default argument expressions which are used (i.e., for which the call-site does not provide explicit arguments) are reevaluated at every such call-site. Default argument expressions which are not used (i.e., for which the call-site provides explicit arguments) are **not** evaluated at such call-sites.
>
> Examples: we use a notation similar to the control-flow section to illustrate the evaluation order.

```
fun f(x: Int = h(), y: Int = g())
...
f() // $1 = h(); $2 = g(); $result = f($1, $2)
f(m(), n()) // $1 = m(); $2 = n(); $result = f($1, $2)
f(y = n(), x = m()) // $1 = n(); $2 = m(); $result = f($2, $1)
f(y = n()) // $1 = n(); $2 = h(); $result = f($2, $1)

fun f(x: Int = h(), y: () -> Int)
...
f(y = {2}) // $1 = {2}; $2 = h(); $result = f($2, $1)
f { 2 } // $1 = {2}; $2 = h(); $result = f($2, $1)
f(m()) { 2 } // $1 = m(); $2 = {2}; $result = f($1, $2)
```

Operator calls work in a similar way: every operator evaluates in the same order as its expansion does, unless specified otherwise.

> Note: this means that the containment-checking operators are effectively evaluated right-to-left w.r.t. their expansion.

8.20.5 Spread operator expressions

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

Spread operator expression is a special kind of expression which is only applicable in the context of calling a function with variable length parameters. For a spread operator expression `*E` it is required that `E` is of an array type and the expression itself is used as a value argument to a function call. This allows passing an array as a *spread* value argument, providing the elements of an array as the variable length argument of a callable. It is allowed to mix spread arguments with regular arguments, all fitting into the same variable length argument slot, with elements of all spread arguments supplied in sequence.

Example:

```
fun foo(vararg c: String) { ... }
...
val a: String = "a"
val b: Array<String> = arrayOf("b", "c", "d")
val c: String = "e"
val d: Array<String> = arrayOf()
val e: Array<String> = arrayOf("f", "g")
...
foo(a, *b, c, *d, *e)
// is equivalent to
foo("a", "b", "c", "d", "e", "f", "g")
```

Spread operator expressions are not allowed in any other context. See Variable length parameter section for details.

The type of a spread argument must be a subtype of ATS(`kotlin.Array(out` T)) for a variable length parameter of type T.

Example: for parameter `vararg a: Int` the type of a corresponding spread argument must be a subtype of `IntArray`, for parameter `vararg b: T` where `T` is a classifier type the type of a corresponding spread argument must be a subtype of `Array<out T>`.

8.21 Function literals

Kotlin supports using functions as values. This includes, among other things, being able to use named functions (via function references) as parts of expressions. However, sometimes it does not make much sense to provide a separate function declaration, when one would rather define a function in-place. This is implemented using *function literals*.

There are two types of function literals in Kotlin: *lambda literals* and *anonymous function declarations*. Both of these provide a way of defining a function in-place, but have a number of differences which we discuss in their respective sections.

8.21.1 Anonymous function declarations

***anonymousFunction*:**
'fun'
[{*NL*} *type* {*NL*} '.']
{*NL*}
parametersWithOptionalType
[{*NL*} ':' {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

Anonymous function declarations, despite their name, are not declarations per se, but rather expressions which resemble function declarations. They have a syntax very similar to function declarations, with the following key differences:

- Anonymous functions do not have a name;
- Anonymous functions cannot have type parameters;
- Anonymous functions cannot have default parameters;
- Anonymous functions may have variable length parameters, but they are automatically decayed to non-variable length parameters of the corresponding array type via array type specialization;
- Anonymous functions may omit formal parameter types and return type, if they can be inferred from the context.

Anonymous function declaration can declare an anonymous extension function by following the extension function declaration convention.

> Note: as anonymous functions may not have type parameters, you cannot declare an anonymous extension function on a parameterized receiver type.

The type of an anonymous function declaration is the function type constructed similarly to a named function declaration.

8.21.2 Lambda literals

***lambdaLiteral*:**
'{'
{*NL*}
[[*lambdaParameters*] {*NL*} '->' {*NL*}]
statements
{*NL*}
'}'

***lambdaParameters*:**
lambdaParameter {{*NL*} ',' {*NL*} *lambdaParameter*} [{*NL*} ',']

***lambdaParameter*:**
variableDeclaration
| (*multiVariableDeclaration* [{*NL*} `':'` {*NL*} *type*])

Lambda literals are similar to anonymous function declarations in that they define a function with no name. Unlike them, however, lambdas use very different syntax, similar to control structure bodies of other expressions.

Every lambda literal consists of an optional lambda parameter list, specified before the arrow (`->`) operator, and a body, which is everything after the arrow operator.

Lambda body introduces a new statement scope.

Lambda literals have the same restrictions as anonymous function declarations, but additionally cannot have `vararg` parameters.

Lambda literals can introduce destructuring parameters. Lambda parameter of the form `(a, b, ..., n)` (note the parenthesis) declares a destructuring formal parameter, which references the actual argument and its `componentN()` functions as follows (see the operator overloading section for details).

```
val plus: (Pair<Int, Double>) -> String = { (i, d) ->
    "$i + $d = ${i + d}"
}

val plus: (Pair<Int, Double>) -> String = { p ->
    val i = p.component1()
    val d = p.component2()
    "$i + $d = ${i + d}"
}
```

If a lambda expression has no parameter list, it can be defining a function with either zero or one parameter, the exact case dependent on the use context of this lambda. The selection of number of parameters in this case is performed during type inference.

If a lambda expression has no explicit parameter list, but does have one parameter, this parameter can be accessed inside the lambda body using a special property called `it`.

> Note: having no explicit parameter list (no arrow operator) in a lambda is different from having zero parameters (nothing preceding the arrow operator).

Any lambda may define either a normal function or an extension function, the exact case dependent on the use context of the lambda. If a lambda expression defines an extension function, its extension receiver may be accessed using the standard `this` syntax inside the lambda body.

Lambda literals are different from other forms of function declarations in that non-labeled `return` expressions inside lambda body refer to the outer non-lambda

function the expression is used in rather than the lambda expression itself. Such non-labeled returns are only allowed if the lambda and all its parent lambdas (if present) are guaranteed to be inlined, otherwise it is a compile-time error.

If a lambda expression is labeled, it can be returned from using a labeled return expression.

If a **non-labeled** lambda expression is used as a parameter to a function call, the name of the function called may be used as a label.

If a labeled `return` expression is used when there are several matching labels available (e.g., inside several nested function calls with the same name), this is resolved as `return` to the nearest matching label.

Example:

```
// kotlin.run is a standard library inline function
//   receiving a lambda parameter

fun foo() { // (1)
    run b@ { // (2)
        run b@ { // (3)
            return; // returns from (1)
        }
    }
}

fun bar() { // (1)
    run b@ { // (2)
        run b@ { // (3)
            return@b; // returns from (3)
        }
    }
}

fun baz() { // (1)
    run b@ { // (2)
        run c@ { // (3)
            return@b; // returns from (2)
        }
    }
}

fun qux() { // (1)
    run { // (2)
        run { // (3)
            return@run; // returns from (3)
        }
```

```
    }
}

fun quux() { // (1)
    run { // (2)
        run b@ { // (3)
            return@run; // returns from (2)
        }
    }
}

fun quz() { // (1)
    run b@ { // (2)
        run b@ { // (3)
            return@run; // illegal: both run invocations are labeled
        }
    }
}
```

Any properties used inside the lambda body are **captured** by the lambda expression and, depending on whether it is inlined or not, affect how these properties are processed by other mechanisms, e.g. smart casts. See corresponding sections for details.

8.22 Object literals

***objectLiteral*:**
`'object'` [{*NL*} `':'` {*NL*} *delegationSpecifiers* {*NL*}] [{*NL*} *classBody*]

Object literals are used to define anonymous objects in Kotlin. Anonymous objects are similar to regular objects, but they (obviously) have no name and thus can be used only as expressions.

> Note: in object literals, only inner classes are allowed; interfaces, objects or nested classes are forbidden.

Anonymous objects, just like regular object declarations, can have at most one base class and zero or more base interfaces declared in its supertype specifiers.

The main difference between a regular object declaration and an anonymous object is its type. The type of an anonymous object is a special kind of type which is usable (and visible) only in the scope where it is declared. It is similar to a type of a regular object declaration, but, as it cannot be used outside the declaring scope, has some interesting effects.

When a value of an anonymous object type escapes current scope:

- If the type has only one declared supertype, it is implicitly downcasted to this declared supertype;
- If the type has several declared supertypes, there must be an implicit or explicit cast to any suitable type visible outside the scope, otherwise it is a compile-time error.

  Note: an implicit cast may arise, for example, from the results of type inference.

  Note: in this context "escaping current scope" is performed immediately if the corresponding value is declared as a **non-private** global- or classifier-scope property, as those are parts of an externally accessible interface.

  Example:

```
open class Base
interface I

class M {
    fun bar() = object : Base(), I {}
    // Error, as public return type of `bar`
    //   cannot be anonymous

    fun baz(): Base = object : Base(), I {}
    // OK, as an anonymous type is implicitly
    //   cast to Base

    private fun qux() = object : Base(), I {}
    // OK, as an anonymous type does not escape
    //   via private functions

    private fun foo() = object {
        fun bar() { println("foo.bar") }
    }

    fun test1() = foo().bar()

    fun test2() = foo()
    // OK, as an anonymous type is implicitly
    //   cast to Any
}

fun main() {
    M().test1() // OK
    M().test2().bar() // Error: Unresolved reference: bar
}
```

8.22.1 Functional interface lambda literals

If a lambda literal is preceded with a functional interface name, this expression defines an anonymous object, implementing the specified functional interface via the provided lambda literal (which becomes the implementation of its single abstract method).

To be a well-formed functional interface lambda literal, the type of lambda literal must be a subtype of the associated function type of the specified functional interface.

8.23 This-expressions

***thisExpression*:**
`'this'`
| *THIS_AT*

This-expressions are special kind of expressions used to access receivers available in the current scope. The basic form of this expression, denoted by a non-labeled `this` keyword, is used to access the default implicit receiver according to the receiver priority. In order to access other implicit receivers, labeled `this` expressions are used. These may be any of the following:

- `this@type`, where `type` is a name of any classifier currently being declared (that is, this-expression is located in the inner scope of the classifier declaration), refers to the implicit object of the type being declared;
- `this@function`, where `function` is a name of any extension function currently being declared (that is, this-expression is located in the function body), refers to the implicit receiver object of the extension function;
- `this@lambda`, where `lambda` is a label provided for a lambda literal currently being declared (that is, this-expression is located in the lambda expression body), refers to the implicit receiver object of the lambda expression;
- `this@outerFunction`, where `outerFunction` is the name of a function which takes lambda literal currently being declared as an immediate argument (that is, this-expression is located in the lambda expression body), refers to the implicit receiver object of the lambda expression.

  > Note: `this@outerFunction` notation is mutually exclusive with `this@lambda` notation, meaning if a lambda literal is labeled `this@outerFunction` cannot be used.
  >
  > Note: `this@outerFunction` and `this@label` notations can be used only in lambda literals which have an extension function type, i.e., have an implicit receiver.

Important: any other forms of this-expression are illegal and should result in a compile-time error.

In case there are several entities with the same label, labeled `this` refers to the closest label.

Example:

```
interface B
object C

class A/* receiver (1) */ {
    fun B/* receiver (2) */.foo() {
        // `run` is a standard library function
        //   with an extension lambda parameter
        C/* receiver (3) */.run {
            this // refers to receiver (3) of type C
            this@A // refers to receiver (1) of type A
            // this@B // illegal: B is not being declared
            this@foo // refers to receiver (2) of type B
            this@run // refers to receiver (3) of type C
        }
        C/* receiver (4) */.run label@{
            this // refers to receiver (4) of type C
            this@A // refers to receiver (1) of type A
            // this@B // illegal: B is not being declared
            this@foo // refers to receiver (2) of type B
            this@label // refers to receiver (4) of type C
            // this@run // illegal: lambda literal is labeled
        }
    }
}
```

8.24 Super-forms

***superExpression*:**
('super' ['<' {*NL*} *type* {*NL*} '>'] [*AT_NO_WS simpleIdentifier*])
| *SUPER_AT*

Super-forms are special kind of expression which can only be used as receivers in a call or property access expression. Any use of super-form expression in any other context is a compile-time error.

Super-forms are used in classifier declarations to access implementations from the immediate supertypes without invoking overriding behaviour.

If an implementation is not available (e.g., one attempts to access an abstract method of a supertype in this fashion), this is a compile-time error.

The basic form of this expression, denoted by `super` keyword, is used to access the immediate supertype of the currently declared classifier selected as a part of overload resolution. In order to access a specific supertype implementations, extended `super` expressions are used. These may be any of the following:

- `super<Klazz>`, where `Klazz` is a name of one of the immediate supertypes of the currently declared classifier, refers to that supertype and its implementations;
- `super<Klazz>@type`, where `type` is a name of any currently declared classifier and `Klazz` is a name of one of the immediate supertypes of the `type` classifier, refers to that supertype and its implementations.

  > Note: `super<Klazz>@type` notation can be used only in inner classes, as only inner classes can have access to supertypes of other classes, i.e., supertypes of their parent class.

  Example:

```
interface A {
    fun foo() { println("A") }
}
interface B {
    fun foo() { println("B") }
}

open class C : A {
    override fun foo() { println("C") }
}

class E : C() {
    init {
        super.foo() // "C"
        super<C>.foo() // "C"
    }
}

class D : C(), A, B {
    init {
        // Error: ambiguity as several immediate supertypes
        //   with callable `foo` are available here
        // super.foo()
        super<C>.foo() // "C"
        super<B>.foo() // "B"
        // Error: A is *not* an immediate supertype,
        //   as C inherits from A and is considered
```

```
            //   to be "more immediate"
            // super<A>.foo()
        }

        inner class Inner {
            init {
                // Error: C is not available
                // super<C>.foo()
                super<C>@D.foo() // "C"
                super<B>@D.foo() // "B"
            }
        }

        override fun foo() { println("D") }
    }
```

8.25 Jump expressions

***jumpExpression*:**
('throw' {*NL*} *expression*)
| (('return' | *RETURN_AT*) [*expression*])
| 'continue'
| *CONTINUE_AT*
| 'break'
| *BREAK_AT*

Jump expressions are expressions which redirect the evaluation of the program to a different program point. All these expressions have several things in common:

- They all have type `kotlin.Nothing`, meaning that they never produce any runtime value;
- Any code which follows such expressions is never evaluated.

8.25.1 Throw expressions

Throw expression `throw e` allows throwing exception objects. A valid throw expression `throw e` requires that:

- `e` is a value of a runtime-available type;
- `e` is a value of an exception type.

Throwing an exception results in checking active `try`-blocks. See the Exceptions section for details.

8.25.2 Return expressions

A *return expression*, when used inside a function body, immediately stops evaluating the current function and returns to its caller, effectively making the function call expression evaluate to the value specified in this return expression (if any). A return expression with no value implicitly returns the `kotlin.Unit` object.

There are two forms of return expression: a simple return expression, specified using the non-labeled `return` keyword, which returns from the innermost function declaration (or anonymous function declaration), and a labeled return expression of the form `return@Context` which works as follows.

- If `return@Context` is used inside a named function declaration, the name of the declared function may be used as `Context` to refer to that function. If several declarations match the same name, the `return@Context` is considered to be from the nearest matching function;
- If `return@Context` is used inside a non-labeled lambda literal, the name of the function **using** this lambda expression as its argument may be used as `Context` to refer to the lambda literal;
- If `return@Context` is used inside a labeled lambda literal, the label may be used as `Context` to refer to the lambda literal.

If a return expression is used in the context of a lambda literal which is *not inlined* in the current context and refers to any function scope declared outside this lambda literal, it is disallowed and should result in a compile-time error.

> Note: these rules mean a simple return expression inside a lambda expression returns **from the innermost function** in which this lambda expression is defined. They also mean such return expression is allowed only inside **inlined** lambda expressions.

8.25.3 Continue expressions

A *continue expression* is a jump expression allowed only within loop bodies. When evaluated, this expression passes the control to the start of the next loop iteration (aka “continue-jumps”).

There are two forms of continue expressions:

- A simple continue expression, specified using the `continue` keyword, which continue-jumps to the innermost loop statement in the current scope;
- A labeled continue expression, denoted `continue@Loop`, where `Loop` is a label of a labeled loop statement `L`, which continue-jumps to the loop `L`.

If a continue expression is used in the context of a lambda literal which refers to any loop scope outside this lambda literal, it is disallowed and should result in a compile-time error.

8.25.4 Break expressions

A *break expression* is a jump expression allowed only within loop bodies. When evaluated, this expression passes the control to the next program point immediately after the loop (aka "break-jumps").

There are two forms of break expressions:

- A simple break expression, specified using the `break` keyword, which break-jumps to the innermost loop statement in the current scope;
- A labeled break expression, denoted `break@Loop`, where `Loop` is a label of a labeled loop statement `L`, which break-jumps to the loop `L`.

If a break expression is used in the context of a lambda literal which refers to any loop scope outside this lambda literal, it is disallowed and should result in a compile-time error.

Chapter 9

Operator overloading

Some syntax forms in Kotlin are *defined by convention*, meaning that their semantics are defined through syntactic expansion of one syntax form into another syntax form.

Particular cases of definition by convention include:

- Arithmetic and comparison operators;
- `invoke` convention;
- Operator-form assignments;
- For-loop statements;
- Delegated properties;
- Destructuring declarations.

  Important: another case of definition by convention is safe navigation, which is covered in more detail in its respective section.

There are several points shared among all the syntax forms defined using definition by convention:

- The expansions are hygienic: if they introduce new identifiers that were not present in original syntax, all such identifiers are not accessible outside the expansion and cannot clash with any other declarations in the program;
- The expressions captured by an expansion are using *call-by-need* evaluation strategy, meaning that they are evaluated only once during first usage specified in the expansion even if the expansion itself has more than one usage of such an expression;
- An expansion may lead to another expansion, following the same rules;
- All call expressions that are produced by expansion are only allowed to use operator functions.

This expansion of a particular syntax form to a different piece of code is usually defined in the terms of *operator* functions.

Operator functions are function which are declared with a special keyword `operator` and are not different from regular functions when called via function calls. However, operator functions can also be used in definition by convention.

> Note: it is not important whether an operator function is a member or an extension, nor whether it is suspending or not. The only requirements are the ones listed in the respected sections.
>
> For example, for an operator form `a + b` where `a` is of type `A` and `b` is of type `B` any of the following function definitions are applicable:

```
class A {
    // member function
    operator fun plus(b: B) = ...
    // suspending member function
    suspend operator fun plus(b: B) = ...
}

// extension function
operator fun A.plus(b: B) = ...
// suspending extension function
suspend operator fun A.plus(b: B) = ...
```

> Assuming additional implicit receiver of this type is available, it may also be an extension defined in another type:

```
object Ctx {
    // extension that is a member of some context type
    operator fun A.plus(b: B) = ...

    fun add(a: A, b: B) = a + b
}
```

> Note: different platforms may add additional criteria on whether a function may be considered a suitable candidate for operator convention.

The details of individual expansions are available in the sections of their respective operators, here we would like to describe how they *interoperate*.

For example, take the following declarations:

```
class A {
    operator fun inc(): A { ... }
}

object B {
    operator fun get(i: Int): A { ... }
    operator fun set(i: Int, value: A) { ... }
}
```

```
object C {
    operator fun get(i: Int): B { ... }
}
```

The expression `C[0][0]++` is expanded (see the Expressions section for details) using the following rules:

- The operations are expanded in order of their priority.
- First, the increment operator is expanded, resulting in:

  ```
  C[0][0] = C[0][0].inc()
  ```

- Second, the assignment to an indexing expression (produced by the previous expansion) is expanded, resulting in:

  ```
  C[0].set(C[0][0].inc())
  ```

- Third, the indexing expressions are expanded, resulting in:

  ```
  C.get(0).set(C.get(0).get(0).inc())
  ```

  Important: although the resulting expression contains several instances of the subexpression `C.get(0)`, as all these instances were created from the same original syntax form, the subexpression is evaluated only once, making this code roughly equivalent to:

  ```
  val $tmp = C.get(0)
  $tmp.set($tmp.get(0).inc())
  ```

9.1 Destructuring declarations

A special case of definition by convention is the destructuring declaration of properties, which is available for local properties, parameters of lambda literals and the iteration variable of for-loops. See the corresponding sections for particular syntax.

This convention allows to introduce a number (one or more) of properties in the place of one by immediately *destructuring* the property during construction. The immediate value (that is, the initializing expression of the local property, the value acquired from the operator convention of a for-loop statement, or an argument passed into a lambda body) is assigned to a number of placeholders $p_0, \ldots, p_N$ where each placeholder is either an identifier or a special ignoring placeholder `_` (note that `_` is not a valid identifier in Kotlin). For each identifier the corresponding operator function `componentK` with K being equal to the position of the placeholder in the declaration (**starting from 1**) is called without arguments and the result is assigned to a fresh value referred to as the identifier used. For each ignoring placeholder, no calls are performed and nothing

is assigned. Each placeholder may be provided with an optional type signature T_M which is used in type inference as any property type would. Note that an ignoring placeholder may also be provided with a type signature, in which case although the call to corresponding `componentM` function is not performed, it still must be checked for function applicability during type inference.

Examples:

```
val (x: A, _, z) = f()
```

is expanded to

```
val $tmp = f()
val x: A = $tmp.component1()
val z = $tmp.component3()
```

where `component1` and `component3` are suitable operator functions available on the value returned by `f()`

```
for((x: A, _, z) in f()) { ... }
```

is expanded to (as per for-loop expansion)

```
when(val $iterator = f().iterator()) {
    else -> while ($iterator.hasNext()) {
                val $tmp = $iterator.next()
                val x: A = $tmp.component1()
                val z = $tmp.component3()
                ...
            }
}
```

where `iterator()`, `next()`, `hasNext()`, `component1()` and `component3` are all suitable operator functions available on their respective receivers.

```
foo { (x: A, _, z) -> ... }
```

is expanded to

```
foo { $tmp ->
    val x: A = $tmp.component1()
    val z = $tmp.component3()
    ...
}
```

where `component1()` and `component3` are all suitable operator functions available on the value of lambda argument.

Chapter 10

Packages and imports

A Kotlin project is structured into **packages**. A package contains one or more Kotlin files, with files linked to a package using a *package header*. A file may contain exactly one or zero package headers, meaning each file belongs to exactly one package.

> Note: an absence of a package header in a file means it belongs to the special *root package*.

***packageHeader*:**
['`package`' *identifier* [*semi*]]

> Note: Packages are orthogonal from modules. A module may contain many packages, and a single package can be spread across several modules.

The name of a package is a simple or a qualified path, which creates a package hierarchy.

> Note: unlike many other languages, Kotlin packages *do not* require files to have any specific locations w.r.t. itself; the connection between a file and its package is established only via a package header. It is strongly recommended, however, that the folder structure of a project does correspond to the package hierarchy.

10.1 Importing

Program entities declared in one package may be freely used in any file in the same package with the only two restrictions being module boundaries and visibility constraints. In order to use an entity from a file belonging to a different package, the programmer must use *import directives*.

importList:
{*importHeader*}

importHeader:
`'import'` *identifier* [(`'.'` `'*'`) | *importAlias*] [*semi*]

importAlias:
`'as'` *simpleIdentifier*

An import directive contains a simple or a qualified path, with the name of an imported entity as its last component. A path may include not only a package, but also an object or a type, in which case it refers to the companion object of that type. The last component may reference any named declaration within that scope (that is, top-level scope of all files in the package or an object declaration scope) may be imported using their names.

There are two special kinds of imports: star-imports ending in an asterisk (`*`) and renaming imports employing the `as` operator. Star-imports import all named entities inside the corresponding scope, but have weaker priority during overload resolution of functions and properties. Renaming imports work just like regular imports, but introduce the entity into the current file with the specified name.

Imports from objects have certain limitations: only object members may be imported and star-imports are not allowed.

Imports are local to their files, meaning if an entity is introduced into file A.kt from package `foo.bar`, it does not introduce that entity to any other file from package `foo.bar`.

There are some packages which have all their entities *implicitly imported* into any Kotlin file, meaning one can access such entity without explicitly using import directives.

> Note: one may, however, import these entities explicitly if they choose to do so.

The following packages of the standard library are implicitly imported:

- `kotlin`
- `kotlin.annotation`
- `kotlin.collections`
- `kotlin.comparisons`
- `kotlin.io`
- `kotlin.ranges`
- `kotlin.sequences`
- `kotlin.text`
- `kotlin.math`

> Note: platform implementations may introduce additional implicitly imported packages, for example, to extend Kotlin code with

the platform-specific functionality. An example of this would be `java.lang` package implicitly imported on the JVM platform.

Importing certain entities may be disallowed by their visibility modifiers.

- `public` entities can be imported anywhere
- `internal` entities can be imported only within the same module
- `protected` entities cannot be imported
- top-level `private` entities can be imported within their declaring file
- other `private` entities cannot be imported

10.2 Modules

A module is a concept on the boundary between the code itself and the resulting application, thus it depends on and influences both of them. A Kotlin module is a set of Kotlin files which are considered to be interdependent and must be handled together during compilation.

In a simple case, a module is a set of files compiled at the same time in a given project.

- A set of files being compiled with a single Kotlin compiler invocation
- A Maven module
- A Gradle project

In a more complicated case involving multi-platform projects, a module may be distributed across several compilations, projects and/or platforms.

For the purposes of Kotlin/Core, modules are important for `internal` visibility. How modules influence particular platforms is described in their respective sections of this specification.

Chapter 11

Overload resolution

Glossary

`type(e)`
: Type of expression `e`

Introduction

Kotlin supports *callable overloading*, that is, the ability for several callables (functions or function-like properties) with the same name to coexist in the same scope, with the compiler picking the most suitable one when such a callable is called. This section describes *overload resolution process* in detail.

Unlike other object-oriented languages, Kotlin does not have only regular class methods, but also top-level functions, local functions, extension functions and function-like values, which complicate the overload resolution process quite a bit. Additionally, Kotlin has infix functions, operator and property overloading, which add their own specifics to this process.

11.1 Receivers

Every function or property that is defined as a method or an extension has one or more special parameters called *receiver* parameters. When calling such a callable using navigation operators (`.` or `?.`) the left hand side value is called an *explicit receiver* of this particular call. In addition to the explicit receiver, each call may indirectly access zero or more *implicit receivers*.

Implicit receivers are available in a syntactic scope according to the following rules:

- Any receiver available in a scope is available in its downwards-linked scopes;
- In a classifier declaration scope (including object and companion object declarations), the declared object is available as implicit `this`;
- In a classifier declaration scope (including object and companion object declarations), the static callables of the declared object are available on a phantom static implicit `this`;
- If a function or a property is an extension, `this` parameter of the extension is also available inside the extension declaration;
- If a lambda expression has an extension function type, `this` argument of the lambda expression is also available inside the lambda expression declaration.

  Important: a phantom static implicit `this` is a special receiver, which is included in the receiver chain for the purposes of handling static functions from enum classes. It may also be used on platforms to handle their static-like entities, e.g., static methods on JVM platform.

The available receivers are prioritized in the following way:

- Receivers provided in the most inner scope have higher priority as ordered w.r.t. link relation;
- The implicit `this` receiver has higher priority than phantom static implicit `this`;
- The phantom static implicit `this` receiver has higher priority than the current class companion object receiver;
- Current class companion object receiver has higher priority than any of the superclass companion objects;
- Superclass companion object receivers are prioritized according to the inheritance order.

  Important: these rules mean implicit receivers are always totally ordered w.r.t. their priority, as no two implicit receivers can have the same priority.

  Important: DSL-specific annotations (marked with `kotlin.DslMarker` annotation) change the availability of implicit receivers in the following way: for all types marked with a particular DSL-specific annotation, only the highest priority implicit receiver is available in a given scope.

The implicit receiver having the highest priority is also called the *default implicit receiver*. The default implicit receiver is available in a scope as `this`. Other available receivers may be accessed using labeled this-expressions.

If an implicit receiver is available in a given scope, it may be used to call callables implicitly in that scope without using the navigation operator.

For extension callables, the receiver used as the extension receiver parameter is called *extension receiver*, while the implicit receiver associated with the declaration scope the extension is declared in is called *dispatch receiver*. For a particular callable invocation, any or both receivers may be involved, but, if an extension receiver is involved, the dispatch receiver must be implicit.

> Note: by definition, local extension callables do not have a dispatch receiver, as they are declared in a statement scope.
>
> Note: there may be situations in which *the same implicit receiver* is used as both the dispatch receiver and the extension receiver for a particular callable invocation, for example:

```
interface Y

class X : Y {
    fun Y.foo() {} // `foo` is an extension for Y,
                   //   needs extension receiver to be called

    fun bar() {
        foo() // `this` reference is both
              //   the extension and the dispatch receiver
    }
}

fun <T> mk(): T = TODO()

fun main() {
    val x: X = mk()
    val y: Y = mk()

    // y.foo()
    // Error, as there is no implicit receiver
    //   of type X available

    with (x) {
        y.foo() // OK!
    }
}
```

11.2 The forms of call-expression

Any function in Kotlin may be called in several different ways:

- A fully-qualified call without receiver: `package.foo()`;
- A call with an explicit receiver: `a.foo()`;

- An infix function call: `a foo b`;
- An overloaded operator call: `a + b`;
- A call without an explicit receiver: `foo()`.

Although syntactically similar, there is a difference between the first two kinds of calls: in the first case, `package` is a name of a Kotlin package, while in the second case `a` is a value or a type.

For each of these cases, a compiler should first pick a number of *overload candidates*, which form a set of possibly intended callables (*overload candidate set*, OCS), and then *choose the most specific function* to call based on the types of the function and the call arguments.

> Important: the overload candidates are picked **before** the most specific function is chosen.

11.3 Callables and `invoke` convention

A *callable* X for the purpose of this section is one of the following:

- Function-like callables:
  - A function named X at its declaration site;
  - A constructor of a type named X at its declaration site;
  - Any of the above named Y at its declaration site, but imported into the current scope using a renaming import as X.
- Property-like callables with an operator function `invoke` available as a member or an extension in the current scope:
  - A property named X at its declaration site;
  - An object or a companion object named X at its declaration site;
  - A companion object of a classifier type named X at its declaration site;
  - An enum entry named X at its declaration site;
  - Any of the above named Y at its declaration site, but imported into the current scope using a renaming import as X.

For property-like callables, a call $X(Y_0, \ldots, Y_N)$ is an overloadable operator which is expanded to X.invoke$(Y_0, \ldots, Y_N)$. The call may contain type parameters, named parameters, variable argument parameter expansion and trailing lambda parameters, all of which are forwarded as-is to the corresponding `invoke` function.

The set of implicit receivers itself (denoted by `this` expression) may also be used as a property-like callable using `this` as the left-hand side of the call expression. As with normal property-like callables, `this@A`$(Y_0, \ldots, Y_N)$ is an overloadable operator which is expanded to `this@A.invoke`$(Y_0, \ldots, Y_N)$.

A *member callable* is one of the following:

- a member function-like callable (including constructors);

- a member property-like callable with a member operator `invoke`.

An *extension callable* is one of the following:

- an extension function-like callable;
- a member property-like callable with an extension operator `invoke`;
- an extension property-like callable with a member operator `invoke`;
- an extension property-like callable with an extension operator `invoke`.

  Informally: the mnemonic rule to remember this order is "functions before properties, members before extensions".

A *local callable* is any callable which is declared in a statement scope.

11.4 c-level partition

When calculating overload candidate sets, member callables produce the following sets, considered separately, ordered by higher priority first:

- Member function-like callables;
- Member property-like callables.

Extension callables produce the following sets, considered separately, ordered by higher priority first:

- Extension function-like callables;
- Member property-like callables with extension `invoke`;
- Extension property-like callables with member `invoke`;
- Extension property-like callables with extension `invoke`.

Let us define this partition of callables to overload candidate sets as *c-level partition* (callable-level partition). As this partition is the most fine-grained of all other steps of partitioning resolution candidates into sets, it is always performed **last**, after all other applicable steps.

11.5 Building the overload candidate set (OCS)

11.5.1 Fully-qualified call

If a call is fully-qualified (that is, it contains a complete package path), then the overload candidate set S simply contains all the top-level callables with the specified name in the specified package. As a package name can never clash with any other declared entity, after performing c-level partition on S, the resulting sets are the only ones available for further processing.

Example:

```
package a.b.c

fun foo(a: Int) {}
fun foo(a: Double) {}
fun foo(a: List<Char>) {}
val foo = {}
. . .
a.b.c.foo()
```

Here the resulting overload candidate set contains all the callables named `foo` from the package `a.b.c`.

Important: a fully-qualified callable name has the form `P.n()`, where `n` is a simple callable name and `P` is a complete package path referencing an existing package.

11.5.2 Call with an explicit receiver

If a call is done via a navigation operator (`.` or `?.`), but is not a fully-qualified call, then the left hand side value of the call is the explicit receiver of this call.

A call of callable `f` with an explicit receiver `e` is correct if at least one of the following holds:

1. `f` is an accessible member callable of the classifier type `type(e)` or any of its supertypes;
2. `f` is an accessible extension callable of the classifier type `type(e)` or any of its supertypes, including top-level, local and imported extensions.
3. `f` is an accessible static member callable of the classifier type `e`.

   Important: callables for case 2 include not only regular extension callables, but also extension callables from any of the available implicit receivers. For example, if class `P` contains a member extension function `f` for another class `T` and an object of class `P` is available as an implicit receiver, extension function `f` may be used for such call if `T` conforms to the type `type(e)`.

If a call is correct, for a callable `f` with an explicit receiver `e` of type `T` the following sets are analyzed (**in the given order**):

1. Non-extension member callables named `f` of type `T`;
2. Local extension callables named `f`, whose receiver type conforms to type `T`, in the current scope and its upwards-linked scopes, ordered by the size of the scope (smallest first), excluding the package scope;
3. Explicitly imported extension callables named `f`, whose receiver type conforms to type `T`;
4. Extension callables named `f`, whose receiver type conforms to type `T`, declared in the package scope;

5. Star-imported extension callables named `f`, whose receiver type conforms to type `T`;
6. Implicitly imported extension callables named `f` (either from the Kotlin standard library or platform-specific ones), whose receiver type conforms to type `T`.

> Note: here type `U` conforms to type `T`, if $T <: U$.

> Note: a call to an extension callable with an explicit extension receiver, as noted above, may involve an implicit dispatch receiver. In this case, the case with **no implicit receiver** is considered first; then, for each implicit receiver available, a separate number of sets is constructed according to the rules for implicit receivers. These sets are considered in the order of the implicit receiver priority.

There is a important special case here, however, as a callable may be a property-like callable with an operator function `invoke`, and these may belong to different sets (e.g., the property itself may be star-imported, while the `invoke` operator on it is a local extension). In this situation, such callable belongs to the **lowest priority** set of its parts (e.g., for the above case, priority 5 set).

> Example: when trying to resolve between an explicitly imported extension property (priority 3) with a member `invoke` (priority 1) and a local property (priority 2) with a star-imported extension `invoke` (priority 5), the first one wins (`max(3, 1) < max(2, 5)`).

When analyzing these sets, the **first** set which contains **any** applicable callable is picked for c-level partition, which gives us the resulting overload candidate set.

> Important: this means, among other things, that if the set constructed on step Y contains the overall most suitable candidate function, but the set constructed on step $X < Y$ is not empty, the callables from set X will be picked despite them being less suitable overload candidates.

After we have fixed the overload candidate set, we search this set for the most specific callable.

Call with an explicit type receiver

A call with an explicit receiver may be performed not only on a value receiver, but also on a *type* receiver.

> Note: type receivers can appear when working with enum classes or interoperating with platform-dependent code.

They mostly follow the same rules as calls with an explicit value receiver. However, for a callable `f` with an explicit type receiver `T` the following sets are analyzed (**in the given order**):

1. Static member callables named `f` of type `T`;

2. The overload candidate sets for call `T.f()`, where `T` is a companion object of type `T`.

Call with an explicit super-form receiver

A call with an explicit receiver may be performed not only on a value receiver, but also on a super-form receiver.

They mostly follow the same rules as calls with an explicit value receiver. However, there are some differences which we outline below.

For a callable `f` with an explicit basic super-form receiver `super` in a classifier declaration with supertypes `A1`, `A2`, ... , `AN` the following sets are considered for **non-emptiness**:

1. Non-extension member callables named `f` of type `A1`;
2. Non-extension member callables named `f` of type `A2`;
3. ...;

n. Non-extension member callables named `f` of type `AN`.

If at least two of these sets are non-empty, this is a compile-time error. Otherwise, the non-empty set (if any) is analyzed as usual.

For a callable `f` with an explicit extended super-form receiver `super<A>` the following sets are analyzed (**in the given order**):

1. Non-extension member callables named `f` of type `A`.

Additionally, in either case, abstract callables are not considered valid candidates for the overload resolution process.

11.5.3 Infix function call

Infix function calls are a special case of function calls with explicit receiver in the left hand side position, i.e., `a foo b` may be an infix form of `a.foo(b)`.

However, there is an important difference: during the overload candidate set construction the only callables considered for inclusion are the ones with the `infix` modifier. This means we consider only function-like callables with `infix` modifier and property-like callables with an `infix` operator function `invoke`. All other callables are not considered for inclusion. Aside from this difference, candidates are selected using the same rules as for normal calls with explicit receiver.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for calls with explicit receiver.

Different platform implementations may extend the set of functions considered as infix functions for the overload candidate set.

11.5.4 Operator call

According to the operator overloading section, some operator expressions in Kotlin can be overloaded using definition-by-convention via specifically-named functions. This makes operator expressions semantically equivalent to function calls with explicit receiver, where the receiver expression is selected based on the operator used.

However, there is an important difference: during the overload candidate set construction the only functions considered for inclusion are the ones with the `operator` modifier. All other functions (and any properties) are not considered for inclusion. Aside from this difference, candidates are selected using the same rules as for normal calls with explicit receiver.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for calls with explicit receiver.
>
> Note: this also means that all the properties available through the `invoke` convention are non-eligible for operator calls, as there is no way of specifying the `operator` modifier for them; even though the `invoke` callable is required to always have such modifier. As `invoke` convention itself is an operator call, it is impossible to use more than one `invoke` convention in a single call.

Different platform implementations may extend the set of functions considered as operator functions for the overload candidate set.

> Note: these rules are valid not only for dedicated operator expressions, but also for other operator-based defined-by-convention calls, e.g., `for`-loop iteration conventions, operator-form assignments or property delegation.

11.5.5 Call without an explicit receiver

A call which is performed with a simple path is a call **without** an explicit receiver. As such, it may either have one or more implicit receivers or reference a top-level function.

> Note: this case does not include calls using the `invoke` operator function where the left-land side of the call is not an identifier, but some other kind of expression (as this is not a simple path). These cases are handled the same way as operator calls and need no further special treatment.
>
> Example:

```
fun foo(a: Foo, b: Bar) {
    (a + b)(42)
    // Such a call is handled as if it is
```

```
        //   (a + b).invoke(42)
    }
```

As with calls with explicit receiver, we first pick an overload candidate set and then search this set for the most specific function to match the call.

For an identifier named `f` the following sets are analyzed (**in the given order**):

1. Local non-extension callables named `f` in the current scope and its upwards-linked scopes, ordered by the size of the scope (smallest first), excluding the package scope;
2. The overload candidate sets for each pair of implicit receivers `e` and `d` available in the current scope, calculated as if `e` is the explicit receiver, in order of the receiver priority;
3. Top-level non-extension functions named `f`, in the order of:
   a. Callables explicitly imported into the current file;
   b. Callables declared in the same package;
   c. Callables star-imported into the current file;
   d. Implicitly imported callables (either from the Kotlin standard library or platform-specific ones).

Similarly to how it works for calls with explicit receiver, a property-like callable with an `invoke function` belongs to the **lowest priority** set of its parts.

When analyzing these sets, the **first** set which contains **any** callable with the corresponding name and conforming types is picked for c-level partition, which gives us the resulting overload candidate set.

After we have fixed the overload candidate set, we search this set for the most specific callable.

11.5.6 Call with named parameters

Calls in Kotlin may use named parameters in call expressions, e.g., `f(a = 2)`, where `a` is a parameter specified in the declaration of `f`. Such calls are treated the same way as normal calls, but the overload resolution sets are filtered to only contain callables which have matching formal parameter names for all named parameters from the call.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for the respective type of call.
>
> Note: for properties called via `invoke` convention, the named parameters must be present in the declaration of the `invoke` operator function.

Unlike positional arguments, named arguments are matched by name directly to their respective formal parameters; this matching is performed separately for each function candidate.

While the *number of defaults* does affect resolution process, the fact that some argument was or was not mapped as a named argument does not affect this process in any way.

11.5.7 Call with trailing lambda expressions

A call expression may have a single lambda expression placed outside of the argument list or even completely replacing it (see this section for further details). This has no effect on the overload resolution process, aside from the argument reordering which may happen because of variable length parameters or parameters with defaults.

> Example: this means that calls `f(1, 2) { g() }` and `f(1, 2, body = { g() })` are completely equivalent w.r.t. the overload resolution, assuming `body` is the name of the last formal parameter of `f`.

11.5.8 Call with specified type parameters

A call expression may have a type argument list explicitly specified before the argument list (see this section for further details). Such calls are treated the same way as normal calls, but the overload resolution sets are filtered to only contain callables which contain exactly the same number of formal type parameters at declaration site. In case of a property-like callable with `invoke`, type parameters must be present at the `invoke` operator function declaration instead.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for the respective type of call.

11.6 Determining function applicability for a specific call

11.6.1 Rationale

A function is *applicable* for a specific call if and only if the function parameters may be assigned the arguments values specified at the call site and all type constraints of the function type parameters hold w.r.t. supplied or inferred type arguments.

11.6.2 Description

Determining function applicability for a specific call is a type constraint problem.

First, for every non-lambda argument of the function called, type inference is performed. Lambda arguments are excluded, as their type inference needs the results of overload resolution to finish.

Second, the following constraint system is built:

- For every non-lambda argument inferred to have type T_i, corresponding to the function parameter of type U_j, a constraint $T_i <: U_j$ is constructed;
- All declaration-site type constraints for the function are also added to the constraint system;
- For every lambda argument with the number of lambda arguments known to be K, corresponding to the function parameter of type U_m, a special constraint of the form $(\mathrm{FT}(L_1, \ldots, L_K) \to R \,\&\, \mathrm{FTR}(\mathrm{RT}, L_1, \ldots, L_n) \to R) <: U_m$ is added to the constraint system, where $R, \mathrm{RT}, L_1, \ldots, L_K$ are fresh type variables;
- For each lambda argument with an unknown number of lambda arguments (that is, being equal to 0 or 1), corresponding to the function parameter of type U_n, a special constraint of the form $(\mathrm{FT}() \to R \,\&\, \mathrm{FT}(L) \to R \,\&\, \mathrm{FTR}(\mathrm{RT}) \to R \,\&\, \mathrm{FTR}(\mathrm{RT}, L) \to R) <: U_m$ is added to the constraint system, where R, RT, L are fresh type variables;

If this constraint system is sound, the function is applicable for the call. Only applicable functions are considered for the next step: choosing the most specific candidate from the overload candidate set.

Receiver parameters are handled in the same way as other parameters in this mechanism, with one important exception: any receiver of type `kotlin.Nothing` is deemed not applicable for any member callables, regardless of other parameters. This is due to the fact that, as `kotlin.Nothing` is the subtype of any other type in Kotlin type system, it would have allowed **all** member callables of **all** available types to participate in the overload resolution, which is theoretically possible, but very resource-consuming and does not make much sense from the practical point of view. Extension callables are still available, because they are limited to the declarations available or imported in the current scope.

> Note: although it is impossible to create a value of type `kotlin.Nothing` directly, there may be situations where performing overload resolution on such value is necessary; for example, it may occur when doing safe navigation on values of type `kotlin.Nothing?`.

11.7 Choosing the most specific candidate from the overload candidate set

11.7.1 Rationale

The main rationale for choosing the most specific candidate from the overload candidate set is the following:

> The most specific callable can forward itself to any other callable from the overload candidate set, while the opposite is not true.

If there are several functions with this property, none of them are the most specific and an overload resolution ambiguity error should be reported by the compiler.

Consider the following example.

```
fun f(arg: Int, arg2: String) {}        // (1)
fun f(arg: Any?, arg2: CharSequence) {} // (2)
...
f(2, "Hello")
```

Both functions (1) and (2) are applicable for the call, but function (1) could easily call function (2) by forwarding both arguments into it, and the reverse is impossible. As a result, function (1) is more specific of the two.

```
fun f1(arg: Int, arg2: String) {
    f2(arg, arg2) // VALID: can forward both arguments
}
fun f2(arg: Any?, arg2: CharSequence) {
    f1(arg, arg2) // INVALID: function f1 is not applicable
}
```

The rest of this section will describe how the Kotlin compiler checks for this property in more detail.

11.7.2 Algorithm of MSC selection

When an overload resolution set S is selected and it contains more than one callable, we need to choose the most specific candidate from these callables. The selection process uses the type constraint facilities of Kotlin, in a way similar to the process of determining function applicability.

For every two distinct members of the candidate set F_1 and F_2, the following constraint system is constructed and solved:

- For every non-default argument of the call and their corresponding declaration-site parameter types $X_1, \ldots, X_N$ of F_1 and $Y_1, \ldots, Y_N$ of

F_2, a type constraint $X_K <: Y_K$ is built **unless both X_K and Y_K are built-in integer types.** If both X_K and Y_K are built-in integer types, a type constraint $\text{Widen}(X_K) <: \text{Widen}(Y_K)$ is built instead, where Widen is the integer type widening operator. During construction of these constraints, all declaration-site type parameters $T_1, \ldots, T_M$ of F_1 are considered bound to fresh type variables $T_1^\sim, \ldots, T_M^\sim$, and all type parameters of F_2 are considered free;

- If F_1 and F_2 are extension callables, their extension receivers are also considered non-default arguments of the call, even if implicit, and the corresponding constraints are added to the constraint system as stated above. For non-extension callables, only declaration-site parameters are considered;
- All declaration-site type constraints of $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ are also added to the constraint system.

  Note: this constraint system checks whether F_1 can forward itself to F_2.

If the resulting constraint system is sound, it means that F_1 is equally or more applicable than F_2 as an overload candidate (aka applicability criteria). The check is then repeated with F_1 and F_2 swapped.

This check may result in one of the following outcomes:

1. Only one of the two candidates is more applicable than the other;
2. Neither of the two candidates is more applicable than the other;
3. Both F_1 and F_2 are more applicable than the other.

In case 1, the more applicable candidate of the two is found and no additional steps are needed.

In case 2, an additional step is taken: a non-parameterized callable is a more specific candidate than any parameterized callable. If this step does not allow for deciding the more specific candidate, this is an **overload ambiguity** which must be reported as a compile-time error.

In case 3, several additional steps are performed in order.

- Any non-parameterized callable is a more specific candidate than any parameterized callable (same as case 2). If there are several non-parameterized candidates, further steps are limited to those candidates;
- For each candidate we count the number of default parameters *not* specified in the call (i.e., the number of parameters for which we use the default value). The candidate with the least number of non-specified default parameters is a more specific candidate;
- For all candidates, the candidate having any variable-argument parameters is less specific than any candidate without them.

  Note: it may seem strange to process built-in integer types in a way different from other types, but it is needed for cases when the

call argument is an integer literal with an integer literal type. In this particular case, several functions with different built-in integer types for the corresponding parameter may be applicable, and the `kotlin.Int` overload is selected to be the most specific.

Important: compiler implementations may extend these steps with additional checks, if they deem necessary to do so.

If after these additional steps there are still several candidates which are equally applicable for the call, this is an **overload ambiguity** which must be reported as a compile-time error.

Note: unlike the applicability test, the candidate comparison constraint system is **not** based on the actual call, meaning that, when comparing two candidates, only constraints visible at *declaration site* apply.

If the callables in check are properties with available `invoke`, the same process is applied in two steps:

- First, the properties are compared for applicability and the most applicable property is chosen as described above. If several properties are equally applicable, this is an overload ambiguity as usual;
- Second, for the property selected at first step, the most applicable operator `invoke` overload is chosen.

11.8 Resolving callable references

Callable references introduce a special case of overload resolution which is somewhat similar to how regular calls are resolved, but different in several important aspects.

First, property and function references are treated equally, as both kinds of references have a type which is a subtype of a function type. Second, the type information needed to perform the resolution steps is acquired from *expected type* of the reference itself, rather than the types of arguments and/or result. The `invoke` operator convention **does not** apply to callable reference candidates. Third, and most important, is that, in the case of a call with a callable reference as a parameter, the resolution is **bidirectional**, meaning that both the callable being called and the callable being referenced are to be resolved *simultaneously*.

11.8.1 Resolving callable references not used as arguments to a call

In a simple case when the callable reference is not used as an argument to an overloaded call, its resolution is performed as follows:

- For each callable reference candidate, we perform the following steps:
  - We build its type constraints and add them to the constraint system of the expression the callable reference is used in;
  - A callable reference is deemed applicable if the constraint system is sound;
- For all applicable candidates, the resolution sets are built according to the same rules as building OCS for regular calls;
- If the highest priority set contains more than one callable, this is an overload ambiguity and should be reported as a compile-time error.
- Otherwise, the single callable in the set is chosen as the result of the resolution process.

  Note: this is different from the overload resolution for regular calls in that no most specific candidate selection process is performed inside the sets

  Important: when building the OCS for a callable reference, `invoke` operator convention does not apply, and all property references are treated equally as function references, being placed in the same sets. For example, consider the following code:

```
fun foo() = 1
val foo = 2
...
val y = ::foo
```

  Here both function `foo` and property `foo` are valid candidates for the callable reference and are placed *in the same candidate set*, thus producing an overload ambiguity. It is not important whether there is a suitable `invoke` operator available for the type of property `foo`.

  Example: consider the following two functions:

```
fun foo(i: Int): Int = 2          // (1)
fun foo(d: Double): Double = 2.0 // (2)
```

  In the following case:

```
val x: (Int) -> Int = ::foo
```

  candidate (1) is picked, because (assuming CRT is the type of the callable reference) the constraint CRT <: FT(`kotlin.Int`) → `kotlin.Int` is built and only candidate (1) is applicable w.r.t. this constraint.

  In another case:

```
fun bar(f: (Double) -> Double) {}

bar(::foo)
```

candidate (2) is picked, because (assuming CRT is the type of the callable reference) the constraint CRT <: FT(`kotlin.Double`) → `kotlin.Double` is built and only candidate (2) is applicable w.r.t. this constraint.

Please note that no bidirectional resolution is performed here as there is only one candidate for `bar`. If there were more than one candidate, the bidirectional resolution process would apply, possibly resulting in an overload resolution failure.

11.8.2 Bidirectional resolution for callable calls

If a callable reference (or several callable references) is itself an argument to an overloaded function call, the resolution process is performed for both callables *simultaneously*.

Assume we have a call `f(::g, b, c)`.

1. For each overload candidate `f`, a separate overload resolution process is completed as described in other parts of this section, up to the point of picking the most specific candidate. During this process, the only constraint for the callable reference `::g` is that it is an argument of a function type;
2. For the most specific candidate `f` found during the previous step, the overload resolution process for `::g` is performed as described here and the most specific candidate for `::g` is selected.

   Note: this may result in selecting the most specific candidate for `f` which has no available candidates for `::g`, meaning the bidirectional resolution process fails when resolving `::g`.

When performing bidirectional resolution for calls with multiple callable reference arguments, the algorithm is exactly the same, with each callable reference resolved separately in step 2. This ensures the overload resolution process for every callable being called is performed only once.

11.9 Type inference and overload resolution

Type inference in Kotlin is a very complicated process, and it is performed *after* overload resolution is done; meaning type inference may not affect the way overload resolution candidate is picked in any way.

Note: if we had allowed interdependence between type inference and overload resolution, we would have been able to create an infinitely oscillating behaviour, leading to an infinite compilation.

11.10 Conflicting overloads

In cases when it is known two callables are definitely interlinked in overload resolution (e.g., two member function-like callables declared in the same classifier), meaning they will always be considered together for overload resolution, Kotlin compiler performs *conflicting overload* detection for such callables.

Two callables `f` and `g` are *definitely interlinked* in overload resolution, if the following are true.

- `f` is not overriding `g` (and vice versa);
- `f` and `g` belong to the same level of c-level partition;
- `f` and `g` are declared in the same scope.

Different platform implementations may extend which callables are considered as definitely interlinked.

Two definitely interlinked callables `f` and `g` may create a *overload conflict*, if they could result in an overload ambiguity on most regular call sites.

To check whether such situation is possible, we compare `f` and `g` w.r.t. their applicability for a phantom call site with a fully specified argument list (i.e., with no used default arguments). If both `f` and `g` are equally or more specific to each other and neither of them is selected by the additional steps of MSC selection, we have an overload conflict.

Different platform implementations may extend which callables are considered as conflicting overloads.

Chapter 12

Control- and data-flow analysis

Several Kotlin features such as variable initialization analysis and smart casting analysis require performing control- and data-flow analyses. This section describes them and their applications.

12.1 Control flow graph

We define all control-flow analyses for Kotlin on a classic model called a control-flow graph (CFG). A CFG of a program is a graph which loosely defines all feasible paths the flow of a particular program can take during execution. All CFGs given in this section are *intraprocedural*, meaning that they describe the flow inside a *single* function, not taking function calls into account. CFG may, however, include multiple function bodies if said functions are *declared* inside each other (as is the case for lambdas).

The following sections describe CFG *fragments* associated with a particular Kotlin code construct. These fragments are introduced using visual notation rather than relational notation to simplify the understanding of the graph structure. To represent intermediate values created during computation, we use *implicit registers*, denoted `$1`, `$2`, `$3`, etc. These are considered to be unique in each CFG fragment (assigning the same register twice in the same CFG may only occur in unrelated program paths) and in the complete CFG, too. The numbers given are only notational.

We introduce special `eval` nodes, represented in *dashed lines*, to connect CFG fragments into bigger fragments. `eval x` here means that this node must be replaced with the whole CFG fragment associated with `x`. When this replacement

is performed, the value produced by `eval` is the same value that the meta-register `$result` holds in the corresponding fragment. All incoming edges of a fragment are connected to the incoming edges of the `eval` node, while all outgoing edges of a fragment are connected to the outgoing edges of the `eval` node. It is important, however, that, if such edges are absent either in the fragment or in the `eval` node, they (edges) are removed from the CFG.

We also use the `eval b` notation where `b` is not a single statement, but rather a control structure body. The fragment for a control structure body is the sequence of fragments for its statements, connected in the program order.

Some of the fragments have two kinds of outgoing edges, labeled `t` and `f` on the pictures. In a similar fashion, some `eval` nodes have two outgoing edges with the same labels. If such a fragment is inserted into such a node, only edges with matching labels are merged into each other. If either the fragment or the node have only unlabeled outgoing edges, the process is performed same as above.

For some types of analyses, it is important which boolean conditions hold on a control flow path. We use special `assume` nodes to introduce these conditions. `assume x` means that boolean condition `x` is always `true` when program flow passes through this particular node.

Some nodes are *labeled*, similarly to how statements may be labeled in Kotlin. Labeled nodes are considered CFG-unique and are handled as follows: if a fragment mentions a particular labeled node, this node is the same as any other node with this label in the complete CFG (i.e., a singular actual node is shared between all its labeled references). This is important when building graphs representing loops.

There are two other special kinds of nodes: `unreachable` nodes, signifying unreachable code, and `backedge` nodes, important for some kinds of analyses.

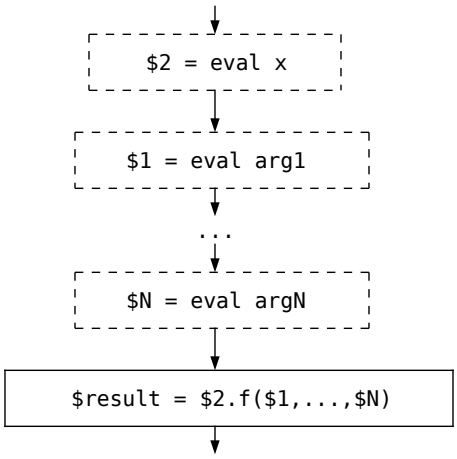
12.1.1 Expressions

Simple expressions, like literals and references, do not affect the control-flow of the program in any way and are irrelevant w.r.t. CFG.

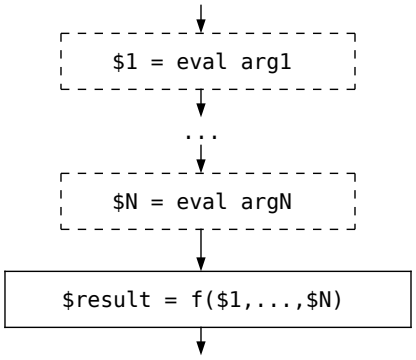
Function calls and operators

> Note: we do not consider operator calls as being different from function calls, as they are just special types of function calls. Henceforth, they are not treated separately.

```
x.f(arg1,..., argN)
```



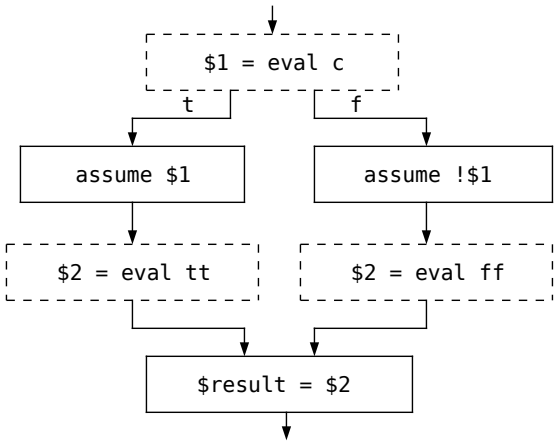

```
f(arg1,..., argN)
```



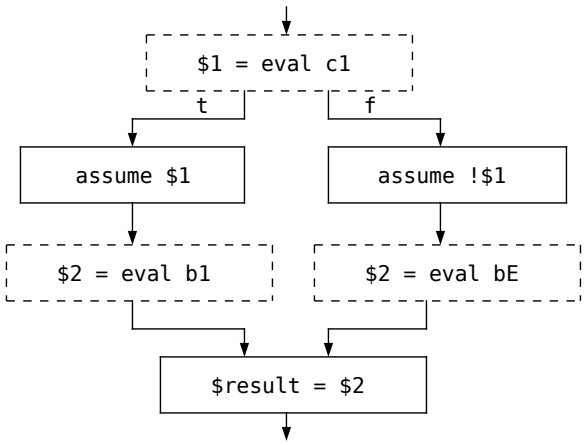
Conditional expressions

> Note: to simplify the notation, we consider only `if`-expressions with both branches present. Any `if`-statement in Kotlin may be trivially turned into such an expression by replacing the missing `else` branch with a `kotlin.Unit` object expression.

```
if(c) tt else ff
```




```
when {
    c1 -> b1
    else -> bE
}
```



Important: we only consider `when` expressions having exactly two branches for simplicity. A `when` expression with more than two branches may be trivially desugared into a series of nested when expression as follows:

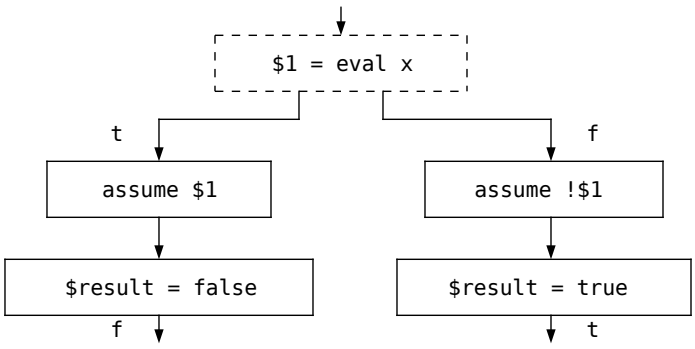
```
when {
    <entry1>
    <entries...>
    else -> bE
}
```

is the same as

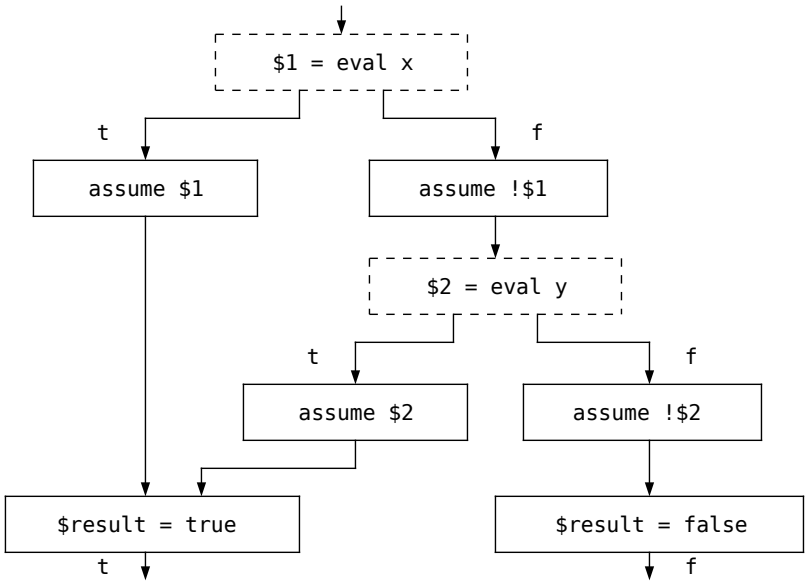
```
when {
    <entry1>
    else -> {
        when {
            <entries...>
            else -> bE
        }
    }
}
```

Boolean operators

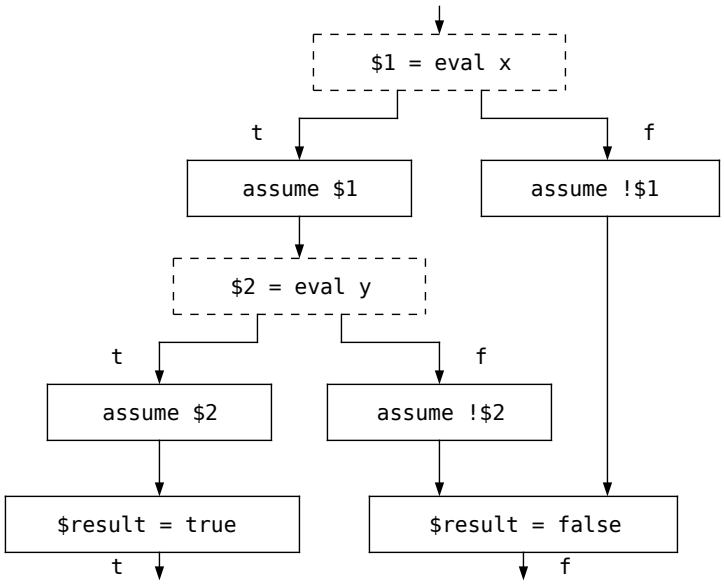
```
!x
```




```
x || y
```

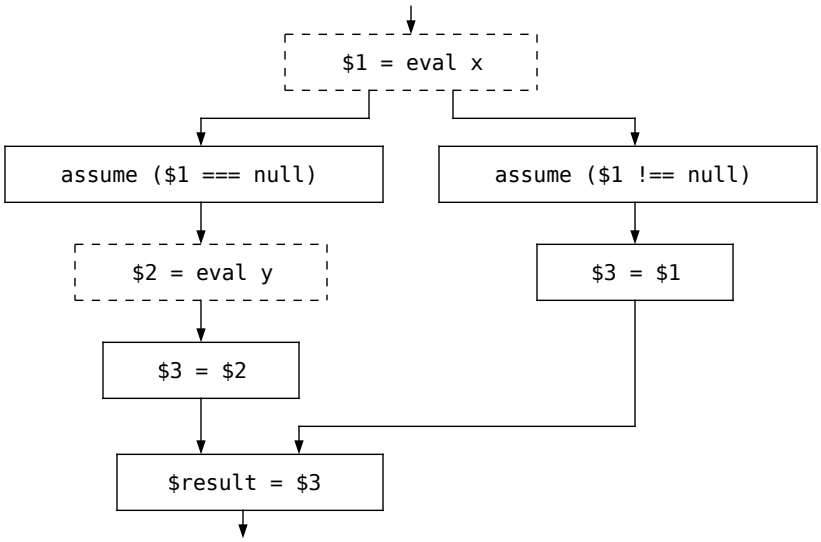


`x && y`

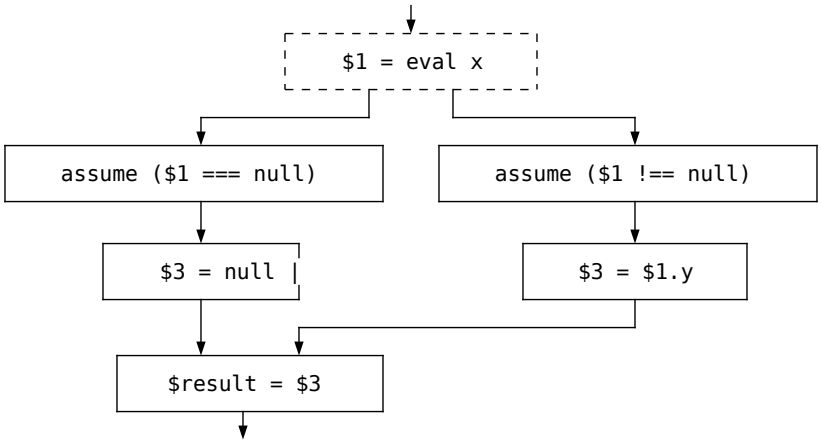


Other expressions

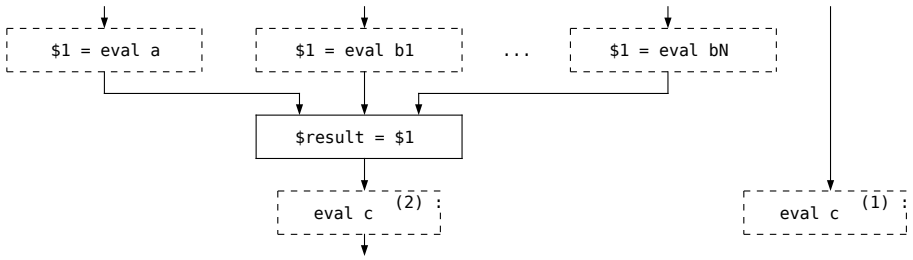
`x ?: y`



x?.y

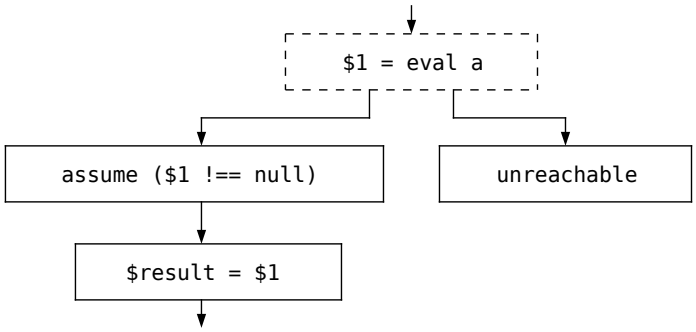



```
try { a... }
catch (e1: T1) { b1... }
...
catch (eN: TN) { bN... }
finally { c... }
```

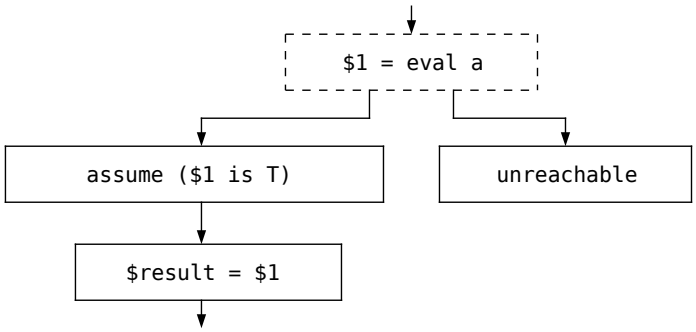


Important: in this diagram we consider `finally` block *twice*. The (1) block is used when handling the `finally` block and its body. The (2) block is used when considering the `finally` block w.r.t. rest of the CFG.

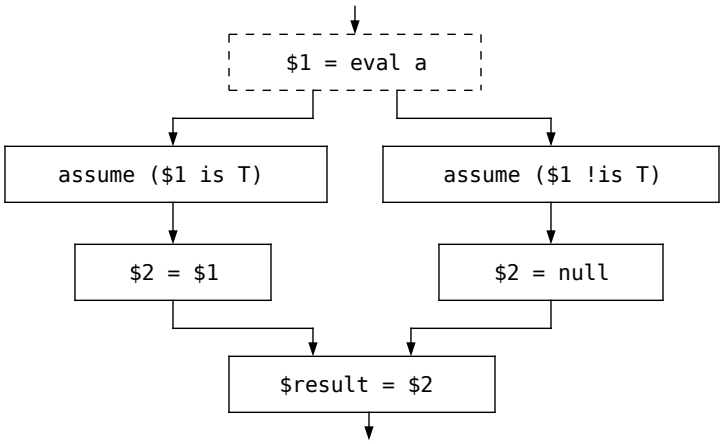
```
a!!
```



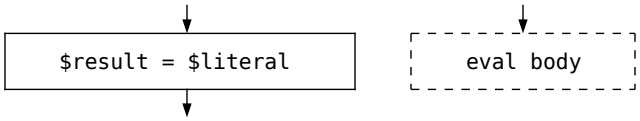

```
a as T
```



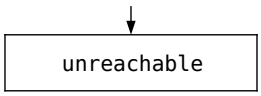

```
a as? T
```



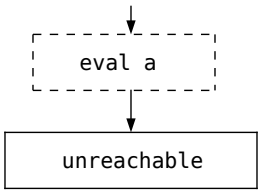

```
{ a: T ... -> body... }
```



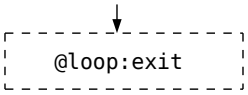

```
return
return@label
```



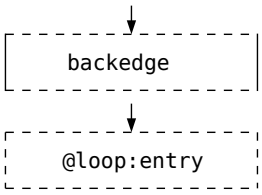

```
return a
return@label a
throw a
```




```
break@loop
```



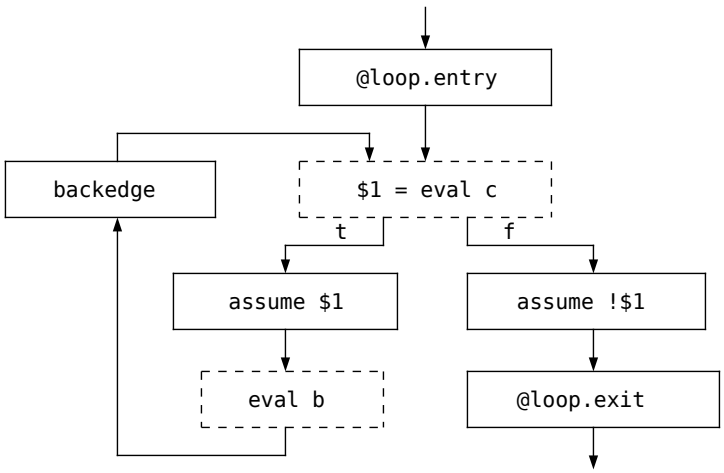

```
continue@loop
```



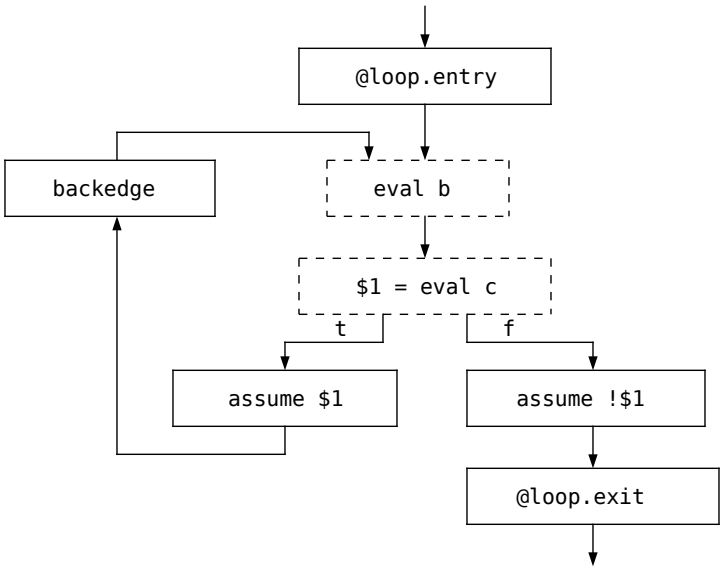
12.1.2 Statements

> Note: to simplify the notation, we consider only labeled loops, as unlabeled loops may be trivially turned into labeled ones by assigning them a unique label.

```
loop@ while(c) { b... }
```

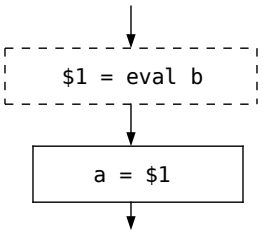



```
loop@ do { b... } while(c)
```

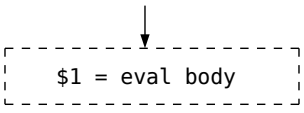


12.1.3 Declarations

```
var a = b
var a by b
val a = b
val a by b
```



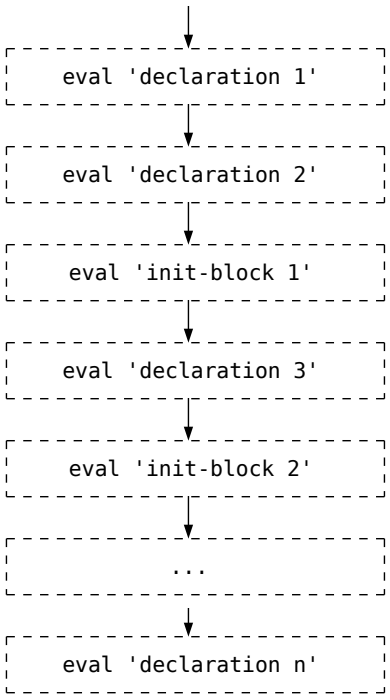

```
fun f() { body... }
```




```
class A (...) {
    'declaration 1'
    'declaration 2'
    'init-block 1'
```

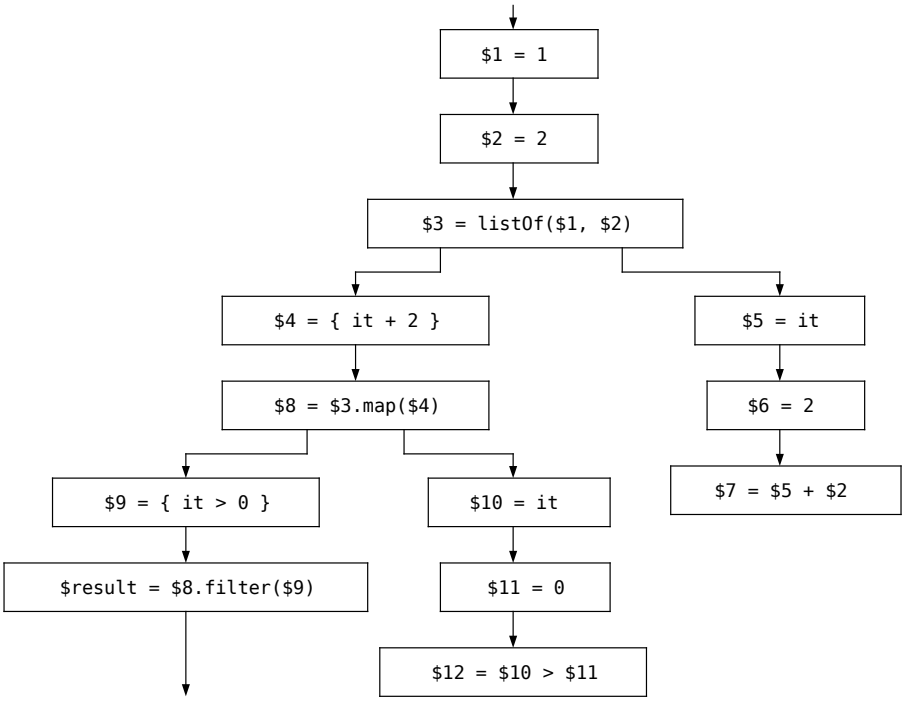
```
    'declaration 3'
    'init-block 2'
    ...
}
```

For every declaration and init block in a class body, the control flow is propagated through every element in the order of their appearance. Here we give a simplified example.

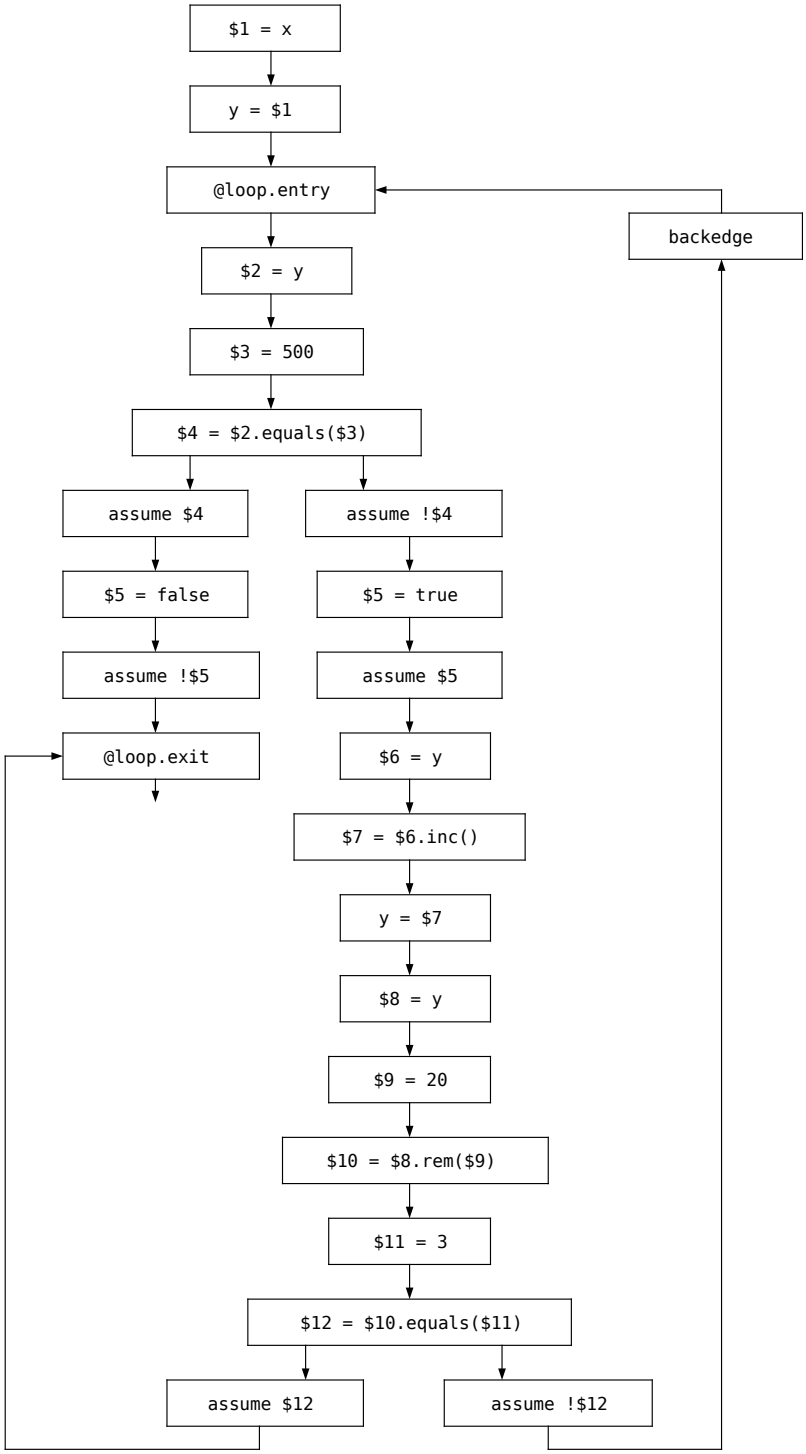


12.1.4 Examples

```
fun f() = listOf(1, 2).map { it + 2 }.filter { it > 0 }
```




```
fun f(x: Int) {
    var y = x
    loop@ while(y != 500) {
        y++
        if(y % 20 == 3) break@loop
    }
}
```


$1 = x
y = $1
@loop.entry
backedge
$2 = y
$3 = 500
$4 = $2.equals($3)
assume $4
assume !$4
$5 = false
$5 = true
assume !$5
assume $5
@loop.exit
$6 = y
$7 = $6.inc()
y = $7
$8 = y
$9 = 20
$10 = $8.rem($9)
$11 = 3
$12 = $10.equals($11)
assume $12
assume !$12

12.1.5 `kotlin.Nothing` and its influence on the CFG

As discussed in the type system section of this specification, `kotlin.Nothing` is an uninhabited type, meaning an instance of this type can never exist at runtime. For the purposes of control-flow graph (and related analyses) this means, as soon as an expression is known statically to have `kotlin.Nothing` type, all subsequent code is **unreachable**.

> Important: each specific analysis may decide to either use this information or ignore it for a given program. If unreachability from `kotlin.Nothing` is used, it can be represented in different ways, e.g., by changing the CFG structure or via *killDataFlow* instructions.

12.2 Performing analyses on the control-flow graph

The analyses defined in this document follow the pattern of analyses based on monotone frameworks, which work by modeling abstract program states as elements of lattices and joining these states using standard lattice operations. Such analyses may achieve limited path sensitivity via the analysis of conditions used in the `assume` nodes.

In short, an analysis is defined on the CFG by introducing:

- A lattice **S** (a partially ordered set that has both a greatest lower bound and a least upper bound defined for every pair of its elements) of values, called *abstract states*;
- A *transfer function* for mapping CFG nodes to the elements of **S**, essentially a set of rules on how to calculate an abstract state for each node of the CFG either directly or by using abstract states of other nodes.

The result of an analysis is a *fixed point* of the transfer function for each node of the given CFG, i.e., an abstract state for each node such that the transfer function maps the state to itself. For the particular shapes of the transfer function used in program analyses, given a finite **S**, the fixed point always exists, although the details of how this works go out of scope of this document.

12.2.1 Preliminary analysis and *killDataFlow* instruction

Some analyses described further in this document are based on special instruction called *killDataFlow*(v) where v is a program variable. These are not present in the graph representation described above and need to be inferred before such analyses may actually take place.

killDataFlow inference is based on a standard control-flow analysis with the lattice of natural numbers over "min" and "max" operations. That is, for every assignable property x an element of this lattice is a natural number N, with the least upper bound of two numbers defined as maximum function and the greatest lower bound as minimum function.

> Note: such lattice has 0 as its bottom element and does not have a top element.

We assume the following transfer functions for our analysis.

$$[\![\texttt{x = y}]\!](s) = s[x \to s(x) + 1]$$

$$[\![\texttt{backedge}]\!](s) = \{\star \to 0\}$$

$$[\![l]\!](s) = \bigsqcup_{p \in predecessor(l)} [\![p]\!](s)$$

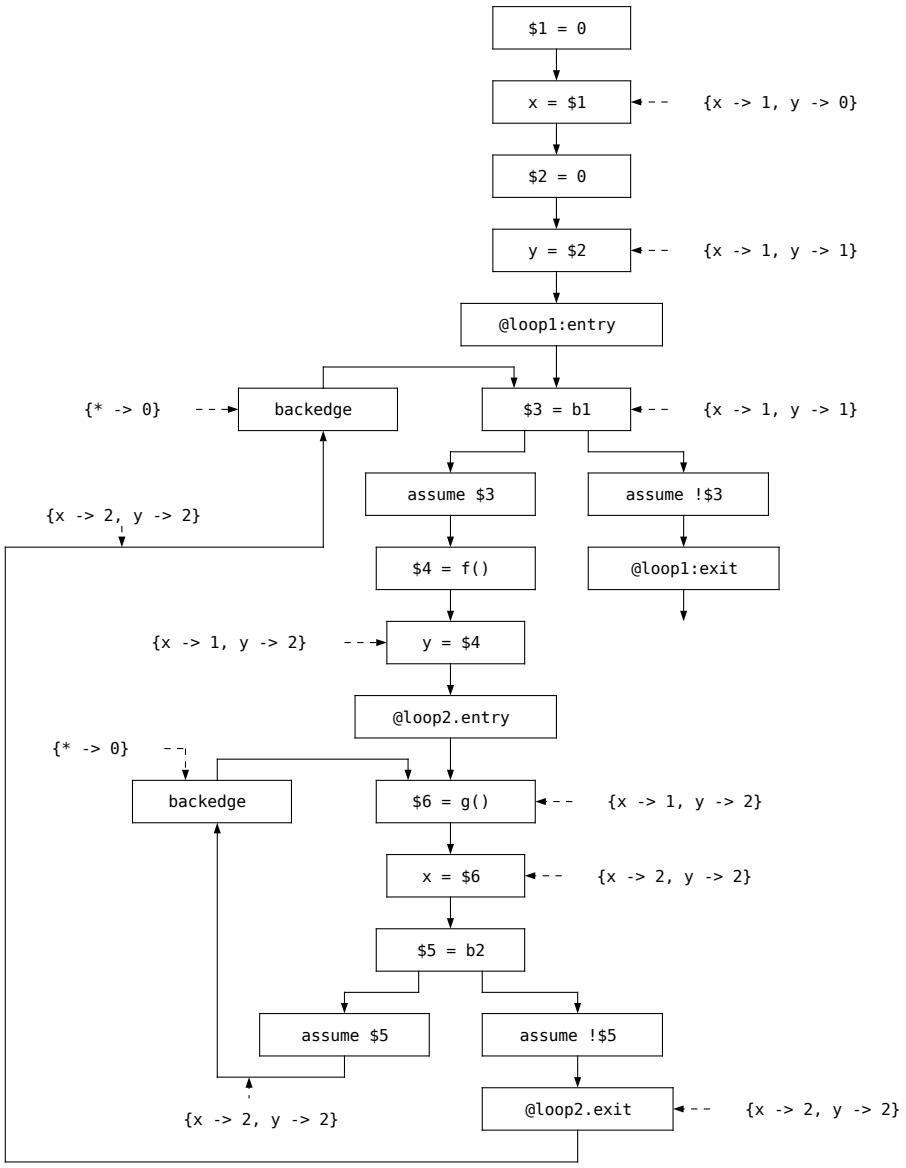
After running this analysis, for every backedge b and every variable x present in s, if $\exists b_p, b_s : b_p \in predecessors(b) \wedge b_s \in successors(b) \wedge [\![b_p]\!](x) > [\![b_s]\!](x)$, a $killDataFlow(x)$ instruction must be inserted after b.

> Informally: this somewhat complicated condition matches variables which have been assigned to in the loop body w.r.t. this loop's backedge.
>
> Note: this analysis does involve a possibly **infinite** lattice (a lattice of natural numbers) and may seem to diverge on some graphs. However, if we assume that every backedge in an arbitrary CFG is marked with a `backedge` instruction, it is trivial to prove that no number in the lattice will ever exceed the number of assignments (which is **finite**) in the analyzed program as any loop in the graph will contain at least one backedge.

As an example, consider the following Kotlin code:

```
var x: Int = 0
var y: Int = 0
while (b1) {
    y = f()
    do {
        x = g()
    } while (b2)
}
```

which results in the following CFG diagram (annotated with the analysis results where it is important):



There are two backedges: one for the inner loop (the inner backedge) and one for the outer loop (the outer backedge). The inner backedge has one predecessor with state $\{\mathtt{x} \to 2, \mathtt{y} \to 2\}$ and one successor with state $\{\mathtt{x} \to 1, \mathtt{y} \to 2\}$ with the value for `x` being less in the successor, meaning that we need to insert *killDataFlow*(`x`) after the backedge. The outer backedge has one predecessor with state $\{\mathtt{x} \to 2, \mathtt{y} \to 2\}$ and one successor with state $\{\mathtt{x} \to 1, \mathtt{y} \to 1\}$ with values for both variables being less in the successor, meaning we need to insert *killDataFlow*(`x`) and *killDataFlow*(`y`) after the backedge.

12.2.2 Variable initialization analysis

Kotlin allows non-delegated properties to not have initializers in their declaration as long as the property is *definitely assigned* before its first usage. This property is checked by the variable initialization analysis (VIA). VIA operates on abstract values from a flat *assignedness* lattice of two values $\{Assigned, Unassigned\}$. The analysis itself uses abstract values from a map lattice of all property declarations to their abstract states based on the assignedness lattice. The abstract states are propagated in a forward manner using the standard join operation to merge states from different paths.

The CFG nodes relevant to VIA include only property declarations and direct property assignments. Every property declaration adds itself to the domain by setting the *Unassigned* value to itself. Every direct property assignment changes the value for this property to *Assigned*.

The results of the analysis are interpreted as follows. For every property, any usage of the said property in any statement is a compile-time error unless the abstract state of this property at this statement is *Assigned*. For every read-only property (declared using `val` keyword), any assignment to this property is a compile-time error unless the abstract state of this property is *Unassigned*.

As an example, consider the following Kotlin code:

```
/*  1 */ val x: Int     //{x → Unassigned, ⋆ → ⊥}
/*  2 */ var y: Int     //{x → Unassigned, y → Unassigned, ⋆ → ⊥}
/*  3 */ if (c) {        //
/*  4 */     x = 40     //{x → Assigned, y → Unassigned, ⋆ → ⊥}
/*  5 */     y = 4      //{x → Assigned, y → Assigned, ⋆ → ⊥}
/*  6 */ } else {       //
/*  7 */     x = 20     //{x → Assigned, y → Unassigned, ⋆ → ⊥}
/*  8 */ }             //{x → Assigned, y → ⊤, ⋆ → ⊥}
/*  9 */ y = 5         //{x → Assigned, y → Assigned, ⋆ → ⊥}
/* 10 */ val z = x + y //{x → Assigned, y → Assigned, z → Assigned}
```

There are no incorrect operations in this example, so the code does not produce any compile-time errors.

Let us consider another example:

```
/* 1 */ val x: Int     //{x → Unassigned, ⋆ → ⊥}
/* 2 */ var y: Int     //{x → Unassigned, y → Unassigned, ⋆ → ⊥}
/* 3 */ while (c) {     //{x → ⊤, y → ⊤, ⋆ → ⊥} Error!
/* 4 */     x = 40     //{x → ⊤, y → ⊤, ⋆ → ⊥}
/* 5 */     y = 4      //{x → ⊤, y → ⊤, ⋆ → ⊥}
/* 6 */ }              //
/* 7 */ val z = x + y //{x → ⊤, y → ⊤, ⋆ → ⊥} More errors!
```

In this example, the state of both properties at line 3 is $\top$, as it is the least upper bound of the states from lines 5 and 2 (from the `while` loop), which is derived to be $\top$. This leads to a compile-time error at line 4 for `x`, because one cannot reassign a read-only property.

At line 7 there is another compile-time error when both properties are used, as there are paths in the CFG which reach line 7 when the properties have not been assigned (i.e., the case when the `while` loop body was skipped).

12.2.3 Smart casting analysis

See the corresponding section for details.

12.2.4 Function contracts

> Note: as of Kotlin 1.5.0, contracts for user-defined functions are an experimental feature and, thus, not described here

Some standard-library functions in Kotlin are defined in such a way that they adhere to a specific *call contract* that affects the way calls to such functions are analysed from the perspective of the caller's control flow graph. A function's call contract consists of one or more *effects*.

There are several kinds of effects:

- Calls-in-place effect for a function-type parameter of the function;
- Returns-implies-condition effect for a boolean parameter of the function;
- Particular implementations may introduce other types of effects.

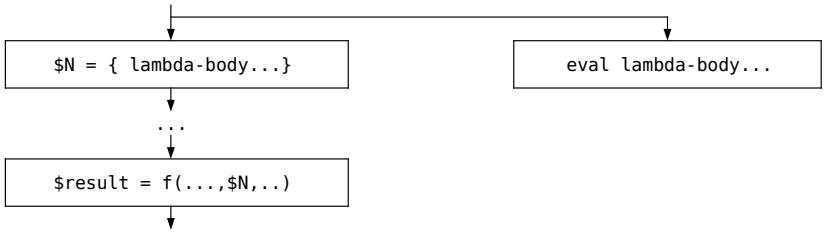
Calls-in-place effect of function F for a function-type parameter P specifies that for every call of F parameter P will be also invoked as a function. This effect may also have one of the three invocation types:

- *At-least-once*, meaning that P will be invoked at least once;
- *Exactly-once*, meaning that P will be invoked exactly once;
- *At-most-once*, meaning that P will be invoked at most once.

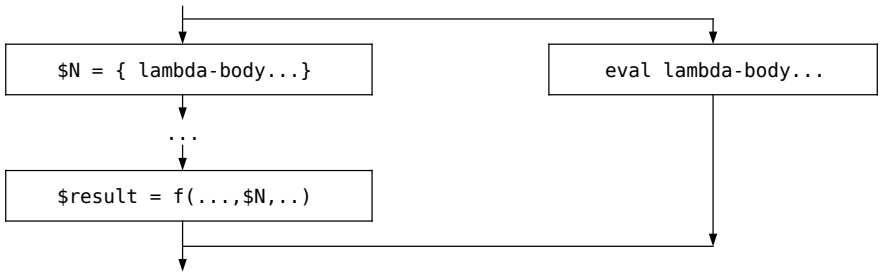
These effects change the call graph that is produced for a function call of F when supplied a lambda-expression parameter for P. Without any effect, the graph looks like this:

For a function call

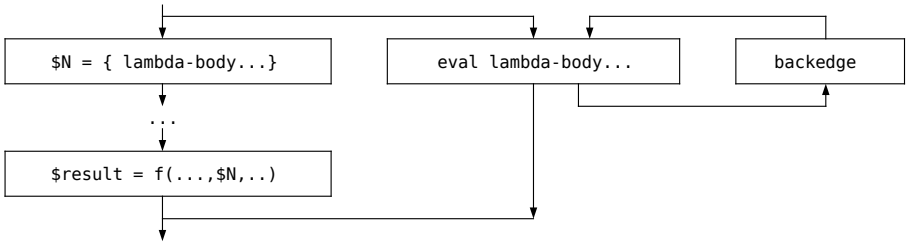
```
f(..., { lambda-body... }, ...)
```



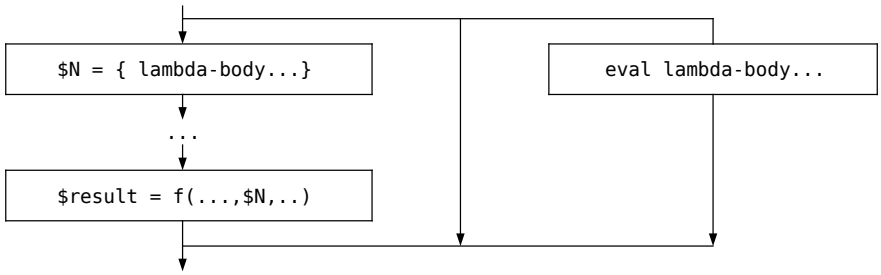
Please note that control flow information is passed inside the lambda body, but no information is extracted from it. If the corresponding parameter P is introduced with *exactly-once* effect, this changes to:



If the corresponding parameter P is introduced with *at-least-once* effect, this changes to:



If the corresponding parameter P is introduced with *at-most-once* effect, this changes to:



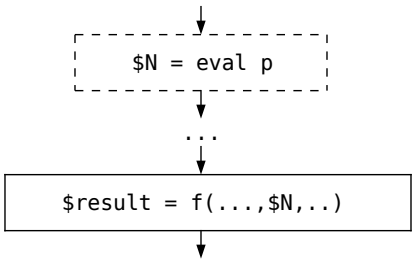
This allows the control-flow information to be extracted from lambda expression

according to the policy of its invocation.

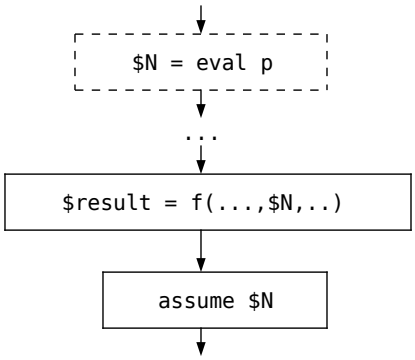
Returns-implies-condition effect of function F for a boolean parameter P specifies that if, when invoked normally, a call to F returns, P is assumed to be true. For a function call

```
f(..., p, ...)
```

this changes normal call graph that looks like this:



to look like this:



The following standard library functions have contracts with the following effects:

- `kotlin.run`, `kotlin.with`, `kotlin.let`, `kotlin.apply`, `kotlin.also` (all overloads): calls-in-place effect with invocation kind "exactly-once" for its functional argument;
- `kotlin.check`, `kotlin.require` (all overloads): returns-implies-condition effect on the boolean parameter.

Examples:

This code would result in a initialized variable analysis violation if `run` was not a standard function with corresponding contract:

```
val x: Int
run { // run invokes its argument exactly once
```

```
    x = 4
}
// could be error: x is not initialized
// but is ok
println(x)
```

Several examples of contract-introduced smart-cast:

```
val x: Any = ...
check(x is Int)
// x is known to be Int thanks to assume introduced by
// the contract of check
val y = x + 4 // would be illegal without contract
```

```
val x: Int? = ...
// x is known to be non-null thanks to assume introduced by
// the contract of require
require(x != null)
val y = x + 4 // would be illegal without contract
```

References

1. Frances E. Allen. “Control flow analysis.” ACM SIGPLAN Notices, 1970.
2. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. “Principles of program analysis.” Springer, 2015.
3. Kam, John B., and Jeffrey D. Ullman. “Monotone data flow analysis frameworks.” Acta informatica 7.3 (1977): 305-317.

Chapter 13

Kotlin type constraints

Some complex tasks that need to be solved when compiling Kotlin code are formulated best using *constraint systems* over Kotlin types. These are solved using constraint solvers.

13.1 Type constraint definition

A *type constraint* in general is an inequation of the following form: $T <: U$ where T and U are concrete Kotlin types. As Kotlin has parameterized types, T and U may be *free type variables*: unknown types which may be substituted by any other type in Kotlin.

Please note that, in general, not all type parameters are considered as free type variables in a constraint system. Some type variables may be *fixed* in a constraint system; for example, type parameters of a parameterized class *inside* its body are unknown types, but are not free type variables either. A fixed type variable describes an unknown, but fixed type which is *not* to be substituted.

We will use the notation T_i for a type variable and $\tilde{T}_i$ for a fixed type variable. The main difference between fixed type variables and concrete types is that different concrete types may not be equal, but a fixed type variable may be equal to another fixed type variable or a concrete type.

Examples of valid type constraints:

- `List<`$\widetilde{X}$`>` $<:$ Y
- `List<`$\widetilde{X}$`>` $<:$ `List<List<Int>>`
- $\widetilde{X} <: Y$

Every constraint system has general implicit constraints $T_j <:$ `kotlin.Any?` and `kotlin.Nothing` $<: T_j$ for every type T_j mentioned in the system, including

type variables.

13.2 Type constraint solving

There are two tasks which a type constraint solver may perform: checking constraint system for soundness, i.e., if a solution exists, and solving constraint system, i.e., inferring a satisfying substitution of concrete types for all free type variables.

Checking a constraint system for soundness can be viewed as a much simpler case of solving that constraint system: if there is a solution, the system is sound, meaning there are only two possible outcomes. Solving a constraint system, on the other hand, may have multiple possible outcomes, as there may be multiple valid solutions.

> Example: constraint systems which are sound yet no relevant solutions exist.
>
> - $X <: Y$
> - `List<`X`>` $<:$ `Collection<`X`>`

13.2.1 Checking constraint system soundness

Checking constraint system soundness is a satisfiability problem. That is, given a number of constraints in the form $S <: T$ containing zero or more free type variables (also called *inference type variables*), it needs to determine if these constraints are non-contradictory, i.e., if there exists a possible instantiation of these free variables to concrete types which makes all given constraints valid.

This problem can be reduced to finding a set of lower and upper bounds for each of these variables and determining if these bounds are non-contradictory. The algorithm of finding these bounds is implementation-defined and is not guaranteed to prove the satisfiability of given constraints in all possible cases.

A sample bound inference algorithm

The algorithm given in this section is just an example of a family of algorithms that may be applied to the problem given above. A particular implementation is not guaranteed to follow this algorithm, but one may use it as a reference on how this problem may be approached.

> Note: a well-informed reader may notice this algorithm to be similar to the one used by Java. This is not a coincidence: our sample inference algorithm has indeed been inspired by Java's.

The algorithm works in two phases: *reduction* and *incorporation* which are applied to the constraint system and its current solution in turns until a fixpoint or an error is reached (aka reduction-incorporation procedure or RIP). The reduction phase is used to produce bounds for inference variables based on constraints; this phase is also responsible for eliminating the constraints which are no longer needed. The incorporation phase is used to introduce new bounds and constraints from existing bounds.

A bound is similar to a constraint in that it has the form $S <: T$, at least one of S or T is an inference variable. Thus, the current (and also the final) solution is a set of upper and lower bounds for each inference variable. A *resolved type* in this context is any type which does not contain inference variables.

Reduction phase: for each constraint $S <: T$ in the constraint system the following rules are applied:

- If S and T are resolved types and:
  - If $S <: T$, this constraint is eliminated;
  - Otherwise, this is an inference error;
- Otherwise, if S is an inference variable α, a new bound $\alpha <: T$ is added to current solution;
- Otherwise, if T is an inference variable β, a new bound $S <: \beta$ is added to current solution;
- Otherwise, if S is a flexible type of the form $(\alpha..\alpha?)$ where α is an inference variable, a new bound $\alpha <: (T..T?)$ is added to current solution;
- Otherwise, if T is a flexible type of the form $(\alpha..\alpha?)$ where α is an inference variable, a new bound $(S..S?) <: \alpha$ is added to current solution;
- Otherwise, if S is a nullable type of the form $A?$ and:
  - If T is a known non-nullable type (a classifier type, a nullability-asserted type $B!!$, a type variable with a known non-nullable lower bound, or an intersection type containing a known non-nullable type), this is an inference error;
  - Otherwise, if T is also a nullable type of the form $B?$, the constraint is reduced to $A <: B$;
  - Otherwise, the constraint is reduced to $A <: T$;
- Otherwise, if S is a flexible type of the form $(B..A?)$ and:
  - If T is a nullable type of form $C?$, the constraint is reduced to $(B..A) <: C$, or to $A <: C$ if $A \equiv B$;
  - Otherwise, the constraint is reduced to $(B..A) <: T$, or to $A <: T$ if $A \equiv B$;
- Otherwise, if T is a parameterized type $G[A_1, \ldots, A_N]$, among all supertypes of S the one of the form $G[B_1, \ldots, B_N]$ is chosen.
  - If no such supertype exists, this is an inference error;
  - Otherwise, for each $M \in [1, N]$, a type argument constraint for containment $A_M \preceq B_M$ is introduced (see below);
- Otherwise, if T is any other classifier type and T is among supertypes for S, the constraint is eliminated; otherwise, this is an inference error;

- Otherwise, if T is a type variable and:
  - If S is an intersection type containing T, this constraint is eliminated;
  - Otherwise, if T has a lower bound B, the constraint is reduced to $S <: B$;
  - Otherwise, this is an inference error;
- Otherwise, if T is an intersection type $A_1 \,\&\, \ldots \,\&\, A_N$, the constraint is reduced to N constraints $S <: A_M$ for each $M \in [1, N]$;
- Otherwise, if T is a nullable type of the form $B?$ and:
  - If S is a known non-nullable type (a classifier type, a nullability-asserted type $A!!$, a type variable with a known non-nullable lower bound, or an intersection type containing a known non-nullable type), the constraint is reduced to $S <: B$;
  - Otherwise, this is an inference error.

Type argument constraints for a containment relation $Q \preceq F$ are constructed as follows:

> Important: for the purposes of this algorithm, declaration-site variance type arguments are considered to be their equivalent use-site variance versions.

- If either Q or F is a special bivariant type argument $\star$, no constraints are produced;
- If F has the form F' (is invariant):
  - If Q is also invariant and of the form Q', two constraints are produced: $F' <: Q'$ and $Q' <: F'$;
  - If Q has any other variance, this is an inference error;
- If F has the form `out` F' (is covariant):
  - If Q has the form `out` Q' or Q', the following constraint is produced: $Q' <: F'$;
  - If Q has the form `in` Q', the following constraint is produced: `kotlin.Any?` $<: F'$;
- If F has the form `in` F' (is contravariant):
  - If Q has the form `in` Q' or Q', the following constraint is produced: $F' <: Q'$;
  - If Q has the form `out` Q', the following constraint is produced: $F' <:$ `kotlin.Nothing`.

Incorporation phase: for each bound and particular bound combinations in the current solution, new constraints are produced as follows (it is safe to assume that each constraint is introduced into the system only once, so if this step produces constraints that have already been reduced, they are not added into the system):

- For each inference variable α, for each pair of bounds $S <: \alpha$ and $\alpha <: T$, a new constraint is produced: $S <: T$;
- For each inference variable α, if there is a pair of bounds $S <: \alpha$ and $\alpha <: S$ (i.e., α is equivalent to S), for each bound $Q <: P$ where Q or P

contains α, a new constraint is produced: $Q[\alpha := S] <: P[\alpha := S]$;
- For each inference variable α, for each pair of bounds $\alpha <: S$ and $\alpha <: T$ where S has a supertype of the form $G[A_1, \ldots, A_N]$ and T has a matching supertype of the form $G[B_1, \ldots, B_N]$, for each matching supertype G and each $M \in [1, N]$, if both A_M and B_M are invariant and have forms A'_M and B'_M respectively, the following new constraints are produced: $A'_M <: B'_M$ and $B'_M <: A'_M$.

13.2.2 Finding optimal constraint system solution

As any constraint system may have multiple valid solutions, finding one which is "optimal" in some sense is not possible in general, because the notion of the best solution for a task depends on the said task. To deal with this, a constraint system allows two additional types of constraints:

- A *pull-up* constraint for type variable T, denoted $\uparrow T$, signifying that when finding a substitution for this variable, the optimal solution is the largest one according to subtyping relation;
- A *push-down* constraint for type variable T, denoted $\downarrow T$, signifying that when finding a substitution for this variable, the optimal solution is the smallest one according to subtyping relation.

If a variable has no constraints of these kinds associated with it, it is assumed to have a pull-up implicit constraint. The process of instantiating the free variables of a constraint system starts by finding the bounds for each free variable (as mentioned in the previous section) and then, given these bounds, continues to pick the right type from them. Excluding other free variables, this boils down to:

- For a variable with a push-down constraint, the solution is the greatest lower bound of all upper bounds for this variable, excluding other free variables;
- For a variable with a pull-up constraint, the solution is the least upper bound of all lower bounds for this variable, excluding other free variables;
- For a variable with both or none, the solution is also the least upper bound of all lower bounds for this variable, excluding other free variables.

If there are inference variables dependent on other inference variables (α is dependent on β iff there is a bound $\alpha <: T$ or $T <: \alpha$ where T contains β), this process is performed in stages.

During each stage a set of inference variables not dependent on other inference variables (but possibly dependent on each other) is selected, the solutions for these variables are found using existing bounds, and after that these variables are **resolved** in the current bound set by replacing all of their instances in other bounds by the solution. This may trigger a new RIP.

After that, a new independent set of inference variables is picked and this process is repeated until an inference error occurs or a solution for each inference variable

is found.

13.2.3 The relations on types as constraints

In other sections (for example, Expressions and Statements) the relations between types may be expressed using the type operations found in the type system section of this document.

The greatest lower bound of two types is converted directly as-is, as the greatest lower bound is always an intersection type.

The least upper bound of two types is converted as follows. If type T is defined to be the least upper bound of A and B, the following constraints are produced:

- $A <: T$
- $B <: T$
- $\downarrow T$
- $\uparrow A$
- $\uparrow B$

  Important: the results of finding GLB or LUB via a constraint system may be different from the results of finding them via a normalization procedure (i.e., imprecise); however, they are sound w.r.t. bound, meaning a constraint system GLB is still a lower bound and a constraint system LUB is still an upper bound.

  Example:

  Let's assume we have the following code:

  ```
  val e = if (c) a else b
  ```

  where `a`, `b`, `c` are some expressions with unknown types (having no other type constraints besides the implicit ones).

  Assume the type variables generated for them are A, B and C respectively, the type variable for `e` is E. According to the conditional expression rules, this produces the following relations:

  - $C <:$ `kotlin.Boolean`
  - $E = \text{LUB}(A, B)$

  These, in turn, produce the following explicit constraints:

  - $C <:$ `kotlin.Boolean`
  - $A <: E$
  - $B <: E$
  - $\downarrow E$
  - $\uparrow A$
  - $\uparrow B$

which, w.r.t. general and pull-up implicit constraints, produce the following solution:

- $C \rightarrow$ `kotlin.Boolean`
- $A \rightarrow$ `kotlin.Any?`
- $B \rightarrow$ `kotlin.Any?`
- $E \rightarrow$ `kotlin.Any?`

Chapter 14

Type inference

Kotlin has a concept of *type inference* for compile-time type information, meaning some type information in the code may be omitted, to be inferred by the compiler. There are two kinds of type inference supported by Kotlin.

- Local type inference, for inferring types of expressions locally, in statement/expression scope;
- Function signature type inference, for inferring types of function return values and/or parameters.

Type inference is a type constraint problem, and is usually solved by a type constraint solver. For this reason, type inference is applicable in situations when the type context contains enough information for the type constraint solver to create an optimal constraint system solution w.r.t. type inference problem.

> Note: for the purposes of type inference, an optimal solution is the one which does not contain any free type variables with no explicit constraints on them.

Kotlin also supports flow-sensitive types in the form of smart casts, which have direct effect on type inference. Therefore, we will discuss them first, before talking about type inference itself.

14.1 Smart casts

Kotlin introduces a limited form of flow-sensitive typing called *smart casts*. Flow-sensitive typing means some expressions in the program may introduce changes to the compile-time types of variables. This allows one to avoid unneeded explicit casting of values in cases when their runtime types are guaranteed to conform to the expected compile-time types.

Flow-sensitive typing may be considered a specific instance of traditional data-flow analysis. Therefore, before we discuss it further, we need to establish the data-flow framework, which we will use for smart casts.

14.1.1 Data-flow framework

Smart cast lattices

We assume our data-flow analysis is run on a classic control-flow graph (CFG) structure, where most non-trivial expressions and statements are simplified and/or desugared.

Our data-flow domain is a map lattice SmartCastData = Expression $\to$ SmartCastType, where Expression is any Kotlin expression and SmartCastType = Type $\times$ Type sublattice is a product lattice of smart cast data-flow facts of the following kind.

- First component describes the type, which an expression definitely **has**
- Second component describes the type, which an expression definitely **does not have**

The sublattice order, join and meet are defined as follows.

$$P_1 \times N_1 \sqsubseteq P_2 \times N_2 \Leftrightarrow P_1 <: P_2 \wedge N_1 :> N_2$$

$$P_1 \times N_1 \sqcup P_2 \times N_2 = \mathrm{LUB}(P_1, P_2) \times \mathrm{GLB}(N_1, N_2)$$
$$P_1 \times N_1 \sqcap P_2 \times N_2 = \mathrm{GLB}(P_1, P_2) \times \mathrm{LUB}(N_1, N_2)$$

Note: a well-informed reader may notice the second component is behaving very similarly to a *negation* type.

$$\begin{aligned}(P_1 \,\&\, \neg N_1) \mid (P_2 \,\&\, \neg N_2) &\sqsubseteq (P_1 \mid P_2) \,\&\, (\neg N_1 \mid \neg N_2) \\ &= (P_1 \mid P_2) \,\&\, \neg(N_1 \,\&\, N_2) \\ (P_1 \,\&\, \neg N_1) \,\&\, (P_2 \,\&\, \neg N_2) &= (P_1 \,\&\, P_2) \,\&\, (\neg N_1 \,\&\, \neg N_2) \\ &= (P_1 \,\&\, P_2) \,\&\, \neg(N_1 \mid N_2)\end{aligned}$$

This is as intended, as "type which an expression definitely does not have" is exactly a negation type. In smart casts, as Kotlin type system does not have negation types, we overapproximate them when needed.

Smart cast transfer functions

The data-flow information uses the following transfer functions.

$$
\begin{array}{ll}
[\![assume(x\ \texttt{is}\ T)]\!]\,(s) & = s[x \to s(x) \sqcap (T \times \top)] \\
[\![assume(x\ \texttt{!is}\ T)]\!]\,(s) & = s[x \to s(x) \sqcap (\top \times T)] \\
\\
[\![x\ \texttt{as}\ T]\!]\,(s) & = s[x \to s(x) \sqcap (T \times \top)] \\
[\![x\ \texttt{!as}\ T)]\!]\,(s) & = s[x \to s(x) \sqcap (\top \times T)] \\
\\
[\![assume(x == null)]\!]\,(s) & = s[x \to s(x) \sqcap (\texttt{kotlin.Nothing?} \times \top)] \\
[\![assume(x \texttt{!=} null)]\!]\,(s) & = s[x \to s(x) \sqcap (\top \times \texttt{kotlin.Nothing?})] \\
\\
[\![assume(x === null)]\!]\,(s) & = s[x \to s(x) \sqcap (\texttt{kotlin.Nothing?} \times \top)] \\
[\![assume(x \texttt{!==} null)]\!]\,(s) & = s[x \to s(x) \sqcap (\top \times \texttt{kotlin.Nothing?})] \\
\\
[\![assume(x == y)]\!]\,(s) & = s[x \to s(x) \sqcap s(y), \\
 & \qquad y \to s(x) \sqcap s(y)] \\
[\![assume(x \texttt{!=} y)]\!]\,(s) & = s[x \to s(x) \sqcap swap(isNullable(s(y))), \\
 & \qquad y \to s(y) \sqcap swap(isNullable(s(x)))] \\
\\
[\![assume(x === y)]\!]\,(s) & = s[x \to s(x) \sqcap s(y), \\
 & \qquad y \to s(x) \sqcap s(y)] \\
[\![assume(x \texttt{!==} y)]\!]\,(s) & = s[x \to s(x) \sqcap swap(isNullable(s(y))), \\
 & \qquad y \to s(y) \sqcap swap(isNullable(s(x)))] \\
\\
[\![x = y]\!]\,(s) & = s[x \to s(y)] \\
\\
[\![killDataFlow(x)]\!]\,(s) & = s[x \to (\top \times \top)] \\
\\
[\![l]\!]\,(s) & = \bigsqcup_{p \in predecessor(l)} [\![p]\!]\,(s)
\end{array}
$$

where

$$swap(P \times N) = N \times P$$

$$isNullable(s) = \begin{cases} (\texttt{kotlin.Nothing?} \times \top) & \text{if } s \sqsubseteq (\texttt{kotlin.Nothing?} \times \top) \\ (\top \times \top) & \text{otherwise} \end{cases}$$

Important: transfer functions for `==` and `!=` are used only if the corresponding `equals` implementation is known to be equivalent to reference equality check. For example, generated `equals` implementation for data classes is considered to be equivalent to reference equality check.

Note: in some cases, after the CFG simplification a program location l may be duplicated and associated with several locations $l_1, \ldots, l_N$ in the resulting CFG. If so, the data-flow information for l is calculated as

$$[\![l]\!] = \bigsqcup_{i=1}^{N} [\![l_i]\!]$$

Note: a *killDataFlow* instruction is used to reset the data-flow information in cases, when a compiler deems necessary to stop its propagation. For example, it may be used in loops to speed up data-flow analysis convergence. This is the current behaviour of the Kotlin compiler.

After the data-flow analysis is done, for a program location l we have its data-flow information $[\![l]\!]$, which contains data-flow facts $[\![l]\!]\,[e] = (P \times N)$ for an expression e.

14.1.2 Smart cast types

The data-flow information is used to produce the smart cast type as follows.

First, smart casts may influence the compile-time type of an expression e (called *smart cast sink*) only if the sink is *stable*.

Second, for a stable smart cast sink e we calculate the overapproximation of its possible type.

$$[\![l]\!]\,[e] = (P \times N) \Rightarrow smartCastTypeOf(e) = typeOf(e) \& P \& approxNegationType(N)$$

$$approxNegationType(N) = \begin{cases} \texttt{kotlin.Any} & \text{if } \texttt{kotlin.Nothing?} <: N \\ \texttt{kotlin.Any?} & \text{otherwise} \end{cases}$$

As a result, $smartCastTypeOf(e)$ is used as a compile-time type of e for most purposes (including, but not limited to, function overloading and type inference of other values).

> Note: the most important exception to when smart casts are used in type inference is direct property declaration.

```
fun noSmartCastInInference() {
    var a: Any? = null

    if (a == null) return

    var c = a // Direct property declaration

    c // Declared type of `c` is Any?
      // However, here it's smart casted to Any
}

fun <T> id(a: T): T = a

fun smartCastInInference() {
    var a: Any? = null

    if (a == null) return

    var c = id(a)

    c // Declared type of `c` is Any
}
```

Smart casts are introduced by the following Kotlin constructions.

- Conditional expressions (`if`)
- When expressions (`when`);
- Elvis operator (operator `?:`);
- Safe navigation operator (operator `?.`);
- Logical conjunction expressions (operator `&&`);
- Logical disjunction expressions (operator `||`);
- Not-null assertion expressions (operator `!!`);
- Cast expressions (operator `as`);
- Type-checking expressions (operator `is`);
- Simple assignments;

- Platform-specific cases: different platforms may add other kinds of expressions which introduce additional smart cast sources.

  Note: property declarations are not listed here, as their types are derived from initializers.

  Note: for the purposes of smart casts, most of these constructions are simplified and/or desugared, when we are building the program CFG for the data-flow analysis. We informally call such constructions *smart cast sources*, as they are responsible for creating smart cast specific instructions.

14.1.3 Smart cast sink stability

A smart cast sink is *stable* for smart casting if its value cannot be changed via means external to the CFG; this guarantees the smart cast conditions calculated by the data-flow analysis still hold at the sink. This is one of the necessary conditions for smart cast to be applicable to an expression.

Smart cast sink stability breaks in the presence of the following aspects.

- concurrent writes;
- mutable value capturing;
- separate module compilation;
- custom getters;
- delegation.

The following smart cast sinks are considered stable.

1. Immutable local or classifier-scope properties without delegation or custom getters;
2. Mutable local properties without delegation or custom getters, if the compiler can prove that they are effectively immutable, i.e., cannot be changed by external means;
3. Immutable properties of immutable stable properties without delegation or custom getters, if they are declared in the current module.

Effectively immutable smart cast sinks

We will call redefinition of e ***direct*** redefinition, if it happens in the same declaration scope as the definition of e. If e is redefined in a nested declaration scope (w.r.t. its definition), this is a ***nested*** redefinition.

> Note: informally, a nested redefinition means the property has been captured in another scope and may be changed from that scope in a concurrent fashion.

We define ***direct*** and ***nested*** smart cast sinks in a similar way.

Example:

```
fun example() {
    // definition
    var x: Int? = null

    if (x != null) {
        run {
            // nested smart cast sink
            x.inc()

            // nested redefinition
            x = ...
        }
        // direct smart cast sink
        x.inc()
    }

    // direct redefinition
    x = ...
}
```

A mutable local property P defined at D is considered effectively immutable at a direct sink S, if there are no nested redefinitions on any CFG path between D and S.

A mutable local property P defined at D is considered effectively immutable at a nested sink S, if there are no nested redefinitions of P and all direct redefinitions of P precede S in the CFG.

Example:

```
fun directSinkOk() {
    var x: Int? = 42 // definition
    if (x != null)   // smart cast source
        x.inc()      // direct sink
    run {
        x = null     // nested redefinition
    }
}

fun directSinkBad() {
    var x: Int? = 42 // definition
    run {
        x = null     // nested redefinition
                     //   between a definition
                     //   and a sink
    }
```

```
    if (x != null)   // smart cast source
        x.inc()      // direct sink
}

fun nestedSinkOk() {
    var x: Int? = 42     // definition
    x = getNullableInt() // direct redefinition
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
}

fun nestedSinkBad01() {
    var x: Int? = 42     // definition
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
    x = getNullableInt() // direct redefinition
                         //   after the nested sink
}

fun nestedSinkBad02() {
    var x: Int? = 42     // definition
    run {
        x = null         // nested redefinition
    }
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
}
```

14.1.4 Loop handling

As mentioned before, a compiler may use *killDataFlow* instructions in loops to avoid slow data-flow analysis convergence. In the general case, a loop body may be evaluated zero or more times, which, combined with *killDataFlow* instructions, causes the smart cast sources from the loop body to *not* propagate to the containing scope. However, some loops, for which we can have static guarantees about how their body is evaluated, may be handled differently. For the following loop configurations, we consider their bodies to be definitely evaluated *one or more* times.

- `while (true) { ... }`
- `do { ... } while (condition)`

Note: in the current implementation, only the exact `while (true)` form is handled as described; e.g., `while (true == true)` does not work.

Note: one may extend the number of loop configurations, which are handled by smart casts, if the compiler implementation deems it necessary.

Example:

```
fun breakFromInfiniteLoop() {
    var a: Any? = null

    while (true) {
        if (a == null) return

        if (randomBoolean()) break
    }

    a // Smart cast to Any
}

fun doWhileAndSmartCasts() {
    var a: Any? = null

    do {
        if (a == null) return
    } while (randomBoolean())

    a // Smart cast to Any
}

fun doWhileAndSmartCasts2() {
    var a: Any? = null

    do {
        println(a)
    } while (a == null)

    a // Smart cast to Any
}
```

14.1.5 Bound smart casts

In some cases, it is possible to introduce smart casting *between properties* if it is known at compile-time that these properties are *bound* to each other. For instance, if a variable `a` is initialized as a copy of variable `b` and both are stable, they are guaranteed to reference the same runtime value and any assumption about `a` may be also applied to `b` and vice versa.

Example:

```
val a: Any? = ...
val b = a

if (b is Int) {
    // as a and b point to the same value,
    // a also is Int
    a.inc()
}
```

In more complex cases, however, it may not be trivial to deduce that two (or more) properties point to the same runtime object. This relation is known as *must-alias* relation between program references and it is implementation-defined in which cases a particular Kotlin compiler may safely assume this relation holds between two particular properties at a particular program point. However, it must guarantee that if two properties are considered bound, it is impossible for these properties to reference two different values at runtime.

One way of implementing bound smart casts would be to divide the space of stable program properties into disjoint *alias sets* of properties, and the analysis described above links the smart cast data flow information to sets of properties instead of single properties.

Such view could be further refined by considering special *alias sets* separately; e.g., an alias set of definitely non-null properties, which would allow the compiler to infer that `a?.b !== null` implies `a !== null` (for non-nullable `b`).

14.2 Local type inference

Local type inference in Kotlin is the process of deducing the compile-time types of expressions, lambda expression parameters and properties. As previously mentioned, type inference is a type constraint problem, and is usually solved by a type constraint solver.

In addition to the types of intermediate expressions, local type inference also performs deduction and substitution for generic type parameters of functions and types involved in every expression. You can use the Expressions part of this

specification as a reference point on how the types for different expressions are constructed.

> Important: additional effects of smart casts are considered in local type inference, if applicable.

Type inference in Kotlin is bidirectional; meaning the types of expressions may be derived not only from their arguments, but from their usage as well. Note that, albeit bidirectional, this process is still local, meaning it processes one statement at a time, strictly in the order of their appearance in a scope; e.g., the type of property in statement S_1 that goes before statement S_2 cannot be inferred based on how S_1 is used in S_2.

As solving a type constraint system is not a definite process (there may be more than one valid solution for a given constraint system), type inference may create several valid solutions. In particular, one may always derive a constraint $A <: T <: B$ for every free type variable T, where types A and B are both valid solutions.

> Note: this is valid even if T is a free type variable without any explicit constraints, as every type in Kotlin has an implicit constraint `kotlin.Nothing` $<: T <:$ `kotlin.Any`?.

In these cases an optimal constraint system solution is picked w.r.t. local type inference.

> Note: for the purposes of local type inference, an optimal solution is the one which does not contain any free type variables with no explicit constraints on them.

14.3 Function signature type inference

Function signature type inference is a variant of local type inference, which is performed for [function declarations], lambda literals and anonymous function declarations.

14.3.1 Named and anonymous function declarations

As described here, a named function declaration body may come in two forms: an expression body (a single expression) or a control structure body. For the latter case, an expected return type must be provided or is assumed to be `kotlin.Unit` and no special kind of type inference is needed. For the former case, an expected return type may be provided or can be inferred using local type inference from the expression body. If the expected return type is provided, it is used as an expected constraint on the result type of the expression body.

> Example:

```
fun <T> foo(): T { ... }
fun bar(): Int = foo() // an expected constraint T' <: Int
// allows the result of `foo` to be inferred automatically.
```

14.3.2 Statements with lambda literals

Complex statements involving one or more lambda literals introduce an additional level of complexity to type inference and overload resolution mechanisms. As mentioned in the overload resolution section, the overload resolution of callables involved in such statements is performed regardless of the contents of the lambda expressions and before any processing of their bodies is performed (including local type inference).

For a complex statement S involving (potentially overloaded) callables $C_1, \ldots, C_N$ and lambda literals $L_1, \ldots, L_M$, excluding the bodies of these literals, they are processed as follows.

1. An empty type constraint system Q is created;
2. The overload resolution, if possible, picks candidates for $C_1, \ldots, C_N$ according to the overload resolution rules;
3. For each lambda literal with unspecified number of parameters, we decide whether it has zero or one parameter based on the form of the callables and/or the expected type of the lambda literal. If there is no way to determine the number of parameters, it is assumed to be zero. If the number of parameters is determined to be one, the phantom parameter `it` is proceeded in further steps as if it was a named lambda parameter;

   > Important: the presence or absence of the phantom parameter `it` in the lambda body does not influence this process in any way.

4. For each lambda body $L_1, \ldots, L_N$, the expected constraints on the lambda arguments and/or lambda result type from the selected overload candidates (if any) are added to Q, and the overload resolution for all statements in these bodies is performed w.r.t. updated type constraint system Q. This may result in performing steps 1-3 in a recursive *top-down* fashion for nested lambda literals;

   > Important: in some cases overload resolution may fail to pick a candidate, e.g., because the expected constraints are incomplete, causing the constraint system to be unsound. If this happens, it is implementation-defined whether the compiler continues the top-down analysis or stops abruptly.

5. When the top-down analysis is done and the overload candidates are fixed, local type inference is performed on each lambda body and each statement

bottom-up, from the most inner lambda literals to the outermost ones, processing one lambda literal at a time, with the following additions.

- When inferring type of the return value (the last expression of a lambda body and/or the subjects for return expressions referring to this lambda literal), the additional constraints introduced on the result type of this lambda literal are added to Q;
- If inference with these constraints fails, but the result type is a subtype of `kotlin.Unit`, the inference is repeated without the additional constraints on the return value;
- The type of each lambda literal is considered to be the functional type $\mathrm{FT}(P_1, \ldots, P_S) \to R$, where $P_1, \ldots, P_S$ are the types of its parameters inferred from external constraints or specified in the lambda literal itself and R is the inferred type of its return value in the presence of external constraints.

The external constraints on lambda parameters, return value and body may come from the following sources:

- The (possibly overloaded) callable which uses the lambda literal as an argument;

  Note: as overload resolution is performed before any lambda literal inference takes place, this candidate is always known before external constraints are needed;

- The expected type of the declaration which uses the lambda literal as its body or initializer.

  Examples:

```
fun <T> foo(): T { ... }
fun <R> run(body: () -> R): R { ... }

fun bar() {
    val x = run {
        run {
            run {
                foo<Int>() // last expression inferred to be of type Int
            } // this lambda is inferred to be of type () -> Int
        } // this lambda is inferred to be of type () -> Int
    } // this lambda is inferred to be of type () -> Int
    // x is inferred to be of type Int

    val y: Double = run { // this lambda has an external constraint R' <: Double
        run { // this lambda has an external constraint R'' <: Double
            foo() // this call has an external constraint T' <: Double
                  // allowing to infer T to be Double in foo
        }
```

```
        }
    }
```

14.4 Bare type argument inference

Bare type argument inference is a special kind of type inference where, given a type T and a constructor TC, the type arguments $A_0, A_1 \dots A_N$ are inferred such that $TC[A_0, A_1 \dots A_N] <: S$ where $T <: S$. It is used together with *bare types* syntax sugar that can be employed in type checking and casting operators. The process is performed as follows.

First, let's consider the simple case of T being non-nullable, non-intersection type. Then, a simple type constraint system is constructed by introducing type variables for $A_0, A_1 \dots A_N$ and then solving the constraint $TC[A_0, A_1 \dots A_N] <: T$.

If T is an intersection type, the same process is performed for every member of the intersection type individually and then the resulting type argument values for each parameter A_K are merged using the following principle:

- If all values for a particular parameters are star-projections, the result is a star-projection;
- If some of the values are not star-projections and are strictly equal to each other, the result is one of their values;
- Else, the result is a star-projection.

If T is a nullable type $U?$, the steps given above are performed for its non-nullable counterpart type U.

Chapter 15

Builder-style type inference

Some functions or parameters of functions in the standard library are annotated with the special `@BuilderInference` annotation, making calls to these functions eligible for the special kind of type inference: **builder-style type inference**. In order to allow builder-style inference for a function parameter, this parameter must hold the following properties:

- It must be of an extension-function type;
- It must be marked with the `@BuilderInference` annotation.

In essense, the builder-style inference allows a receiver of an extension lambda parameter to be inferred from its usage in addition to standard type information sources. For a call to an eligible function with a lambda parameter L the inference is performed as described above, but the type parameters of the receiver parameter of the lambda expression are *postponed* till the body of the lambda expression is proceeded. After the inference of statements inside the lambda body, these parameters are inferred using an additional type inference step:

- For each call to a member function of the receiver or an extension function of the receiver marked with the `@BuilderInference` annotation, the type constraints are collected and gathered into a single constraint system;
- This system is solved in order to infer the actual type arguments of the receiver.

  Note: notable examples of builder-style inference-enabled functions are `kotlin.sequence` and `kotlin.iterator`. See standard library documentation for details.

Chapter 16

Runtime type information

The *runtime type information* (RTTI) is the information about Kotlin types of values available from these values at runtime. RTTI affects the semantics of certain expressions, changing their evaluation depending on the amount of RTTI available for particular values, implementation, and platform:

- The type checking operator
- The cast expression, especially the `as?` operator
- Class literals and the values they evaluate to

Runtime types are particular instances of RTTI for a particular value at runtime. These model a subset of the Kotlin type system. Namely, the runtime types are limited to classifier types, function types and a special case of `kotlin.Nothing?` which is the type of `null` reference and the only nullable runtime type. This includes the classifier types created by anonymous object literals. There is a slight distinction between a Kotlin type system type and its runtime counterpart:

- On some platforms, some particular types may have the same runtime type representation. This means that checking or casting values of these types works the same way as if they were the same type
- Generic types with the same classifier are not required to have different runtime representations. One cannot generally rely on them having the same representation outside of a particular platform. Platform specifications must clarify whether some or all types on these platforms have this feature.

RTTI is also the source of information for platform-specific *reflection* facilities in the standard library.

The types actual values may have are limited to class and object types and function types as well as `kotlin.Nothing?` for the `null` reference. `kotlin.Nothing` (not to be confused with its nullable variant `kotlin.Nothing?`) is special in the way that this type is never encountered as a runtime type even though it may

have a platform-specific representation. The reason for this is that this type is used to signify non-existent values.

16.1 Runtime-available types

Runtime-available types are the types that can be guaranteed (during compilation) to have a concrete *runtime* counterpart. These include all the runtime types, their nullable variants as well as `reified` type parameters, that are guaranteed to inline to a runtime type during type parameter substitution. Only runtime-available types may be passed (implicitly or explicitly) as substitutions to reified type parameters, used for type checks and safe casts. During these operations, the nullability of the type is checked using reference-equality to `null`, while the rest is performed by accessing the runtime type of a value and comparing it to the supplied runtime-available type.

For all generic types that are not expected to have RTTI for their generic arguments, only "raw" variants of generic types (denoted in code using the star-projected type notation or a special parameter-less notation) are runtime-available.

> Note: one may say that classifier generics are *partially* runtime available due to them having information about only the classifier part of the type

Exception types must be runtime-available to enable type checks that the `catch` clause of `try`-expression performs.

Only non-nullable runtime types may be used in `class` literal expressions. These include reified type parameters with non-nullable upper bounds, as well as all classifier and function types.

16.2 Reflection

Particular platforms may provide more complex facilities for runtime type introspection through the means of *reflection* — special platform-provided part of the standard library that allows to access more detailed information about types and declarations at runtime. It is, however, platform-specific and one must refer to particular platform documentation for details.

Chapter 17

Exceptions

An *exception* type declaration is any type declaration that meets the following criteria:

- It is a class or object declaration;
- It has `kotlin.Throwable` as one of its supertype (either explicitly or implicitly);
- It has no type parameters.

Any object of an exception type may be *thrown* or *caught*.

17.1 Catching exceptions

A `try`-expression becomes *active* once the execution of the program enters it and stops being active once the execution of the program leaves it. If there are several active `try`-expressions, the one that became active last is *currently active*.

If an exception is thrown while a try-expression is currently active and this try-expression has any `catch`-blocks, those `catch`-blocks are checked for applicability for this exception. A `catch`-block is applicable for an exception object if the runtime type of this expression object is a subtype of the bound exception parameter of the `catch`-block.

> Note: the applicability check is subject to Kotlin runtime type information limitations and may be dependent on the platform implementation of runtime type information, as well as the implementation of exception classes.

If a `catch`-block is applicable for the exception thrown, the code inside the block is evaluated and the value of the block is returned as the value of a

`try`-expression. If the `try`-expression contains a `finally`-block, the body of this block is evaluated after the body of the selected `catch` block. If these evaluations results in throwing other exceptions (including the one caught by the `catch`-block), they are propagated as if none of the `catch`-blocks were applicable.

> Important: the `try`-expression itself is not considered active inside its own `catch` and `finally` blocks.

If none of the `catch`-blocks of the currently active `try`-expression are applicable for the exception, the `finally` block (if any) is still evaluated, and the exception is propagated, meaning the next active `try`-expression becomes currently active and is checked for applicability.

If there are no active `try`-blocks, the execution of the program finishes, signaling that the exception has reached top level.

17.2 Throwing exceptions

Throwing an exception object is done using `throw`-expression. A valid throw expression `throw e` requires that:

- `e` is a value of a runtime-available type;
- `e` is a value of an exception type.

Throwing an exception results in checking active `try`-blocks.

> Note: Kotlin does not specify whether throwing exceptions involves construction of a program stack trace and how the actual exception handling is implemented. This is a platform-dependent mechanism.

Chapter 18

Annotations

Annotations are a form of syntactically-defined metadata which may be associated with different entities in a Kotlin program. Annotations are specified in the source code of the program and may be accessed on a particular platform using platform-specific mechanisms both by the compiler (and source-processing tools) and at runtime (using reflection facilities). Values of annotation types cannot be created directly, but can be operated on when accessed using platform-specific facilities.

18.1 Annotation values

An annotation value is a value of a special annotation type. An annotation type is a special kind of class type which is allowed to include read-only properties of the following types:

- Integer types;
- Enum types;
- String type;
- Other annotation types;
- Arrays of any type listed above.

Annotation classes are not allowed to have any member functions, constructors or mutable properties. They are also not allowed to have declared supertypes and are considered to be implicitly derived from `kotlin.Annotation`.

18.2 Annotation retention

The retention level of an annotation declares which compilation artifacts (for

a particular compiler on a particular platform) *retain* this kind of annotation. There are the following types of retention available:

- Source retention (accessible by source-processing tools);
- Binary retention (retained in compilation artifacts);
- Runtime retention (accessible at runtime).

  Each subsequent level inherits what is accessible on the previous levels.

For availability and particular ways of accessing the metadata specified by these annotations please refer to the corresponding platform-specific documentation.

18.3 Annotation targets

The *target* of a particular type of annotations is the kind of program entity which this annotations may be placed on. There are the following targets available:

- A class declaration (including annotation classes);
- An annotation class declaration;
- A type parameter;
- A property declaration;
- A property backing field;
- A property getter;
- A property setter;
- A local property declaration;
- A value parameter in function or constructor declaration;
- A constructor;
- A function declaration;
- A type usage;
- An arbitrary expression;
- A Kotlin file;
- A type alias declaration.

18.4 Annotation declarations

Annotations are declared using annotation class declarations. See the corresponding section for details.

Annotations may be declared *repeatable* (meaning that the same annotation may be applied to the same entity more than once) or *non-repeatable* (meaning that only one annotation of a particular type may be applied to the same entity).

18.5 Built-in annotations

- `kotlin.annotation.Retention`

  `kotlin.annotation.Retention` is an annotation which is only used on annotation classes to specify their annotation retention level. It has the following single field:

  - `val value: AnnotationRetention = AnnotationRetention.RUNTIME`

    The retention level of the annotated annotation.

  `kotlin.annotation.AnnotationRetention` is an enum class with the following values (see Annotation retention section for details):

  - `SOURCE`;
  - `BINARY`;
  - `RUNTIME`.

- `kotlin.annotation.Target`

  `kotlin.annotation.Target` is an annotation which is only used on annotation classes to specify targets those annotations are valid for. It has the following single field:

  - `vararg val allowedTargets: AnnotationTarget`

    The allowed annotation targets of the annotated annotation.

  `kotlin.annotation.AnnotationTarget` is an enum class with the following values (see Annotation targets section for details):

  - `CLASS`;
  - `ANNOTATION_CLASS`;
  - `TYPE_PARAMETER`;
  - `PROPERTY`;
  - `FIELD`;
  - `LOCAL_VARIABLE`;
  - `VALUE_PARAMETER`;
  - `CONSTRUCTOR`;
  - `FUNCTION`;
  - `PROPERTY_GETTER`;
  - `PROPERTY_SETTER`;
  - `TYPE`;
  - `EXPRESSION`;
  - `FILE`;
  - `TYPEALIAS`.

- `kotlin.annotation.Repeatable`

`kotlin.annotation.Repeatable` is an annotation which is only used on annotation classes to specify whether this particular annotation is repeatable. Annotations are non-repeatable by default.

- `kotlin.RequiresOptIn` / `kotlin.OptIn`

  `kotlin.RequiresOptIn` is an annotation class with two fields:

  - `val message: String = ""`

    The message describing the particular opt-in requirements.

  - `val level: Level = Level.ERROR`

    The severity level of the experimental status with two possible values: `Level.WARNING` and `Level.ERROR`.

  This annotation is used to introduce implementation-defined experimental language or standard library features.

  `kotlin.OptIn` is an annotation class with a single field:

  - `vararg val markerClass: KClass<out Annotation>`

    The classes which this annotation allows to use.

  This annotation is used to explicitly mark declarations which use experimental features marked by `kotlin.RequiresOptIn`.

  It is implementation-defined how this annotation is processed.

  > Note: before Kotlin 1.4.0, there were two other built-in annotations: `@Experimental` (now replaced by `@RequiresOptIn`) and `@UseExperimental` (now replaced by `@OptIn`) serving the same purpose which are now deprecated.

- `kotlin.Deprecated` / `kotlin.ReplaceWith`

  `kotlin.Deprecated` is an annotation class with the following fields:

  - `val message: String`

    A message supporting the deprecation.

  - `val replaceWith: ReplaceWith = ReplaceWith("")`

    An optional replacement for the deprecated code.

  - `val level: DeprecationLevel = DeprecationLevel.WARNING`

    The deprecation level with three possible values: `DeprecationLevel.WARNING`, `DeprecationLevel.ERROR` and `DeprecationLevel.HIDDEN`.

  `kotlin.ReplaceWith` is itself an annotation class containing the information on how to perform the replacement in case it is provided. It has the following fields:

  - `val expression: String`

    The replacement code.

  - `vararg val imports: String`

    An array of imports needed for the replacement code to work correctly.

  `kotlin.Deprecated` is a built-in annotation supporting the deprecation cycle for declarations: marking some declarations as outdated, soon to be replaced with other declarations, or not recommended for use. It is implementation-defined how this annotation is processed, with the following recommendations:

  - Attempting to use a declaration with deprecation level of `DeprecationLevel.WARNING` should produce a compile-time warning;
  - Attempting to use a declaration with deprecation level of `DeprecationLevel.ERROR` should produce a compile-time error.

- `kotlin.Suppress`

  `kotlin.Suppress` is an annotation class with the following single field:

  - `vararg val names: String`

    The names of features this annotation is suppressing.

  `kotlin.Suppress` is used to optionally mark any piece of code as suppressing some language feature, such as a compiler warning, an IDE mechanism or a language feature. The names of features which one can suppress with this annotation are implementation-defined, as is the processing of this annotation itself.

- `kotlin.SinceKotlin`

  `kotlin.SinceKotlin` is an annotation class with the following single field:

  - `val version: String`

    The version of Kotlin language.

  `kotlin.SinceKotlin` is used to mark a declaration which is only available since a particular version of the language. These mostly refer to standard library declarations. It is implementation-defined how this annotation is processed.

- `kotlin.UnsafeVariance`

  `kotlin.UnsafeVariance` is an annotation class with no fields which is only applicable to types. Any type instance marked by this annotation explicitly states that the variance errors arising for this particular type instance are to be ignored by the compiler.

- `kotlin.DslMarker`

  `kotlin.DslMarker` is an annotation class with no fields which is applicable only to other annotation classes. An annotation class annotated with `kotlin.DslMarker` is marked as a marker of a specific DSL (domain-specific language). Any type annotated with such a marker is said to belong to that specific DSL. This affects overload resolution in the following way: no two implicit receivers with types belonging to the same DSL are available in the same scope. See Overload resolution section for details.

- `kotlin.PublishedApi`

  `kotlin.PublishedApi` is an annotation class with no fields which is applicable to any declaration. It may be applied to any declaration with `internal` visibility to make it available to `public inline` declarations. See Declaration visibility section for details.

- `kotlin.BuilderInference`

  Marks the annotated function of function argument as eligible for builder-style type inference. See corresponding section for details.

  > Note: as of Kotlin 1.4.0, this annotation is experimental and, in order to use it in one's code, one must explicitly enable it using opt-in annotations given above. The particular marker class used to perform this is implementation-defined.

- `kotlin.RestrictSuspension`

  In some cases we may want to limit which suspending functions can be called in another suspending function with an extension receiver of a specific type; i.e., if we want to provide a coroutine-enabled DSL, but disallow the use of arbitrary suspending functions. To do so, the type `T` of that extension receiver needs to be annotated with `kotlin.RestrictSuspension`, which enables the following limitations.

  – Suspending functions with an extention receiver of type `T` are restricted from calling other suspending functions besides those accessible on this receiver.
  – Suspending functions of type `T` can be called only on an extention receiver.

Chapter 19

Asynchronous programming with coroutines

19.1 Suspending functions

Most functions in Kotlin may be marked *suspending* using the special `suspend` modifier. There are almost no additional restrictions: regular functions, extension functions, top-level functions, local functions, lambda literals, all these may be suspending functions.

> Note: the following functions and function values cannot be marked as suspending.
>
> - anonymous function declarations;
> - constructors;
> - property getter/setters;
> - delegation-related operator functions.
>
> Note: platform-specific implementations may extend the restrictions on which kinds of functions may be suspending.

Suspending functions have a suspending function type, also marked using the `suspend` modifier.

A suspending function is different from non-suspending functions by potentially having zero or more *suspension points* — statements in its body which may pause the function execution to be resumed at a later moment in time. The main source of suspension points are calls to other suspending functions which represent possible suspension points.

> Note: suspension points are important because at these points another function may start in the same flow of execution, leading to potential

changes in the shared state.

Non-suspending functions may not call suspending functions directly, as they do not support suspension points. Suspending functions may call non-suspending functions without any limitations; such calls do not create suspension points. This restriction is also known as "function colouring".

> Important: an exception to this rule are non-suspending inlined lambda parameters: if the higher-order function invoking such a lambda is called from a suspending function, this lambda is allowed to also have suspension points and call other suspending functions.
>
> Note: suspending functions interleaving each other in this manner are not dissimilar to how functions from different threads interact on platforms with multi-threading support. There are, however, several key differences. First, suspending functions may pause only at suspension points, i.e., they cannot be paused at an arbitrary execution point. Second, this interleaving may happen on a single platform thread.
>
> In a multi-threaded environment suspending functions may also be interleaved by the platform-dependent concurrent execution, independent of the interleaving of coroutines.

The implementation of suspending functions is platform-dependent. Please refer to the platform documentation for details.

19.2 Coroutines

A *coroutine* is a concept similar to a thread in traditional concurrent programming, but based on *cooperative multitasking*, e.g., the switching between different execution contexts is done by the coroutines themselves rather than the operating system or a virtual machine.

In Kotlin, coroutines are used to implement suspending functions and can switch contexts only at suspension points.

A call to a suspending function creates and starts a coroutine. As one can call a suspending function only from another suspending function, we need a way to bootstrap this process from a non-suspending context.

> Note: this is required as most platforms are unaware of coroutines or suspending functions, and do not provide a suspending entry point. However, a Kotlin compiler may elect to provide a suspending entry point on a specific platform.

One of the ways of starting suspending function from a non-suspending context is via a *coroutine builder*: a non-suspending function which takes a suspending

function type argument (e.g., a suspending lambda literal) and handles the coroutine lifecycle.

The implementation of coroutines is platform-dependent. Please refer to the platform documentation for details.

19.3 Implementation details

Despite being platform-dependent, there are several aspects of coroutine implementation in Kotlin, which are common across all platforms and belong to the Kotlin/Core. We describe these details below.

19.3.1 `kotlin.coroutines.Continuation<T>`

Interface `kotlin.coroutines.Continuation<T>` is the main supertype of all coroutines and represents the basis upon which the coroutine machinery is implemented.

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resumeWith(result: Result<T>)
}
```

Every suspending function is associated with a generated `Continuation` subtype, which handles the suspension implementation; the function itself is adapted to accept an additional continuation parameter to support the Continuation Passing Style. The return type of the suspending function becomes the type parameter `T` of the continuation.

`CoroutineContext` represents the context of the continuation and is an indexed set from `CoroutineContext.Key` to `CoroutineContext.Element` (e.g., a special kind of map). It is used to store coroutine-local information, and takes important part in [Coroutine interception].

`resumeWith` function is used to propagate the results in between suspension points: it is called with the result (or exception) of the last suspension point and resumes the coroutine execution.

To avoid the need to explicitly create the `Result<T>` when calling `resumeWith`, the coroutine implementation provides the following extension functions.

```
fun <T> Continuation<T>.resume(value: T)
fun <T> Continuation<T>.resumeWithException(exception: Throwable)
```

19.3.2 Continuation Passing Style

Each suspendable function goes through a transformation from normally invoked function to continuation passing style (CPS). For a suspendable function with parameters $p_1, p_2, \ldots, p_N$ and result type T a new function is generated, with an additional parameter p_{N+1} of type `kotlin.coroutines.Continuation<T>` and return type changed to `kotlin.Any?`. The calling convention for such function is different from regular functions as a suspendable function may either *suspend* or *return*.

- If the function returns a result, it is returned directly from the function as normal;
- If the function suspends, it returns a special marker value `COROUTINE_SUSPENDED` to signal its suspended state.

The calling convention is maintained by the compiler during the CPS transformation, which prevents the user from manually returning `COROUTINE_SUSPENDED`. If the user wants to suspend a coroutine, they need to perform the following steps.

- Access the coroutine's continuation object by calling `suspendCoroutineUninterceptedOrReturn` intrinsic or any of its wrappers;
- Store the continuation object to resume it later;
- Pass the `COROUTINE_SUSPENDED` marker to the intrinsic, which is then returned from the function.

As Kotlin does not currently support denotable union types, the return type is changed to `kotlin.Any?`, so it can hold both the original return type T and `COROUTINE_SUSPENDED`.

19.3.3 Coroutine state machine

Kotlin implements suspendable functions as state machines, since such implementation does not require specific runtime support. This dictates the explicit `suspend` marking (function colouring) of Kotlin coroutines: the compiler has to know which function can potentially suspend, to turn it into a state machine.

Each suspendable lambda is compiled to a continuation class, with fields representing its local variables, and an integer field for current state in the state machine. Suspension point is where such lambda can suspend: either a suspending function call or `suspendCoroutineUninterceptedOrReturn` intrinsic call. For a lambda with N suspension points and M return statements, which are not suspension points themselves, $N + M$ states are generated (one for each suspension point plus one for each non-suspending return statement).

Example:

```
// Lambda body with multiple suspension points
val a = a()
val y = foo(a).await() // suspension point #1
b()
val z = bar(a, y).await() // suspension point #2
c(z)

// State machine code for the lambda after CPS transformation
//     (written in pseudo-Kotlin with gotos)
class <anonymous> private constructor(
    completion: Continuation<Any?>
): SuspendLambda<...>(completion) {
    // The current state of the state machine
    var label = 0

    // local variables of the coroutine
    var a: A? = null
    var y: Y? = null

    fun invokeSuspend(result: Any?): Any? {
        // state jump table
        if (label == 0) goto L0
        if (label == 1) goto L1
        if (label == 2) goto L2
        else throw IllegalStateException()

      L0:
        // result is expected to be `null` at this invocation

        a = a()
        label = 1
        // 'this' is passed as a continuation
        result = foo(a).await(this)
        // return if await had suspended execution
        if (result == COROUTINE_SUSPENDED)
            return COROUTINE_SUSPENDED
      L1:
        // error handling
        result.throwOnFailure()
        // external code has resumed this coroutine
        // passing the result of .await()
        y = (Y) result
        b()
        label = 2
        // 'this' is passed as a continuation
        result = bar(a, y).await(this)
```

```
            // return if await had suspended execution
            if (result == COROUTINE_SUSPENDED)
                return COROUTINE_SUSPENDED
          L2:
            // error handling
            result.throwOnFailure()
            // external code has resumed this coroutine
            // passing the result of .await()
            Z z = (Z) result
            c(z)
            label = -1 // No more steps are allowed
            return Unit
        }

        fun create(completion: Continuation<Any?>): Continuation<Any?> {
          <anonymous>(completion)
        }

        fun invoke(completion: Continuation<Any?>): Any? {
            create(completion).invokeSuspend(Unit)
        }
    }
```

19.3.4 Continuation interception

Asynchronous computations in many cases need to control how they are executed, with varying degrees of precision. For example, in typical user interface (UI) applications, updates to the interface should be executed on a special UI thread; in server-side applications, long-running computations are often offloaded to a separate thread pool, etc.

Continuation interceptors allow us to intercept the coroutine execution between suspension points and perform some operations on it, usually wrapping the coroutine continuation in another continuation. This is done using the `kotlin.coroutines.ContinuationInterceptor` interface.

```
interface ContinuationInterceptor : CoroutineContext.Element {
    companion object Key : CoroutineContext.Key<ContinuationInterceptor>
    fun <T> interceptContinuation(continuation: Continuation<T>): Continuation<T>
    fun releaseInterceptedContinuation(continuation: Continuation<*>)
}
```

As seen from the declaration, `ContinuationInterceptor` is a `CoroutineContext.Element`, and to perform the continuation interception, an instance of `ContinuationInterceptor` should be available in the coroutine context, where it is used similarly to the following line of code.

```
val intercepted = continuation.context[ContinuationInterceptor]?.interceptContinuation(continuati
```

When the cached `intercepted` continuation is no longer needed, it is released using `ContinuationInterceptor.releaseInterceptedContinuation(...)`.

> Note: this machinery is performed "behind-the-scenes" by the coroutine framework implementation.

19.3.5 Coroutine intrinsics

Accessing the low-level continuations is performed using a limited number of built-in intrinsic functions, which form the complete coroutine API. The rest of asynchronous programming support is provided as a Kotlin library `kotlinx.coroutines`.

The complete built-in API for working with coroutines is shown below (all of these are declared in package `kotlin.coroutines.intrinsics` of the standard library).

```
fun <T> (suspend () -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit>

suspend fun <T>
    suspendCoroutineUninterceptedOrReturn(
        block: (Continuation<T>) -> Any?): T

fun <T> (suspend () -> T).
    startCoroutineUninterceptedOrReturn(
        completion: Continuation<T>): Any?

fun <T> Continuation<T>.intercepted(): Continuation<T>

// Additional functions for types with explicit receiver

fun <R, T> (suspend R.() -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit>

fun <T> (suspend R.() -> T).
    startCoroutineUninterceptedOrReturn(
        completion: Continuation<T>): Any?
```

Function `createCoroutineUnintercepted` is used to create a coroutine corresponding to its extension receiver suspending function, which invokes the passed `completion` continuation upon completion. This function does not start the coroutine, however; to do that, one have to call

`Continuation<T>.resumeWith` function on the created continuation object. Suspending function `suspendCoroutineUninterceptedOrReturn` provides access to the current continuation (similarly to how `call/cc` works in Scheme). If its lambda returns the `COROUTINE_SUSPENDED` marker, it also suspends the coroutine. Together with `Continuation<T>.resumeWith` function, which resumes or starts a coroutine, these functions form a complete coroutine API built into the Kotlin compiler.

Chapter 20

Concurrency

Kotlin Core does not specify the semantics of the code running in concurrent environments. For information on threading APIs, memory model, supported synchronization and other concurrency-related capabilities of Kotlin on your platform, please refer to platform documentation.