Kotlin Language Specification

Table of Contents

1. Purpose and Format of this Specification

1.1. General

Kotlin is a general-purpose, statically- and strongly-typed programming language. Kotlin supports imperative, object-oriented and functional programming styles. Its object-oriented system is based on single class inheritance and multiple interface inheritance. Only classes can contain data and initialization code, and interfaces can contain abstract functional members and, optionally, default implementations for them.

The Kotlin type system is based on nominal (rather than structural) subtyping with subtyping polymorphism (virtual functional members), with support of generics — a form of parametric polymorphism featuring constrained type-parameters, generic variance and type projections (a simple restricted form of existential types somewhat similar to Java wildcards).

Being a statically-typed means that every expression has a known compile-time type and that the language guarantees that at runtime the expression will evaluate to an instance of that type (except certain cases caused by generic type erasure and indicated by compiler warnings).

Being a strongly-typed means that any conversion between types that involves an evaluation, changes the value representation or may fail at runtime requires an explicit invocation of a conversion function or an explicit downcast — it never can implicitly occur in a simple assignment. Implicit conversions are reserved for safe upcasts enabled by subtyping and variance that do not change the state or identity of the object being converted, but simply allows to see through a reference of a different (but compatible) type. No user-defined implicit conversions are supported either.

Kotlin has a unified type system — it has a single root type `kotlin.Any?` and every other type is a subtype of this root type.

Types in Kotlin are not a purely compile-time concept. Some information about types is preserved at runtime, and any value can be dynamically queried for its exact type, and dynamically checked for compatibility with any given type. But runtime type information is not precise — it is not possible to distinguish between different instantiations of the same generic type (arrays being an exception), and between arrays of corresponding nullable and non-nullable types.

Although Kotlin provides a number of predefined types, some of them having a special meaning in the language, it is possible to declare used-defined types in the form of classes and interfaces.

Kotlin supports functional programming paradigm and so functions are considered to be first-class values — they have type, they can be passed as arguments, returned as return values, and stored in local variables, fields of classes and elements of arrays.

TODO: dynamic types, non-nullable types.

Kotlin syntax is intended to be concise, but without impeding readability. It supports traditional

nested block structure based on curly braces, but also features implicit semicolon inference, relieving a user of typing trailing semicolons in most cases. Whitespace (except new lines) is generally insignificant (except within string and character literals), but in some cases it is required to separate certain tokens.

TODO: Describe main purposes, features and supported paradigms of Kotlin, general design principles.

1.2. Normative and Informative Text

TODO

1.3. Notations

All numbers in this specification are given in decimal notation unless explicitly specified otherwise. All characters that can be interpreted both as characters from ASCII range and as similarly looking characters from a different part of Unicode character set shall be interpreted as ASCII characters. Logical connectives such as "or", "and", "not", "if ... then ..." have their usual meaning as in the two-valued classical logic. In particular, "or" is not exclusive, and "if A, then B" is exactly equivalent to "B or not A". If an exclusive "or" is intended, it is usually written in form "exactly one of the following is true: ...", or as "A or B, but not both". A phrase "A is true" is an exact equivalent of just "A", and "A is false" is an exact equivalent of "not A".

"An error" means "a compile-time error" unless specified otherwise.

When the specification states that some condition "cannot occur", it means that it is a compile-time error if such condition occurs. When the specification states that some condition "is not significant", it means that the observable behavior of the program does not change when this condition changes.

When the specification refers to the "first" entry in some list originating in some syntactic representation in a source (e.g. "the first type-parameter of a function" or "the first statement in a block") it refers to the lexically leftmost entry (i.e. starting with the smallest offset in the source file). Similarly, the "second" means the leftmost to the right of the first, the "n-th" means the leftmost to the right of the (n-1)-th, and the "last" means the rightmost.

TODO: metavariables, font used for Kotlin code fragments.

1.4. Referenced Specifications

1.4.1. General

TODO

1.4.2. Unicode

TODO: Specify what version of the Unicode Standard is supported, and what is the upgrade and compatibility policy is in effect as newer versions of the Standard are released.

1.4.3. IEEE 754

TODO

1.4.4. Java Virtual Machine

TODO

1.4.5. ECMAScript

TODO

1.5. Compatibility Policy

TODO: Specify what the policy for the language evolution with respect to backwards compatibility is, and what the policy for compiler bug fixes is.

1.6. Errata Policy

This specification is intended to be free of contradictions and ambiguous language. But experience shows that for documents of such complexity it is not uncommon that some defects go unnoticed for some time. This sections outlines general principles that shall be used by implementers of a compiler or other language tool in case a contradiction or ambiguous language is discovered. TODO

1.7. Glossary

This section gives definitions of some terms used in this specification.

- angle brackets – Unicode characters LESS-THAN SIGN (U+003C) '<' and GREATER-THAN SIGN (U+003E) '>' in those contexts where they used as delimiters rather than operators.
- curly braces – Unicode characters LEFT CURLY BRACKET (U+007B) '{' and RIGHT CURLY BRACKET' (U+007D) '}'.
- textual order – top-to-bottom, left-to-right within a line (actually, order of chars in the source file, unrelated to RTL languages)
- ASCII - TODO
- dot - '.' FULL STOP
- underscore - Unicode character '_' LOW LINE (U+005F).
- enclosed in parentheses/brackets/etc - TODO TODO

2. Language Overview

TODO: Non-normative language overview, with references to detailed normative specifications of all features.

[Informative text

A "Hello world" program is often used to introduce a programming language. In Kotlin it may looks as follows:

```
fun main(args: Array<String>) {
    println("Hello world")
}
```

End of informative text]

3. Compilation Process

3.1. General

TODO

3.2. Compiler Input and Output

TODO: Describe what files and streams can be input and output of the compiler.

3.3. Command Line

TODO: Describe encoding, parsing rules, syntax of the command line, and meaning of supported options.

3.4. Referenced Libraries

3.4.1. General

TODO: Describe how libraries can be referenced from a compilation, how possible conflicts are handled, how ill-formed libraries are handled, and what is an effect (if any) of an order in which libraries are referenced.

3.4.2. Standard Library

TODO: Describe what the standard library is, whether it has to be explicitly referenced, and what happens if it is ill-formed or does not contain some symbols that are assumed to exist in this specification.

3.5. Source Files

3.5.1. General

TODO

3.5.2. Supported Encodings

TODO An implementation may choose to represent all source files in UTF-16 and support only non-surrogate Unicode characters up to U+FFFF. Thus, characters that require a surrogate pair to be represented in UTF-16 would not be allowed on such implementations. For maximum portability, it is recommended that Kotlin programs do not use Unicode characters above U+FFFF.

3.5.3. Ordering of Source Files

TODO: Describe what is an effect (if any) of an order in which source files are provided to the compiler.

3.6. Parsing

TODO: Give an overview of a parsing process, with a reference to detailed specification in §?.

3.7. Target Platforms

TODO

3.7.1. Java Virtual Machine

Platform Limitations

TODO: No dynamic type

3.7.2. ECMAScript

Platform Limitations

TODO: Different behavior of numeric types

3.8. Errors, Warnings and Messages

TODO A compiler warning can be suppressed by applying the `kotlin.Suppress` annotation to the compilation unit, declaration or expression that contains the warning.

3.8.1. Use of Standard Output and Standard Error Streams

TODO

3.8.2. Early Compilation Termination

Affected Files

TODO

3.9. Implementation Limitations

TODO

4. Grammars and Parsing

4.1. General

A production from lexical grammar contains double colon after its name, a production from syntax grammar contains a single colon after its name. Each alternative is given on a separate line. If all alternatives are short (refer to only one production or terminal), they can be given on a single line after the words "one of". Subscript opt denotes that an element is optional. Terminals are given in bold monospace font.

TODO: Grammar notations used in this specification, and general parsing rules (longest match rule, etc.), trailing U+001A ?

4.2. Lexical Grammar

input::

    shebang$_{opt}$_input-elements$_{opt}$ (TODO: implement standalone shebang)

shebang::

    #! *input-characters$_{opt}$*

input-characters::

    input-characters$_{opt}$ input-character

input-character::

    Any Unicode character point except *new-line-character*

new-line-character::

    line-feed
    carriage-return

new-line::

    line-feed
    carriage-return line-feed$_{opt}$

line-feed::

    LINE FEED (U+000A)

carriage-return::

    CARRIAGE RETURN (U+000D)

input-element::

whitespace
comment
token

whitespace::

    new-line
    SPACE (U+0020)
    CHARACTER TABULATION (U+0009)
    FORM FEED (U+000C)

comment::

    end-of-line-comment
    delimited-comment

end-of-line-comment::

    // input-characters$_{opt}$*

delimited-comment::

    */** delimited-comment-parts$_{opt}$ asterisks */*

delimited-comment-parts::

    delimited-comment-parts$_{opt}$ delimited-comment-part

delimited-comment-part::

    delimited-comment
    not-asterisk
    asterisks not-slash-or-asterisk

asterisks::

    *asterisks$_{opt}$ ****

not-asterisk::

    Any Unicode character ****

not-slash-or-asterisk::

    Any Unicode character except *** and */*

token::

    identifier
    field-identifier
    keyword
    integer-literal
    real-literal
    char-literal
    string-literal
    operator

identifier::

> *regular-identifier*
> *escaped-identifier*

field-identifier::

> **$* <nospace> identifier* (TODO: consider moving to syntax grammar)

regular-identifier::

> *keyword-or-identifier* other than a *keyword*

keyword-or-identifier::

> *identifier-start identifier-parts$_{opt}$*

identifier-start::

> *letter*

letter::

> Any Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

identifier-parts::

> *identifier-parts$_{opt}$ identifier-part*

identifier-part::

> *identifier-start digit*

digit::

> Any Unicode character of class Nd

escaped-identifier::

> *backtick escape-identifier-characters backtick* (TODO: Unicode escapes)

backtick::

> GRAVE ACCENT (U+0060)

escape-identifier-characters::

> *escape-identifier-characters$_{opt}$ escape-identifier-character*

escape-identifier-character

> Any *input-character* except *backtick*

keyword::

> one of *as break class continue do else false for fun if in interface is null*
> *object package return super this throw true try typealias val var when while*
> any *keyword-or-identifier* consisting of one or more characters _

decimal-digit::

> one of *0 1 2 3 4 5 6 7 8 9*

integer-literal::

decimal-digits integer-literal-suffix_{opt}

integer-literal-suffix::

    L

decimal-digits::

    decimal-digits_{opt} decimal-digit

float-literal::

    TODO

hex-digit::

    one of **0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F**

hex-digits::

    hex-digits_{opt} hex-digit

hex-literal::

    0x hex-digits
    0X hex-digits

char-literal::

    TODO

4.2.1. Identifiers

An identifier can start with a letter or underscore, followed by zero or more letters, underscores or decimal digits. Any token of this form is an identifier, unless it is explicitly reserved as a keyword. Identifiers are case-sensitive: `kotlin`, `Kotlin` and `KOTLIN` are 3 different identifiers. A single underscore and sequences of 2, 3, etc. underscores are reserved keywords, but otherwise underscore is valid everywhere in an identifier. *[Note:* A digit is defined as a member of Unicode class Nd, that class contains many characters beyond ten regular ASCII digits. *End Note] [Note:* Unicode escape sequences are not supported in regular identifiers. *End Note]*

4.2.2. Keywords

If a use of an identifier with a spelling matching a keyword is desired, an escaped identifier may be used.

4.2.3. Decimal Literals

A decimal literal is a sequence of one or more ASCII decimal digits, followed by an optional `L` (LATIN CAPITAL LETTER L, U+004C). A literal without a suffix represents a constant value of type `Int` (TODO: it is more complicated). A literal with suffix `L` represents a constant value of type Long. A decimal literal represents a constant value that results from interpreting the literal in decimal notation. The value of a literal must lie within the range of its type. [Note: Decimal literals always represent non-negative integers. For example, an expression `-5` is the unary operator minus applied to an operand that is a decimal literal `5`, rather than a single decimal literal. End Note]

It is an error if a decimal literal other than `0` starts with digit `0`. *[Example:* Literals `00`, `007` are errors. *End Example]* It is an error if a decimal literal has an adjacent identifier or keyword before or after it without any characters separating these tokens. *[Example:* `0less1` is an error. *End Example]*

4.2.4. Hexadecimal Literals

A hexadecimal literal consists of a prefix `0x` or `0X`, followed a sequence of one or more hexadecimal digits, followed by an optional `L` (LATIN CAPITAL LETTER L, U+004C). A literal without a suffix represents a constant value of type `Int`. A literal with suffix `L` represents a constant value of type `Long`. A hexadecimal digit is an ASCII decimal digit or an upper-case or lower-case letter in the range `A-F` (corresponding upper-case and lower-case letters are equivalent, having numeric values 10-15, respectively). Prefixes `0x` and `0X` are equivalent. A hexadecimal literal represents a constant value that results from interpreting the literal in hexadecimal notation (with leading zeros allowed). The value of a literal must lie within the range of its type.

TODO: overflow behavior

4.3. Syntax Grammar

expression-or-block:

    expression
    assignment
    block

expression:

    disjunction

assignment:

    assignable-expression assignment-operator disjunction

assignment-operator:

    =
    +=
    -=
    *=
    /=
    %=

disjunction:

    conjunction
    disjunction *||* conjunction

conjunction:

    equality
    conjunction *&&* equality

equality:

    comparison

equality equality-operator comparison

equality-operator:

  ==

  !=

  ===

  !==

comparison:

  infix-operation

  comparison comparison-operator infix-operation

comparison-operator:

  >

  <

  >=

   ⇐

infix-operation:

  elvis-expression

  elvis-expression is-operator type

  infix-operation in-operator elvis-expression

is-operator:

  is

  ! <nospace> is

in-operator:

  in

  ! <nospace> in

elvis-expression:

  infix-function-call

  elvis-expression ?: infix-function-call

infix-function-call:

  range-expression

  infix-function-call simple-name range-expression

range-expression:

  additive-expression

  range-expression .. additive-expression

additive-expression:

  multiplicative-expression

  additive-expression additive-operator multiplicative-expression

additive-operator:

    +

    -

multiplicative-expression:

    as-expression

    multiplicative-expression multiplicative-operator as-expression

multiplicative-operator:

- + /
   %

as-expression:

    prefix-unary-expression

    as-expression as-operator type

as-operator:

    as

    as <nospace> ?

prefix-unary-expression:

    postfix-unary-expression

    prefix-unary-operator prefix-unary-expression

    annotation prefix-unary-expression

    label prefix-unary-expression

prefix-unary-operator:

    +

- + ++ + — + !

annotation:

    TODO

label:

    TODO

postfix-unary-expression: assignable-expression
invocation-expression
postfix-unary-expression postfix-unary-operator

postfix-unary-operator:

    ++ + — + !!

assignable-expression:

    primary-expression

    indexing-expression

    member-access

primary-expression:

    parenthesized-expression
    literal
    function-literal
    this-expression
    super-expression
    object-literal
    simple-name
    field-name
    callable-reference
    package-expression
    jump-expression
    conditional-expression
    loop-expression
    try-expression

callable-reference:

    simple-name
    TODO (qualified reference)

parenthesized-expression:

    (*expression*)

literal:

    boolean-literal
    integer-literal
    float-literal
    character-literal
    string-literal
    null-literal

boolean-literal:

    true
    false

null-literal:

    null

this-expression:

    this *label-reference*$_{opt}$

super-expression:

    super *supertype-reference*$_{opt}$ label-referenceopt

supertype-reference:

    < *type* >

label-reference:

    @ <nospace> *identifier*

invocation-expression:

    postfix-unary-expression type-arguments$_{opt}$ argument-list$_{opt}$ trailing-lambda
    postfix-unary-expression type-arguments$_{opt}$ argument-list
    postfix-unary-expression type-arguments

TODO: Specify rules that guarantee correct grouping in parsing of types `() -> Int?` and `() → Unit.() → Unit`. TODO: Specify places where parsing as blocks is preferred to parsing as function literals (`if`, `else`, `when`, loops). Block is parsed as a function literal if it has at least one leading label or annotation (only if there is a potential ambiguity – there are places where it can only be parsed as a block, e.g. after `try` keywords, and no labels or annotations are allowed there). Parsing of else branches in nested `if` statements (possibly within a `when` statement). When newline commits a statement, and when statement is parsed greedily.

Possible empty statements: `if(true) else;`

[Example: Parsing functional types before → token in lambdas and with statements. *End Example]*

[Note: Parenthesization or leading → forces parsing of a block as a function literal. _End Note]

4.3.1. Grammar Ambiguities

TODO: Specify what are possible grammar ambiguities (if any), and how they shall be handled.

5. Symbols

5.1. General

Programs can declare and reference different kinds of entities called symbols. Examples of symbols are packages, classes, functions, parameters or variables. A symbol can have a name, or can be anonymous. Symbols can be introduced using syntactic constructs called declarations, or be implicitly declared. Every symbol that is not a package can have at most one declaration. A package can have one or more declarations.

5.2. Declaring Symbol

Every symbol (except TODO?) has exactly one declaring symbol. Usually, the declaration of a symbol is immediately textually contained within the declaration of its declaring symbol. For example, the declaring symbol of a method is the class in which it is declared, and the declaring symbol of a parameter is the method in which signature the parameter occurs. Sometimes, but not always, a symbol is a member of its declaring symbol. In the previous example, the method is a member of its declaring class, but the parameter is not a member of its declaring method.

5.3. Implicit Declarations

TODO: What symbols can be declared implicitly (e.g. default constructor, or `it` parameter in lambdas, `field` in accessors).

5.4. Modifiers

Syntax for some declarations allow to provide a modifier list that may contain zero or more modifiers. Each declaration kind can have its own rules governing allowed modifiers and valid combinations of them, but general rules are given in this section.

A modifier is one of the following keywords:

```
abstract annotation companion crossinline data enum external final in infix inline inner internal
lateinit noinline open operator out override private protected public reified sealed tailrec vararg
```

Among those, only `in` is a hard keyword, all others are soft keywords that are reserved only in a modifier position.

An order of modifiers in a modifier list is not significant. The same modifier cannot appear more than once in the same modifier list. Modifiers `public`, `private`, `protected` and `internal` are called *visibility modifiers*, they are used to specify the visibility of a declared symbol. Only declarations of type members can have `protected` modifier.

The following pairs of modifiers are incompatible in the same modifier list (in any order, possibly separated by other modifiers):

```
final open
final abstract
final sealed
open sealed
```

The modifier `open` is redundant if `abstract` is specified. Modifier `abstract` is redundant if `sealed` is specified.

5.5. Visibility

Every symbol has an associated visibility domain – a region or regions of program text where it could be referenced from explicitly or implicitly. Explicit reference to a symbol involves a textual usage of its name (at least simple name), while implicit does not. Every case that constitutes an implicit reference to a symbol is explicitly called out in the specification. [Example: Declaration of an implicitly typed local variable whose type T is inferred from its initializer constitutes an implicit reference to the type T. End Example] The visibility domain of a symbol S is the intersection of the visibility domain of its declaring symbol and the declared visibility of the symbol S. In case S does not have a declaring symbol, its visibility domain is just its declared visibility.

5.5.1. Declared Visibility

The declared visibility of a symbol is determined by a visibility modifier present at the symbol declaration, if any.

5.6. Packages

All declarations in a program are grouped in packages. Different packages can have declarations of the same kind and with the same simple. Thus, packages allow to avoid name clashes, serve as namespaces and enable grouping of the symbols having related functionality. A package name is a sequence of one of more (not necessary distinct) identifiers separated by dots. There is also a single default package that has no name.

TODO: define "module"

A single module can contain declarations from multiple packages, and symbols in the same package can be declared in different modules. Declarations in a single file always introduce symbols into the same package, whose name specified in the package directive in the file header. If a file contains no package directive, the declarations in this file introduce symbols into the default package.

5.7. Classes

TODO

5.8. Interfaces

TODO

5.9. Properties and Fields

TODO

5.10. Methods and Functions

5.10.1. General

A function is a callable fragment of code that has zero or more parameters, a return value and a set of local variables. Each invocation of a function creates a separate set of storage locations for its parameters and local variables, independent from other invocations of the same function, referred to as a logical stack frame. A logical stack frame can be different from a physical stack frame because of tail call optimizations, inlining and other reasons. Lifetime of some parameters and variables can follow special rules if they are captured in a closure (§Closure).

Evaluation of a function can cause an invocation of another function (or a recursive invocation of the same function) that effectively suspends evaluation of the current function, pushes a new frame on the call stack, and transfers control flow to the beginning of the body of the callee. If evaluation of a function completes normally, the topmost stack frame is discarded, and control flow is

returned back to its caller, and its evaluation is resumed at the point immediately following the invocation (the return value, becomes the value of the completed invocation expression). Evaluation of a function can also result in an exception (§Exception), which can be either caught and handled inside the same function, or result in an abrupt completion of the function, and propagation of the exception along the stack to a closest direct or indirect caller that can catch and handle it.

Functions can be either named or anonymous (lambdas). Named functions can be either top-level functions, local functions or member functions (methods).

5.10.2. Type-parameter List

A type-parameter list specifies names and bounds of the type-parameters of a function. Syntactically, a type-parameter list is one or more type-parameter declarations, separated by commas, and enclosed in angle brackets. Names of all type-parameters in a type-parameter list must be distinct.

5.10.3. Parameter List

A parameter list specifies names and types of the parameters of a function. Syntactically, a parameter list is zero or more parameter declarations, separated by commas, and enclosed in parentheses. Names of all formal parameters in a formal parameter list must be distinct. A name of a formal parameter in a formal parameter list cannot be the same as a name of a type-parameter in a type-parameter list associated with the same function.

[Example:

```
fun <T> f(T : Int) // error
```

End example]

TODO: type-parameter list, parameter list.

5.10.4. Generic Functions

A function having a type-parameter list is a generic function.

TODO

5.10.5. Local and Top-Level Functions

TODO

5.11. Locals

Read-only local variables are declared using `val` keyword. Mutable local variables are declared using `var` keyword.

5.12. Parameters

Parameters behave similarly to local variables. They are always immutable and always definitely assigned. Lifetime of parameters and their behavior when captured in a closure are the same as for local variables. TODO

5.13. Singleton Objects

An object declaration introduces two symbols of different kinds, but with the same name: a class and a value that is the only instance of that class. The class is final and is guaranteed to have exactly one instance. *[Note:* there are some caveats about initialization time of such instances in cases their declaration depend on each other in a circular manner. *End Note]* Unless explicitly stated otherwise, object declarations follow the same rules as class declarations. An object declaration can specify a superclass and zero or more superinterfaces. An object declaration cannot have type-parameters or access type-parameters from an outer scope. An object declaration can have the following modifiers: `companion`, `final`, `internal`, `private`, `protected`, `public`. An object declaration cannot be `inner`, `open` or `abstract`. Although the `final` modifier is allowed, it is completely redundant on object declarations. An object declaration can have no constructors declarations (but has an implicit private parameterless primary constructor that is automatically invoked during object initialization, and can contain `init` blocks that are executed as parts of that primary constructor). An object declaration cannot be local, and cannot be enclosed in a local class declaration. An object declaration can be top-level or be declared within a class declaration, an interface declaration, or an object declaration (possibly, within a companion object). An object declaration cannot access `this` instances of enclosing types either explicitly or implicitly. [TODO: clarify logical connectives in this section]

A class nested immediately within a singleton object cannot be an inner class (but it is possible to have an inner class deeper in the nesting hierarchy).

A singleton object (except companion objects) can not declare a member with the `protected` modifier, and can not declare a property whose accessor has the `protected` modifier.

Singleton objects are initialized in an order determined by trying to topologically sort them by their dependencies. In case of an initialization cycle a complete topological sort is impossible, and it is possible to observe the value of an object involved in the cycle as `null`. This can cause an exception if this value is passed to a function expecting a non-nullable type.

[Example:

```
abstract class A(val x : Any?)
object B : A(C)
object C : A(B)

fun main(args: Array<String>) {
    println(B.x)
    println(C.x) // null
}
```

End Example]

TODO

5.13.1. Companion Objects

Companion object can be declared within classes or interfaces, including generic ones. An optional name can be specified for a companion object, if no name is specified, the default name `Companion` is used. The name of a companion object (either explicit or default) must be distinct from names of all other members declared in the containing type. No more than one companion object per class or interface is allowed. When the name of a companion object is accessed through dot on the name of the containing class or interface, no type-arguments can be provided to the containing type name, even if the type containing is generic. A companion object can specify at most one superclass and zero or more superinterfaces. [Note: There is no rule that would prevent the supertype of a companion object to be the containing type of the companion object. In such a case, if the containing type is generic then type-arguments must be provided after its name in the supertype list, but those type-argument cannot use type-parameters of the containing type, because they are not in scope throughout the companion object declaration. End Note].

The body of a companion object is optional. If no body is provided, the empty body `{ }` is assumed. Members of a companion object can be accessed directly through the dot on its containing type name (no type-arguments must be provided even if the containing type is generic). The name of a companion object hides its identically named members in this context (the hidden members are still available by lookup in the name of the companion object in this case). [Example: TODO End example] TODO: This hiding behavior is actually not that simple for method names.

A type and its companion object have access to private members of each other, regardless of declared visibility of the companion object.

If a companion object is declared within a generic type, there is still only one instance of a companion object, despite that potentially multiple instantiations of the generic type is possible. Type-parameters of containing types are not available in the declaration of a companion object.

A companion object cannot be declared within a local or inner class.

Declared visibility `private` or `protected` of members of a companion object is understood with respect to its container type rather than to the companion object itself (TODO: implementation bug with `protected`).

TODO: What does it mean for a companion object to be private.

TODO: The name of a containing type can itself be used as an expression that refers to the companion object.

5.13.2. Labels

Certain locations in code can be named for later reference by declaring labels.

TODO: Declaration space and scope for labels

6. Values and Types

6.1. General

A value can be a reference to an object, the null value (a special value compatible with any nullable value type and distinct from a reference to any object) or a primitive value.

It is up to an implementation to choose (for performance optimization, memory optimizations or any other reason) how to store a primitive value in each particular case. This choice does not have to be uniform across the program. A variable can store a primitive value directly, or store a reference to a boxed copy of a primitive value — the language does not provide any means to force either behavior. There can exist multiple references to the same boxed instance, and there can be multiple boxed copies of the same primitive value. A result of the reference equality operator applied to primitive types is unspecified.

Variables of reference types store references to objects rather than objects themselves. An object reference points to an instance of a reference type. Mupliple variables can store references to the same object, and any modifications to the object made through one of those references are observable through all the others. As many references to a single object as needed can be created by a simple assignment (and some other language constructs), but in general there is no way to create a copy of an object instance, unless this feature is explicitly supported by its class. The terms "object" and "object instance" are considered synonyms and are used interchangeably throughout this specification.

Lifetime of objects are managed automatically. There is no need or possibility to explicitly destroy object instances after they are created. An object instance is guaranteed to exists as long as there is at least one reference to it that can be used in an observable way on at least one possible control flow path through the current program (but is permitted to exist longer than required). Provided that this guarantee is satisfied, it is otherwise up to an implementation to decide when objects are destroyed and when memory occupied by them is reclaimed and made available for future allocations. *[Note:* A particular platform may choose to provide an API to monitor or control this process to some degree, but such an API is not required by the language and is out of scope of this specification. *End note]*

The null value is a special immutable atomic (structureless) value. Its exact type is `Nothing?` (it is the only value of this type), and it also belong to every nullable type, usually indicating an absence of any valid value of the corresponding non-nullable type.

TODO: Move detailed discussion of arrays to corresponding section?

An object can be considered an instance of multiple (a finite or infinite number) different types, but the set of these types always contains a single type that is a subtype of every other type in the set and is called the exact type of the object. The exact type of an object object is always either a concrete (not-abstract) class or an array type. *[Example:* TODO *End example]*

An object is either an instance of a class, or an instance of an array type. An array type is either

generic class `Array<T>` or one of the types representing arrays of primitives:

- `BooleanArray` — an array of values of type `Boolean`
- `ByteArray` — an array of values of type `Byte`
- `ShortArray` — an array of values of type `Short`
- `IntArray` — an array of values of type `Int`
- `LongArray` — an array of values of type `Long`
- `CharArray` — an array of values of type `Char`
- `FloatArray` — an array of values of type `Float`
- `DoubleArray` — an array of values of type `Double`

Arrays are different from classes with respect to how they store their data. A class always have a fixed number of storage locations (called fields), the fields potentially can have types, and each instance whose exact type if the class will have that fixed number of fields. If an object with a different number of fields is needed, it is necessary to declare a new class. An array type, on the other side, only defines a type of its storage locations (called elements), but leaves their number undefined. The number of elements of an array is specified only at runtime when a particular instance of that array type is created (this number can be zero or more, and is limited only by platform or available memory). This number remains constant during the whole lifetime of that particular instance, but can be different for different instances although they all belong to the same array type. Also, unlike fields of a class that can be declared immutable and therefore their values cannot be changed after the instance is created, all array elements are always mutable. A field is referred to by its name (or, indirectly, by a name of the corresponding property), but an array element is referred to by its index. An index indicates an offset from the first element of the array (so that the first element has index 0, the second element has index 1, and so on to the last element of the array whose index is equal to the length of the array minus one). [Note: This is known as the zero-based indexing. End note] Indices less than 0, or greater or equal to the length of the array are invalid, and an attempt to access an element by such an index will result in an exception (WHICH?) at runtime. An array of length 0 is called an empty array, and it has no valid indices — any attempt to access its element results in an exception.

[Note: It should not be expected that an implementation will try to pack elements of BooleanArray to use only 1 bit of storage per element. Normally, 1 byte of storage per element will be used. *End Note]*

- TODO: Sort out terminology: we say that `Array<T>` is a class, but make a distinction between array instances and class instances.

Each value has a type. A type is an abstraction capturing common characteristics (e.g. a set of supported operations and information about their possible results) of a family of similar values. Informally, values that belong to the same type are either identical or have similar shape and behavior. The exact scope of this similarity is detailed in this specification. Usually, many different values belong to the same type, but there are types having only a single value, and there are also uninhabited types having no values — they are purely compile-time abstractions, sometimes intentional (like the `Nothing` type), sometimes just a byproduct of the imperfect type system. Ideally, types that have the same set of values and allow the same set of operations on them would be

always consideder identical. In fact, the type system only provides an approximation to this goal.

While exact values of expressions are usually known only at runtime (except constant expressions that are fully evaluated at compile time), types can be known at compile time, so the compiler can reason about them, and guarantee certain desirable properties of a program (e.g. that a certain method is never invoked on an object that does not support it).

One of the main purposes of the Kotlin type system is to maintain the following condition: a storage location can only contain a value that matches its type, and that the result of evaluation of an expression can only be a value that matches its type. This condition is guaranteed to hold, except in a few special circumstances: a heap pollution, a null leak, or an abuse of reflection. All this circumstances are explained in details in corresponding sections of the specification (§TODO).

Values and types are distinct notions and they do not mix (unlike in some programming languages featuring dependent types). Nonetheless, some types can be described using values of type KClass<T>, and their properties can be queried at runtime, using a technique called reflection. It is important to remember that those values are not types, they just describe or represent types.

Each value has a single exact type. This is the narrowest type representable in the language, to which the value belongs.

A type can also be thought as representing a set of all possible values having this type. If a type A represents a subset of values, represented by a type B, the type A …

TODO: only subtyping defined by rules

TODO: exact type, reflection and erasure

6.2. Syntax for Types

TODO A type can be parenthesized, it does not change its meaning.

6.3. Compile-time and Run-time Types

TODO

6.4. Reified Types

A captured type-argument cannot be used as a type-argument for a reified type-parameter. The types kotlin.Nothing, kotlin.Nothing? cannot be used as type-arguments for a reified type-parameter.

DISCUSS: Report an error for: inline fun <reified T : Nothing?> foo(x : T) { }

A type-parameter of a function can be marked as `reified`:

```
inline fun foo<reified T>() {}
```

6.4.1. Semantics, Checks and Restrictions

Definition A well-formed type is called **runtime-available** if - it has the form `C`, where `C` is a classifier (object, class or interface) that has either no type-parameters, or all its type-parameters are `reified`, with the exception for class `Nothing`, - it has the form `G<A1, ⋯, An>`, where `G` is a classifier with `n` type-parameters, and for every type-parameter `Ti` at least one of the following conditions hold: - `Ti` is a `reified` type-parameter and the corresponding type-argument `Ai` is a runtime-available type, - `Ai` is a **star-projection** (e.g. for `List<*>`, `A1` is a star-projection); - it has the form `T`, and `T` is a `reified` type-parameter.

Examples: - Runtime-available types: `String`, `Array<String>`, `List<*>`; - Non-runtime-available types: `Nothing`, `List<String>`, `List<T>` (for any `T`) - Conditional: `T` is runtime-available iff the type-parameter `T` is `reified`, same for `Array<T>`

Only runtime-available types are allowed as - right-hand arguments for `is`, `!is`, `as`, `as?` - arguments for `reified` type-parameters **of calls** (for types any arguments are allowed, i.e. `Array<List<String>>` is still a valid type).

As a consequence, if `T` is a `reified` type-parameter, the following constructs are allowed: - `x is T`, `x !is T` - `x as T`, `x as? T` - reflection access on `T`: `javaClass<T>()`, `T::class` (when supported)

Restrictions regarding reified type-parameters: - Only a type-parameter of an `inline` function can be marked `reified` - The built-in class `Array` is the only class whose type-parameter is marked `reified`. Other classes are not allowed to declare `reified` type-parameters. - Only a runtime-available type can be passed as an argument to a `reified` type-parameter

Notes: - No warning is issued on `inline` functions declaring no inlinable parameters of function types, but having a `reified` type-parameter declared.

6.4.2. Implementation notes for the JVM

In inline functions, occurrences of a `reified` type-parameter `T` are replaced with the actual type-argument. If actual type-argument is a primitive type, its wrapper will be used within reified bytecode.

```
open class TypeLiteral<T> {
    val type: Type
        get() = (javaClass.getGenericSuperclass() as
ParameterizedType).getActualTypeArguments()[0]
}

inline fun <reified T> typeLiteral(): TypeLiteral<T> = object : TypeLiteral<T>() {} //
here T is replaced with the actual type

typeLiteral<String>().type // returns 'class java.lang.String'
typeLiteral<Int>().type // returns 'class java.lang.Integer'
typeLiteral<Array<String>>().type // returns '[Ljava.lang.String;'
typeLiteral<List<*>>().type // returns 'java.util.List<?>'
```

6.5. Nullable Types

TODO Regular reference types (classes, interfaces and arrays) do not support the null value. The language guarantees that all variables of those types are always initialized with a valid object reference prior to their use, and expressions of those types always evaluate to a valid object reference (if their evaluation completes normally).

Because type-parameters can be instantiated both with non-nullable and nullable types (unless their bounds indicate otherwise), the language must handle them conservatively: it does not allow assigning null value to variables whose type is a type-parameter (because it may be instantiated with a non-nullable type), but when reading from them, it cannot assume that the value is not null unless it can be proven via static data-flow analysis (because it also may be instantiated with a nullable type). Every reference type, type-parameter or a primitive type `T` has the corresponding nullable type `T?`. The type `T` is called the underlying type of the nullable type `T?`. It is permitted (although completely redundant) to write `T?` even if `T` is already a nullable type. The set of valid values of a nullable type is the set of valid value of its underlying type plus the additional null value. The null value is denoted using the keyword `null`. The type `kotlin.Nothing?` has only the null value as its valid value and is a subtype of every nullable type. The type of the null literal is `kotlin.Nothing?`. The type `kotlin.Any?` can store any value whatsoever, and is a supertype of every nullable and every non-nullable type.

TODO: Say more about instantiation of type-parameters.

A nullable type cannot be specified as a supertype in a class or interface declaration. A nullable type cannot be used on the left hand side of a dot (either in a type context or expression context), except as a receiver type in extension functional types.

TODO: It looks reasonable to support a non-nullable version `T!!` of type-parameter `T` that is not known to be non-nullable. If the type-parameter `T` is instantiated with a non-nullable type then `T!!` is the same type as `T`. Otherwise, the type-parameter `T` is instantiated with some nullable type `S?`, and `T!!` is the same type as its underlying non-nullable type `S`. Non-nullable versions of type-parameters can be propagated into parameters of lambdas by smart casts. Effectively, `T!!` would be equivalent to the intersection type `T & Any`. *[Note:* The notation T!! should not be confused with the notation T! used for platform types. *End Note]*

6.6. Primitive and Special Types

A primitive type is usually represented by the platform as a fixed-length bit pattern, and can be directly and efficiently manipulated by the hardware. But the language does not prescribe any particular representation for primitive type, but only requires that their observable behavior corresponds to the following descriptions. Primitive values do not have simpler components representable in the language, do no share any state with other primitive values and are effectively immutable. [Note: Individual bits of integer types can be extracted using bitwise operations, see §?. End Note]

- `Boolean` — the Boolean type having exactly two values, represented by literals `true` and `false`.

- `Byte` — a signed (2-complement) 8-bit integer type.

> Short — a signed (2-complement) 16-bit integer type.

- Int — a signed (2-complement) 32-bit integer type.

- Long — a signed (2-complement) 64-bit integer type.

- Char — unsigned 16-bit integer type, intended to represent UTF-16 code points.

- Float — a single-precision 32-bit IEEE 754 floating point type.

- Double — a double-precision 64-bit IEEE 754 floating point type.

Due to different representations, smaller types are not subtypes of bigger ones. Kotlin supports the standard set of arithmetical operations over numbers, which are declared as members of appropriate classes (but the compiler optimizes the calls down to the corresponding instructions).

Primitive types do not have non-private constructors, but their instances could be represented using literal of corresponding types.

6.6.1. The kotlin.Any Type

All classes in Kotlin have a common ultimate superclass `kotlin.Any`. `kotlin.Any` is the default superclass for a class with no superclass declared. `Any` is not `java.lang.Object`, in particular, it does not have any members other than methods equals(), hashCode() and toString(). It is the root on the hierarchy of all non-nullable types. To declare a variable that can store a reference to a value of any type (nullable or not), the nullable type `kotlin.Any?` can be used.

All interfaces are considered to be subtypes of `kotlin.Any`.

`kotlin.Any` has the following parts:

```
public constructor()
public open fun equals(other: Any?): Boolean
public open fun hashCode(): Int
public open fun toString(): String
```

The constructor implementation is an empty block that does nothing.

The default implementation of the method equals performs reference equality check. It can be overridden in derived types to implement a custom equality behavior. It is strongly recommended that any implementation of this method satisfies the following rules:

- Termination. The method shall complete normally (i.e. it shall not go into an infinite loop or throw an exception).

- Purity. An invocation x.equals(y) shall not change any observable state of objects x and y (it may update some caches though).

- Reproducibility. Multiple invocations x.equals(y) shall return the same result (unless state of either x or y has changed in between).

- Reflexivity. For any non-null value x the expression x.equals(x) shall return true.

- Symmetry. For any non-values x and y expressions x.equals(y) and y.equals(x) shall return the

same value (unless state of either x or y has changed in between).

- Transitivity. Invocations x.equals(y) and x.equals(z) shall return the same value if y.equals(z) returns true (unless state of either x, y or z has changed in between).

[Note: Many functions from the standard library may produce unexpected results if any of these rules is violated. It is reasonable to expect implementations of the `equals` method to be conformant to these rules when writing library code, but one should not rely on it when reasoning about code security and other critical properties, because a malicious client could exploit it and create a security breach. *End note]*

The default implementation of the hashCode method returns a hash code for the current object, consistent with the behavior of the equals objects (i.e. equal objects always have the same hash code).

- TODO: `toString` method

`Any` is an `open` non-`abstract` class.

6.6.2. The kotlin.Nothing Type

The type `kotlin.Nothing` is an uninhabited type (i.e. no value can have this type at runtime). Consequently, an evaluation of an expression of type `kotlin.Nothing` never completes normally (for example, it may be a non-terminating computation, may throw an exception, or result in a control flow transfer). The type `kotlin.Nothing` is a subtype of every other type. The corresponding nullable type `kotlin.Nothing?` can have the only value null. Neither `kotlin.Nothing` nor `kotlin.Nothing?` can be used as a reified type-argument.

If a program was compiles with unchecked warnings (possibly, suppressed), and an evaluation of an unchecked cast resulted in a heap pollution, it may be possible to have an expression with compile type `Nothing` that will have a value of some inhabited type.

The type `Nothing` does not have non-private constructors and there is no way to create an instance of this type.

`Nothing` is a `final` class.

6.6.3. The kotlin.Unit Type

The `kotlin.Unit` is a singleton type whose directs superclass is `kotlin.Any`. It does not implement any interfaces. It does not have any members beyond those inherited from `kotlin.Any`. The expression `kotlin.Unit` represents the single instance of the `kotlin.Unit` type. This type may be used as a return type of any function that does not return any meaningful value.

6.6.4. The kotlin.Byte Type

TODO: Describe behavior of the interface members implemented by numeric types.

`Byte` is a `final` class.

6.6.5. The kotlin.Short Type

TODO

`Short` is a `final` class.

6.6.6. The kotlin.Int Type

TODO

`Int` is a `final` class.

6.6.7. The kotlin.Long Type

TODO

`Long` is a `final` class.

6.6.8. The kotlin.Char Type

The type `kotlin.Char` represents a UTF-16 code point, rather than a Unicode character. Some Unicode characters are represented by a pair of surrogates, in which case each of the surrogates is a separate instance of the `kotlin.Char` type.

`Char` is a `final` class.

TODO

6.6.9. The kotlin.Float Type

TODO

`Float` is a `final` class.

6.6.10. The kotlin.Double Type

TODO

`Double` is a `final` class.

6.6.11. The kotlin.Boolean Type

The type `Boolean` represents Boolean values. It has exactly two possible values: `true` and `false`. These values can be syntactically expressed using `true` and `false` Boolean literals. *[Note:* It should not be expected that an implementation will try to pack Boolean values to use only 1 bit of storage per variable. Normally, 1 byte of storage per variable will be used. *End Note]*

TODO: Describe behavior of the interface members implemented by this type.

TODO

`Boolean` is a `final` class.

6.6.12. The kotlin.Array<T> Type

The `kotlin.Array<T>` represents a flat array of elements of type `T`. Unlike other classes whose constituent values are stored in fields, an array's constituent values are its elements. Unlike classes that specify number of their fields in their declarations at compile-time, an array type does not specify the number of its elements. This number is specified when a particular instance of an array is created, and remains unchanged for the lifetime of that instance.

All elements of an array are effectively public, and can be read or updated by any code that has access to the array itself.

The type `Array<T>` has the only public constructor `Array(size: Int, init: (Int) → T)`, no other non-private constructors. There are library methods that enable to create array instances.

When kotlin.Array<T> is used with primitive types, its elements are stored in a boxed form that may not be very efficient. For flat array of primitive types, the standard library provides non-generic types `ByteArray`, `ShortArray`, `IntArray`, `LongArray`, `CharArray`, `BooleanArray`, `FloatArray`, `DoubleArray`. These classes have no inheritance relationship to the `Array<T>` class, but they have similar characteristics and similar sets of methods and properties.

Each of arrays of primitive types has two public constructors:

```
ByteArray(size: Int) // TODO: specify behavior
ByteArray(size: Int, init: (Int) -> Byte)

ShortArray(size: Int)
ShortArray(size: Int, init: (Int) -> Short)

IntArray(size: Int)
IntArray(size: Int, init: (Int) -> Int)

LongArray(size: Int)
LongArray(size: Int, init: (Int) -> Long)

CharArray(size: Int)
CharArray(size: Int, init: (Int) -> Char)

BooleanArray(size: Int)
BooleanArray(size: Int, init: (Int) -> Boolean)

FloatArray(size: Int)
FloatArray(size: Int, init: (Int) -> Float)

DoubleArray(size: Int)
DoubleArray(size: Int, init: (Int) -> Double)
```

and has no other non-private constructors. There are library methods that enable to create array

instances of these types.

`Array<T>` and all primitive array types are `final` classes.

6.6.13. kotlin.String class

Values of the class `String` are strings, i.e. finite immutable sequences of UTF-16 code points. A string can have zero or more code points (their length is limited only by platform or available memory). Strings are immutable, meaning that neither their lengths, nor any of their elements can be modifier after their creation.

The type `String` has the only public constructor `String()` that creates an empty string, and no other non-private constructors. There are library methods named `String` that enable creation of string instances.

`String` is a final class.

6.6.14. kotlin.KClass<T> class

TODO

6.6.15. The dynamic Type

The type dynamic may be supported not on every platform. A supertype cannot be the dynamic type. TODO: Subtyping relationships for dynamic.

The type `dynamic` has no constructors.

6.7. Functional Types

There are two kinds of functional types: free functional types and extension functional types. A free functional type specifies zero or more parameter types, and a return type. An extension functional type in addition specifies a receiver type (before the parameter list, separated by the dot).

If a function does not return any meaningful value (for example, is invoked for the sake of its side effects), it may be declared with kotlin.Unit return type. If a function never returns normally (for example, contains an infinite loop, always throws an exception, or terminates the process), it may be declared with kotlin.Nothing return type.

Functional types are interfaces, not classes, and so do not have any contructors. There are predefined types implementing these intefaces, and their instances are created implicitly (for example, when converting an anonymous function to a function type).

6.8. Platform Types

TODO

6.9. Type Relationships

It is possible that two types having different syntax forms are considered equivalent (even ignoring the possibility of fully qualified names and name aliases). In particular:

- Nested nullable types are equivalent to the corresponding single nullable types (e.g. `String??` is equivalent to `String?`).
- Star-projection is equivalent to the corresponding out-projection (or, in contravariant case, `G<Nothing>`).
- Functional types are equivalent to corresponding named types.

TODO: Type equivalence, inheritance, subtyping, interface implementations.

6.9.1. Restrictions

An inheritance hierarchy shall be acyclic, that is more precisely described by the following rule. Construct a directed graph, where all the declared types in the program (classes, interfaces and objects) are represented by vertices, and there is an edge from type A to type B if the declaration of A lists B among its supertypes (ignoring type-arguments) or if the declaration of type A is directly nested in the declaration of type B. It is an error if the constructed directed graph contains a cycle. It means, for example, that it is an error for a class to inherit from itself, or for two classes to inherit from each other, or for a class to inherit from its nested class.

[Note: It is syntactically impossible for a top-level class to specify a local class as its superclass. *End note]*

6.9.2. Single Instantiation Rule

It is an error if the transitive closure of supertypes of a type contains two different instantiations of a generic type.

TODO: do we permit any flexibility for co- or contravariant types?

TODO: no expanding generic graphs, single instantiation inheritance.

6.10. Conversions

TODO

7. Generics

7.1. General

A class, interface, function or property declaration may contain a type-argument list that declares one or more type-parameters. Such a declaration is called generic. It is an error if two or more type-parameters in the same parameter list have the same name. The scope of type-parameters extends to the entire symbol declaration to which they belong (including supertype list, the entire type-

argument list itself, and any fragments of the declaration that may appear to the left of it). Names of type-parameters can be used like regular type names within their scope. (TODO: nested and inner classes) A generic type declaration serves as a blueprint for multiple type declaration having the same shape. A particular instance of a generic type is obtained by providing type-arguments to its name. Type-arguments are substituted to all occurrences of corresponding type-parameters in the generic type declaration. The substitution is semantic rather than pure textual. For example, the name of a type provided as a type-argument may be hidden or invisible at a particular occurrence of a corresponding type-parameter, so a pure textual substitution of the name would result in that name being unresolved, or bound to an unrelated symbol. *[Note:* A type-parameter within its scope may be used as a type-argument, or otherwise appear as a constituent of a type-argument. *End Note] [Note:* Generic declarations promote better type safety. For example, rather than having a non-generic `ArrayList` that can store elements of any type, and downcasting its elements to a particular type when they are extracted from list, we can declare a generic `ArrayList<T>` and to use its particular instantiation, e.g. `ArrayList<Int>` or `ArrayList<String>` when we need a list of integers or a list of strings, respectively. This way we cannot put an element of a wrong type to a list, and we do not need to downcast when we extract elements. *End Note]*

TODO: Description of generic types and methods, type substitution, scope of type-parameters, type-parameter bounds, projections and their members, skolemization (aka capture conversion), raw types, erasure, single instantiation inheritance rule.

7.2. Type-Parameters

Type-parameters of types and methods. Scope of type-parameters, interaction with nested classes, interaction with local classes. Identity of type-parameters, comparison of signatures, internal and external view of type-parameters, multiple instantiations (cf. TS). Names and ordinal positions. Substitutions, composition of substitutions. Open and closed instantiations, instance type (type of this). Originating declaration. Generic arity.

A type-parameter can never escape its scope — no expression can have a type containing a type-parameter outside of its scope (this does not mean a type-parameter is always denotable — it still can be hidden by other symbols). Thus, if a name in a code refers to a type-parameter, it can never be a type-parameter declared in a library, or even in a different compilation unit (the latter would become false if Kotlin ever introduced partial types).

A type-parameter can only be denoted using a simple name, never using a qualified name (it means that if a type-parameter is hidden, it is not denotable).

A type-parameter name can never appear as the left hand side of a member access operator (dot), but it can appear as a receiver type in extension functional types.

No two type-parameters in the same type-parameter list can have the same name.

TODO

7.3. Instantiation

7.3.1. General

TODO

7.3.2. Type-Argument List

Type-argument list can be provided either to a generic type name to specify a type instantiation or projection, or to a generic method name to specify type-arguments to this method. Type-argument list is delimited by angle brackets and contain one or more type-arguments separated by commas. A type-argument can be either a type or a projection specifier. A projection specifier can only appear in a type-argument list provided to a generic type. It's an error if it appears in a type-argument list for a method. A type-argument list cannot contain zero arguments. In certain contexts, where no type-arguments are to be provided, the whole type-argument list, including delimiting angle brackets, are omitted. In certain contexts a generic type can be used without providing type-arguments – in this case the whole type-argument list, including delimiting angle brackets, is omitted. It is an error to provide an empty type-argument list: `T<>`. Number of type-arguments provided in a generic type instantiation must match generic arity of the generic type. It is an error to provide a type-argument list to a non-generic type. It is an error to provide a type-argument list to an identifier that does not denote a type or a method. *[Note:* certain language constructs (e.g. `super`) allow to specify a type enclosed in angle brackets that may look similar to a type-argument list, but those syntax constructs are not type-argument lists. *End Note]*

TODO: Discuss nested types.

7.4. Substitution

TODO

7.5. Bounds

A type-parameter can have zero or more bounds. A bound is a type that can possibly depend on the type-parameter itself, or on other type-parameters from the same type-parameter list, or on other type-parameters from an outer scope).

A bound for a type-parameter can be specified at its declaration after a colon (so called primary bound). Also, one or more bounds can be specified in a where clause. Bounds in a where clause can be specified even if primary bound is not present. Order of bounds is not important, except that the leftmost bound is used to determine erased signature on platforms supporting erasure (that can result in a signature clash, for example).

A type-parameter cannot specify itself as its own bound, and several type-parameters cannot specify each other as a bound in a cyclic manner. More precisely, for each type-parameter list, construct a directed graph where all type-parameters declared in the type-parameter list are represented as vertices, and there is an edge from type-parameter `T` to type-parameter `S` iff `T` has `S` or `S?` as its bound. It is an error if the constructed directed graph contains a cycle.

It is not syntactically possible for two type-parameters in different type-parameter lists to specify each other as a bound.

It is an error for a bound of a type-parameter to be `Nothing`, or for several bound to have the intersection `Nothing` (`Nothing?` is allowed). Two bounds of the same type-parameter cannot be different instantiations of the same generic type (TODO: in transitive set)

The same bound cannot be specified more than once for the same type-parameter (BUG: implement)

> TODO: Bounds, self-referential bounds, bound satisfaction, substitution. When exactly satisfaction is checked (nested generics, bounds satisfaction in bounds).

7.6. Variance

TODO: Co- and contravariance, safe occurrences of variant type-parameter (including occurrences in type constraints, inner type constraints, and method constraints), safe invocations of private members. Variant conversions, infinite recursion issues, expanding cycles.

TODO: Workaround for `add` methods (extension methods).

7.7. Projections

TODO: Projections, valid occurrences of projections, members of projections. Meaning of projections of variant types. Interaction of declaration-site and use-side bounds, interaction with invisible types. Projections of a generic type in bounds in its own declaration.

7.8. Type Inference

TODO: Type variables, and fixed (external) type-parameters (can originate from the same declaration). Type inference session (can include type variables from multiple declarations or multiple instances of the same declaration). Identity of type variables.

Type inference is a process that tries to infer type-arguments in a generic function invocation (or a group of generic function invocations, chained or nested) where no explicit type-arguments are provided from function arguments and the expected type (if one is present), such that the inferred type-arguments satisfy the bounds on the corresponding type-parameters, and, after substitution of the type-arguments in the function signature, the function is applicable to the provided arguments and the result of the function is assignable to the expected type (if one is present). Type inference is conservative — it may fail to find suitable assignment of type-arguments to the type-parameters despite that one exist, and the function invocation could be successfully typechecked if the programmer explicitly provided suitable type-arguments. Also, it is possible that multiple different valid assignments of type-arguments are possible — in this case, if the type inference succeeds, it will deterministically select one of those assignments. It is possible that the selected assignment leads to a compile-time error elsewhere, that could be avoided if the programmer explicitly provided a different assignment of type-arguments. It is also possible that type inference infers type-arguments that for some reason are not denotable in the given location in the source code (e.g. because they contain captured type variables or type parameters hidden by other declarations), and so could not be provided explicitly. This condition by itself does not prevent the generic function invocation from being typechecked successfully.

7.9. Restrictions on Usage

A generic class cannot have `Throwable` as a superclass.

Classes in Kotlin may have type-parameters:

```
class Box<T>(t: T) {
  var value = t
}
```

To create an instance of such a class, the type-arguments have to be provided:

```
val box: Box<Int> = Box<Int>(1)
```

But if the parameters may be inferred, e.g. from the constructor arguments or by some other means, type-arguments can be omitted:

```
val box = Box(1) // 1 has type Int, so the compiler figures out that we are talking
about Box<Int>
```

• Declaration-site variance

Suppose we have a generic interface `Source<T>` that does not have any methods that take `T` as a parameter, only methods that return `T`:

```
// Java
interface Source<T> {
  T nextT();
}
```

Then, it would be perfectly safe to store a reference to an instance of `Source<String>` in a variable of type `Source<Object>` — there are no consumer-methods to call. But Java does not know this, and still prohibits it:

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!! Not allowed in Java
  // ...
}
```

To fix this, we have to declare objects of type `Source<? extends Object>`, which is sort of meaningless, because we can call all the same methods on such a variable as before, so there's no value added by the more complex type. But the compiler does not know that.

In Kotlin, there is a way to explain this sort of thing to the compiler. This is called **declaration-site variance**: we can annotate the **type-parameter** T of Source to make sure that it is only **returned** (produced) from members of Source<T>, and never consumed. To do this we provide the **out** modifier:

```
abstract class Source<out T> {
  fun nextT(): T
}

fun demo(strs: Source<String>) {
  val objects: Source<Any> = strs // This is OK, since T is an out-parameter
  // ...
}
```

The general rule is: when a type-parameter T of a class C is declared **out**, it may occur only in **out**\-position in the members of C, but in return C<Base> can safely be a supertype of C<Derived>.

In "clever words" they say that the class C is **covariant** in the parameter T, or that T is a **covariant** type-parameter. C can be thought of as being a **producer** of `T's, and NOT a **consumer** of `T's.

The **out** modifier is called a **variance annotation**, and since it is provided at the type-parameter declaration site, we talk about **declaration-site variance**. This is in contrast with Java's **use-site variance** where wildcards in the type usages make the types covariant.

In addition to **out**, Kotlin provides a complementary variance annotation: **in**. It makes a type-parameter **contravariant**: it can only be consumed and never produced. A good example of a contravariant class is Comparable:

```
abstract class Comparable<in T> {
  fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
  x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
  // Thus, we can assign x to a variable of type Comparable<Double>
  val y: Comparable<Double> = x // OK!
}
```

- Type projections
- Use-site variance: Type projections

It is very convenient to declare a type-parameter T as **out** and have no trouble with subtyping on the use site. Yes, it is, when the class in question **can** actually be restricted to only return `T's, but what if it can't? A good example of this is Array:

```
class Array<T>(val size: Int) {
   fun get(index: Int): T { /* ... */ }
   fun set(index: Int, value: T) { /* ... */ }
}
```

This class cannot be either co\- or contravariant in T. And this imposes certain inflexibilities. Consider the following function:

```
fun copy(from: Array<Any>, to: Array<Any>) {
   assert(from.size == to.size)
   for (i in from.indices)
      to[i] = from[i]
}
```

This function is supposed to copy items from one array to another. Let's try to apply it in practice:

```
val ints: Array<Int> = array(1, 2, 3)
val any = Array<Any>(3)
copy(ints, any) // Error: expects (Array<Any>, Array<Any>)
```

Here we run into the same familiar problem: Array<T> is **invariant** in T, thus neither of Array<Int> and Array<Any> is a subtype of the other. Why? Again, because copy **might** be doing bad things, i.e. it might attempt to **write**, say, a String to from, and if we actually passed an array of Int there, a ClassCastException would have been thrown sometime later.

Then, the only thing we want to ensure is that copy() does not do any bad things. We want to prohibit it from **writing** to from, and we can:

```
fun copy(from: Array<out Any>, to: Array<Any>) {
 // ...
}
```

What has happened here is called **type projection**: we said that from is not simply an array, but a restricted (**projected**) one: we can only call those methods that return the type-parameter T, in this case it means that we can only call get(). This is our approach to **use-site variance**, and corresponds to Java's Array<? extends Object>, but in a slightly simpler way.

It is also possible to create an in-projection:

```
fun fill(dest: Array<in String>, value: String) {
 // ...
}
```

Array<in String> corresponds to Java's Array<? super String>, i.e. both an array of CharSequence and

an array of `Object` can be passed to the `fill()` function.

- Star-projections

Sometimes it is necessary to work with a generic type, although nothing is known about the type-argument, it still have to be manipulated it in a safe way. The safe way here is to say that we are dealing with an **out**-projection (the object does not consume any values of unknown types), and that this projection is with the upper bound of the corresponding parameter, i.e. `out Any?` for most cases. Kotlin provides a shorthand syntax for this, that we call a **star-projection**: `Foo<*>` means `Foo<out Bar>` where `Bar` is the upper bound for `Foo's type-parameter.

Note: star-projections are very much like Java's raw types, but safe.

7.10. Generic functions

Not only classes can have type-parameters. Functions can, too. type-parameters are placed before the name of the function:

```
fun <T> singletonList(item: T): List<T> {
  // ...
}

fun <T> T.basicToString() : String {  // extension function
  // ...
}
```

If type-parameters are passed explicitly at the call site, they are specified **after** the name of the function:

```
val l = singletonList<Int>(1)
```

7.11. Generic constraints

The set of all possible types that can be substituted for a given type-parameter may be restricted by **generic constraints**.

- Upper bounds

The most common type of constraint is an **upper bound** that corresponds to Java's **extends** keyword:

```
fun <T : Comparable<T>> sort(list: List<T>) {
  // ...
}
```

The type specified after a colon is the **upper bound**: only a subtype of `Comparable<T>` may be

substituted for T. For example

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype
of Comparable<HashMap<Int, String>>
```

The default upper bound (if none specified) is Any?. Only one upper bound can be specified inside the angle brackets. If the same type-parameter needs more than one upper bound, we need a separate **where**-clause:

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T>
    where T : Comparable,
          T : Cloneable {
  return list.filter { it > threshold }.map { it.clone() }
}
```

7.11.1. Type Projections

TODO A type projection cannot be used as a supertype.

7.11.2. Extended Set of Bounds

If T is a type-parameter, then its *extended set of bounds* S is the transitive closure of its set of bounds under the following rules: * If a nullable type U? is in S, then U is in S. * If a type-parameter U is in S, then all bounds of U are in S.

[Example: Consider the following class declaration:

```
class A<T : U, U : Throwable?>
```

The extended set of bounds of the type-parameter T is { U, Throwable?, Throwable }. End Example]

It is a compile-time error if the extended set of bounds of a type-parameter contains two different instantiations of the same generic type.

7.11.3. Constituent Types

For every type T, there is a corresponding set S of its *constituent types*, defined as the transitive closure of the singleton set {T} under the following rules:

- If a nullable type U? is in S, then U is in S.

- If a constructed generic type G<A1, ⋯, An> is in S, then for each its type-argument Ai that is a type (i.e. not a projection argument), Ai is in S.

- If a constructed generic type G<A1, ⋯, An> is in S, then for each its type-argument Ai that is a projection argument of the form out Ti or in Ti, Ti is in S.

[Note: Note that every type is always a constituent type of itself. *End Note]*

[Example: The constituent types of the type `A<T?, B<out B<X, X>, C<*>>>` are { `A<T, B<out B<X, X>, C<*>>>`, `T?`, `T`, `B<out B<X, X>, C<*>>`, `B<X, X>`, `C<*>`, `X` }. End Example]

7.11.4. Effectively Generic Types

A class or interface is an *effectively generic* type if:

- its declaration has a type-parameter list, or
- it is an inner class nested within an *effectively generic* class.

An *effective type-parameter list* of an effectively generic type `A` is:

- just the type-parameter list of `A` — if A is not an inner class or its immediate container is not an effectively generic class,
- just the effective type-parameter list of the immediate container of `A` — if A is a non-generic inner class,
- the concatenation of the effective type-parameter list of the immediate container of `A` and the type-parameter list of `A` — otherwise.

In the latter case it is assumed that any identically named type-parameters declared in different type-parameter lists are still distinct and are not confused when they appear in the concatenated effective type-parameter list.

TODO: classes within generic methods?

7.12. Restrictions on Generic Types

Here, for the sake of simplicity, we assume that all type declarations are top-level, and ignore existence of nested, inner and local type declarations (including those declared within generic functions). When we refer to type-parameters, we always assume type-parameters of generic types (and never those of generic functions). An extension covering the full language complexity will be presented separately.

7.13. General Rules and Definitions

A type that is parameterized with one or more types is called a generic type. Each generic type has its declaration `C<T1, T2, ⋯>`, where `Ti` are its type-parameters (also referred to as formal type-parameters), and multiple possible instantiations `C<A1, A2, ⋯>` where `Ai` are type-arguments (also referred to as actual type-arguments). A type-argument can be either:

- a type, in which case it is called a simple type-argument, or
- have the form `out X`, `in X` or `*` (where `X` is some type), in which case it is called a projection type-argument. If an instantiation has at least one projection type-argument, it is called a projected generic type, otherwise it is called a simple generic type. A type `C<A1, A2, ⋯>?` is called a nullable generic type, and is also included as a particular case in a more general notion of

generic type. For each type-argument `Ai` of an instantiation `C<A1, A2, ⋯>` there is a corresponding type-parameter `Ti` of the declaration of `C<T1, T2, ⋯>`, having the same position in the type-parameter list as the position of `Ai` in the type-argument list `<A1, A2, ⋯>`. And, vice versa, we can say about a type-argument (of a particular instantiation of a generic type) corresponding to a given type-parameter of the declaration of that generic type.

If `T` is a type-parameter of the declaration of a generic type `C<⋯>`, then `C<⋯>` is called the owner of `T`. A type that is not generic is called a non-generic type. Formal type-parameters can be used within their scope as regular types, and are also classified as non-generic types. Note that a simple type-argument can be either a non-generic type (possibly, a type-parameter), or it can be a generic type (simple or projected). A projection type-argument that is not directly an argument of a generic type `C<⋯>`, but rather is nested somewhere within its simple type-arguments, does not make `C<⋯>` a projected generic type. In any case, the nesting depth of a generic type (either explicitly written in the program or inferred as a type of an expression in a program) is finite. Within a generic type declaration `C<T1, T2, ⋯>` it is possible to use an instantiation `C<T1, T2, ⋯>` where each type-parameter is used as a type-argument corresponding to itself. Such type is called the instance type of the corresponding declaration (it is the type of the expression `this@C`).

7.14. Constituent Types

Suppose `X` is a type. Let `SX` denote the set of types that is the transitive closure of the singleton set { `X` } under the following rules:

- If a nullable type `K?` is in `SX`, then `K` is in `SX`.

- If a constructed generic type `C<A1, A2, ⋯>` is in `SX`, then every its simple type-argument `Ai` is in `SX`.

- If a constructed generic type `C<A1, A2, ⋯>` is in `SX`, then for every its projection type-argument of the form `out Bi` or `in Bi`, the type `Bi` is in `SX`.

An element of the set `SX` is called a constituent type of `X`. Note that for every type `X` the set of its constituent types is finite and contains `X` itself. If a type `Y` is a constituent type of a type `X`, we sometimes say that `X` refers to `Y`.

Examples:

The only constituent type of a non-generic, non-nullable type `T` is `T` itself.

The constituent types of `C<X, Y?>` are `C<X, Y?>`, `X`, `Y?` and `Y`.

The constituent types of `A<T?, B<out B<X, X>, C<*>>>` are `A<T?, B<out B<X, X>, C<*>>>`, `T?`, `T`, `B<out B<X, X>, C<*>>>`, `B<X, X>`, `C<*>` and `X`.

7.15. Bounds

Every type-parameter `Ti` of a generic type `C<⋯>` has a (possibly empty) set of declared upper bounds. Each upper bound is a type (possibly generic, and possibly referring to `Ti` itself or to other type-parameters from the same type-parameter list). For a particular simple instantiation `C<A1, A2, ⋯>` we say that a type-parameter `Ti` has a set of upper bounds adapted to this instantiation, that is

obtained by substitution of type-arguments `A1`, `A2`, ... for corresponding type-parameters in the set of declared upper bounds of `Ti`.

Example:

Consider a generic interface declaration interface `C<T : C<T>>`. The type-parameter `T` has a single declared upper bound `C<T>`, and it refers to the type-parameter `T` itself. Now consider a derived interface declaration `interface B : C<B>`. Its superinterface `C<B>` is an instantiation of the generic interface `C<···>` where `B` is the type-argument corresponding to the type-parameter `T`. The set of upper bounds of the type-parameter `T` adapted to the instantiation `C<B>` has the only element `C<B>` obtained by substitution of the type-argument `B` for the type-parameter `T` in the declared upper bound `C<T>`.

7.16. Skolemization

Suppose `C<···>` is a projected generic type. A skolemization of this type is a simple generic type constructed using the following steps. Replace each projection type-argument in `C<···>` with a fresh type variable (a distinct type variable is synthesized for each type-argument being replaced). For each introduced type variable `Q`, let the set of its upper bounds be the set of upper bounds of the corresponding type-parameter adapted to this. If the variable `Q` was substituted in place of a projection type-argument of the form `out X`, add the type `X` to the set of upper bounds of `Q`. If the variable `Q` was substituted in place a projection type-argument of the form `in Y`, let the type `Y` be a lower bound of `Q`. A type variable introduced during skolemization is called a skolem type variable.

7.17. B-Closure

Suppose `S` is a set of types. Let `Z` denote the set of types that is the transitive closure of `S` under the following rules:

- If a nullable type `K?` in `Z`, then `K` is in `Z`.
- If a type-parameter `T` is in `Z`, then each declared upper bound of `T` is in `Z`.

The set `Z` is called the B-closure of `S`.

7.18. Finite Bound Restriction

Let G be a directed graph whose vertices are all type-parameters of all generic type declarations in the program. For every projection type-argument A in every generic type B<...> in the set of constituent types of every type in the B-closure of the set of declared upper bounds of every type-parameter T in G add an edge from T to U, where U is the type-parameter of the declaration of B<...> corresponding to the type-argument A. It is a compile-time error if the graph G has a cycle.

[Note: An intuitive meaning of an edge X → Y in the graph G is "the exact meaning of bounds for the type-parameter X depends on bounds for the type-parameter Y". *End Note]*

[Example:

The following declaration is invalid, because there is an edge `T` → `T`, forming a cycle:

```
interface A<T : A<*>>
```

The bound A<*> is a projection with an implicit bound. If that bound is made explicit, the type A<*> takes an equivalent form A<out A<*>>. In the same way, it can be further rewritten in an equivalent from A<out A<out A<*>>>, and so on. In its fully expanded form this bound would be infinite. The purpose of this rule is to avoid such infinite types, and type checking difficulties associated with them.

The following pair of declarations is invalid, because there are edges T → S and S → T, forming a cycle:

```
interface B<T : C<*>>
interface C<S : B<*>>
```

The following declaration is invalid, because there are edges K → V and V → K, forming a cycle:

```
interface D<K: D<K, *>, V: D<*, V>>
```

On the other hand, each of the following declarations is valid:

```
interface A<T : A<T>>
interface D<K, V : D<*, V>>
```

End Example]

TODO: Interaction of these algoritms with flexible types. TODO: Importing type declared in Java that violate these rules.

Subtyping relationships is to be decided inductively, i.e. must have a finite proof.

```
interface N<in T>
interface A<S> : N<N<A<A<S>>>>
```

8. Control and Data Flow

8.1. General

TODO

8.2. Definite Assignment

Every local variable must be definitely assigned at every point in reachable code where its value is

read.

TODO: Exact rules.

TODO: Definite assignment for properties

8.3. Smart-casts

The language uses information about preceding checks for null, checks for types (is, !is), safe call operators (?.) and Nothing-returning expression to infer additional information about types of variable (beyond that explicitly specified or inferred from initializers at their declarations) that may be more specific in certain blocks or even expressions. This information is then used to enable wider set of operations on those expressions and to select more specific overloads.

[Example:

```
fun main(args: Array<String>) {
    var x : Any
    x = ""
    x.toUpperCase() // OK, smart cast to String
}
```

End Example]

TODO: Description of the algorithm.

Example of code not handled by the algorithm (https://youtrack.jetbrains.com/issue/KT-8781):

```
fun f(x : Boolean?, y : Boolean?) {
    if (x == true) return
    if (x == false) return
    if (y != x) {
        y.hashCode()
    }
}
```

- When smart casts are enabled

- On local values (always)

- On local variables, if (all three of the following are true) a smart cast is performed not in the loop which changes the variable after the smart cast the smart cast is performed in the same function when the variable is declared, not inside some closure no closure that changes the variable exists before the location of the smart cast

- On private or internal member or top-level values, if they are not abstract / open, delegated, and have no custom getter

- On protected or public member or top-level values, if (both of the following are true) they are not abstract / open, delegated, and have no custom getter a smart cast is performed in the same

module when the value is declared

Smart casts are disabled for member or top-level variables.

II. What information is taken into account (examples)

if / while (x != null) makes x not nullable inside if / while if / while (x is Type) makes x of type Type inside if / while x!! makes x not nullable after !! x as Type makes x of type Type after as x.foo(...).bar(...) or x?.foo(...)?.bar(...) makes x not nullable inside foo / bar arguments x = y makes x of the type of y after the assignment val / var x = y makes of the type of y after the initialization, but val / var x: Type = y makes x of type Type after the initialization ...

8.4. Unreachable Code

Certain regions of code may be proved unreachable via static analysis, which results in a compile-time warning. Definite assignment is not checked within unreachable code, and assignment in unreachable code has no effect of definite assignment state of variables. Otherwise, unreachable code is not exempt from rules of this specification, and any violations result in errors in the same way as in reachable code. Compiler is free to skip code generation for any code proved to be unreachable, but this shall not have any observable effects (beyond those that can be obtained by inspection of the binaries).

TODO

8.4.1. Effects of kotlin.Nothing Type

An evaluation of expression of type kotlin.Nothing never completes normally, and any assignment to a variable of type kotlin.Nothing is unreachable.

TODO

8.4.2. Effects on Nullability

TODO

8.4.3. Effects on Smart-Casts

TODO

9. Application Life Cycle

9.1. Applications vs. Libraries

TODO

9.2. Application Startup

TODO

9.3. Entry Point

TODO

9.4. Type Loading and Initialization

TODO

9.5. Threads

TODO

9.5.1. Race Conditions

TODO

9.6. Unhandled Exceptions

TODO

9.7. Runtime Limitations

9.7.1. General

TODO

9.7.2. Stack Overflow

TODO

9.7.3. Out of Memory Condition

TODO

9.8. Application Termination

TODO

9.9. Garbage Collection

TODO

9.10. Finalization

TODO

10. Object-Oriented Programming Features

10.1. General

TODO

10.2. Inheritance

All classes in Kotlin have a common superclass `Any`, that is a default superclass for a class with no supertypes explicitly specified:

```
class Example // Implicitly inherits from Any
```

`Any` is not `java.lang.Object`; in particular, it does not have any members other than `equals()`, `hashCode()` and `toString()`.

To declare an explicit supertype, it is specified after a colon in the class header:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

If the class has a primary constructor, the base type can (and must) be initialized right there, using the parameters of the primary constructor.

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the **super** keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx) {
    }

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs) {
    }
}
```

By default, all classes in Kotlin are final. More precisely, if a class declaration has the `open` modifier, then the class is considered open, and can be used as a superclass of another class. Otherwise, if the class declaration has the `final` modifier, the class is considered final, and cannot be used as a

superclass. If neither the `open` nor `final` modifier is present, then the class is considered final. [Note: Thus, the `final` modifier is entirely optional and only serves to make the intention explicit. End note]

- Overriding Members

Kotlin requires the explicit `open` modifier for overridable members and the explicit `override` modifier for overrides:

```
open class Base {
  open fun v() {}
  fun nv() {}
}
class Derived() : Base() {
  override fun v() {}
}
```

The **override** modifier is required for `Derived.v()`. If it were missing, it would result in a compile-time error. If there is no **open** annotation on a function, like `Base.nv()`, declaring a method with the same signature in a subclass is illegal, either with **override** or without it. In a final class (e.g. a class with no **open** annotation), open members are prohibited.

A member marked **override** is itself open, i.e. it may be overridden in subclasses. To prohibit re-overriding, use **final**:

```
open class AnotherDerived() : Base() {
  final override fun v() {}
}
```

- Overriding Rules

In Kotlin, implementation inheritance is regulated by the following rule: if a class inherits many implementations of the same member from its immediate supertypes, it must override this member and provide its own implementation (perhaps, using one of the inherited ones). To denote the supertype from which the inherited implementation is taken, the 'super' keyword followed by the supertype name in angle brackets is used, e.g. `super<Base>`:

```
open class A {
  open fun f() { print("A") }
  fun a() { print("a") }
}

interface B {
  fun f() { print("B") } // interface members are 'open' by default
  fun b() { print("b") }
}

class C() : A(), B {
  // The compiler requires f() to be overridden:
  override fun f() {
    super<A>.f() // call to A.f()
    super<B>.f() // call to B.f()
  }
}
```

It is permissible to inherit from both `A` and `B`, and there are no ambiguity problems with `a()` and `b()` since `C` inherits only one implementation of each of these functions. But for `f()` two implementations are inherited by `C`, and thus `f()` has to be overridden in `C` to avoid the ambiguity.

- Abstract Classes

A class and some of its members may be declared **abstract**. An abstract member does not have an implementation in its class. Thus, when some descendant inherits an abstract member, it does not count as an implementation:

```
abstract class A {
  abstract fun f()
}

interface B {
  fun f() { print("B") }
}

class C() : A(), B {
  // It is not required to override f()
}
```

Note that it is allowed, but not required to provide the `open` modifier for an abstract class or function.

A non-abstract open member can be overridden with an abstract one.

```
open class Base {
  open fun f() {}
}

abstract class Derived : Base() {
  override abstract fun f()
}
```

10.3. Interface Implementation

TODO

10.4. Visibility

TODO

10.5. Overriding

In Kotlin, implementation inheritance is regulated by the following rule: if a class inherits many implementations of the same member from its immediate superclasses, it must override this member and provide its own implementation (perhaps, using one of the inherited ones). To denote the supertype from which the inherited implementation is taken, the 'super' keyword followed by the supertype name in angle brackets is used, e.g. super<Base>.

If a member is overridden on one inheritance path, it is considered to be overridden on all inheritance paths.

TODO

10.6. Delegation

TODO

10.7. Extension Members

If a member function in type A is an extension for type B, then in case it's invoked with an explicit receiver, the receiver must be of type B [Note: for example, such a function cannot be invoked on an explicit receiver of type A even if an implicit receiver of type B is in scope. End note]

Kotlin provides the ability to extend a class with new functionality without having to inherit from the class or use any type of design pattern such as Decorator. This is done via special declarations called *extensions*. Kotlin supports *extension functions* and *extension properties*.

- Extension Functions

To declare an extension function, its name has to be prefixed with a *receiver type*, i.e. the type being

extended. The following adds a `swap` function to `MutableList<Int>`:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

The `this` keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot or provided implicitly). Such a function can be called on any `MutableList<Int>`:

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'
```

This function makes sense for any `MutableList<T>`, and so it can be made generic:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

Type-parameters of a generic function are declared before the function name (and before the receiver type, if present). [Rationale: type-parameters may be used in the receiver type, and it is considered prudent to put the declaration before usage. It also enables better IDE support. End rationale]

- Extensions are resolved **statically**

Extensions do not modify classes they extend. By defining an extension, one do not insert new members into a class, but merely make new functions callable with the dot-notation on instances of this class.

Extension functions are dispatched **statically**, i.e. they are not virtual by receiver type. If there's a member and extension of the same type both applicable to given arguments, a member always wins. For example:

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }
```

If `c.foo()` is called on any `c` of type `C`, it will print "member", not "extension".

- Nullable Receiver

Note that extensions can be defined with a nullable receiver type. Such extensions can be called on an expression even if its value is null, and can check for `this == null` inside the body. [Example: This is what allows to call toString() in Kotlin without checking for null: the check happens inside the extension function. End example]

```kotlin
fun Any?.toString(): String {
    if (this == null) return "null"
    // after the null check, 'this' is autocast to a non-null type, so the toString()
below
    // resolves to the member function of the Any class
    return toString()
}
```

- Extension Properties

Similarly to functions, Kotlin supports extension properties:

```kotlin
val <T> List<T>.lastIndex: Int
  get() = size - 1
```

Note that, since extensions do not actually insert members into classes, there's no efficient way for an extension property to have a backing field. This is why initializers are not allowed for extension properties. Their behavior can only be defined by explicitly providing getters/setters.

Example:

```kotlin
val Foo.bar = 1 // error: initializers are not allowed for extension properties
```

- Companion Object Extensions

If a class has a companion object defined, one can also define extension functions and properties for the companion object:

```kotlin
class MyClass {
  companion object { }  // will be called "Companion"
}

fun MyClass.Companion.foo() {
  // ...
}
```

Just like regular members of the companion object, they can be called using only the class name as the qualifier:

```
MyClass.foo()
```

- Scope of Extensions

Most of the time extensions are defined on the top level, i.e. outside of any type declarations:

```
package foo.bar

fun Baz.goo() { ... }
```

To use such an extension outside its declaring package, it has to be imported at the call site:

```
package com.example.usage

import foo.bar.goo // importing all extensions by name "goo"
                   // or
import foo.bar.*   // importing everything from "foo.bar"

fun usage(baz: Baz) {
  baz.goo()
}
```

10.8. Object Construction

TODO

10.9. Runtime Virtual Invocation Dispatch

TODO

10.10. Singleton Objects

Kotlin has a predefined syntax to declaring singleton objects.

TODO

11. Functional Programming Features

11.1. General

TODO

11.2. Functional Values

TODO

11.3. Method References

TODO

11.4. Anonymous Functions

- Higher-Order Functions

A higher-order function is a function that takes functions as parameters, or returns a function. A good example of such a function is `lock()` that takes a lock object and a function, acquires the lock, runs the function and releases the lock:

```
fun lock<T>(lock: Lock, body: () -> T): T {
  lock.lock()
  try {
    return body()
  }
  finally {
    lock.unlock()
  }
}
```

Let us examine the code above: `body` has a [function type](#function-types): `() → T`, so it is supposed to be a function that takes no parameters and returns a value of type `T`. It is invoked inside the **try**-block, while protected by the `lock`, and its result is returned by the `lock()` function.

If we want to call `lock()`, we can pass another function to it as an argument:

```
fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized)
```

Another, often more convenient way is to pass a [function literal](#function-literals-and-function-expressions) (often referred to as *lambda expression*):

```
val result = lock(lock, { sharedResource.operation() })
```

Function literals are described in more [detail below](#function-literals-and-function-expressions), but to facilitate understanding of this section, here is a brief summary:

- A function literal is always surrounded by curly braces,

- Its parameters (if any) are declared before → (parameter types may be omitted),
- The body goes after → (when present).

If a function can be invoked with only one argument, and the argument is a function literal, then instead of using regular syntax f({⋯}), the explicit argument list can be omitted and the function literal specified immediately after the function name (and after the type-argument list, if present).

If a function can be invoked with an argument list, where the last argument is a function literal, the syntax f(⋯) { ⋯ } can be used, where the explicit argument list between the parentheses contains all arguments except the last one, and the last argument, which is a function literal, is specified immediately after the function name (and after the type-argument list, if present). [Note: The explicit argument list may have zero arguments in this case. End note]

```
lock (lock) {
    sharedResource.operation()
}
```

Another example of a higher-order function would be map():

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

This function can be called as follows:

```
val doubled = ints.map { it -> it * 2 }
```

Another helpful convention is that if a function literal has only one parameter, its declaration may be omitted (along with the →), and it will be implicitly named it:

```
ints.map { it * 2 }
```

These conventions allow to write code like this:

```
strings.filter { it.length == 5 }.sortBy { it }.map { it.toUpperCase() }
```

- Inline Functions

Sometimes it is beneficial to enhance performance of higher-order functions using inline functions.

- Function Literals and Function Expressions

A function literal or a function expression is an "anonymous function", i.e. a function that is not declared, but passed immediately as an expression. Consider the following example:

```
max(strings, { a, b -> a.length() < b.length() })
```

Function `max` is a higher-order function, i.e. it takes a function value as the second argument. This second argument is an expression that is itself a function, i.e. a function literal. As a function, it is equivalent to

```
fun compare(a: String, b: String): Boolean = a.length() < b.length()
```

- Function Types

For a function to accept another function as a parameter, the parameter can be declared to be of a function type. For example the aforementioned function `max` is defined as follows:

```
fun max<T>(collection: Collection<out T>, less: (T, T) -> Boolean): T? {
  var max: T? = null
  for (it in collection)
    if (max == null || less(max!!, it))
      max = it
  return max
}
```

The parameter `less` is of type `(T, T) → Boolean`, i.e. a function that takes two parameters of type `T` and returns a `Boolean`: true if the first one is smaller than the second one.

In the body, line 4, `less` is used as a function: it is called by passing two arguments of type `T`.

A function type is written as above, or may have named parameters, for documentation purposes and to enable calls with named arguments.

```
val compare: (x: T, y: T) -> Int = ...
```

- Function Literal Syntax

The full syntactic form of function literals, i.e. literals of function types, is as follows:

```
val sum = { x: Int, y: Int -> x + y }
```

A function literal is always surrounded by curly braces, parameter declarations in the full syntactic form go inside parentheses and have optional type annotations, the body goes after an → sign. If all the optional annotations are left out, then the function literal looks like this:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

It is very common that a function literal has only one parameter. If the signature can be inferred from the context, it is permitted not to declare the single parameter, and it will be implicitly declared and will have the name `it`:

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

- Function Expressions

One thing missing from the function literal syntax presented above is the ability to specify the return type of the function. In most cases, this is unnecessary because the return type can be inferred automatically. To specify it explicitly, an alternative syntax can be used: a *function expression*.

```
fun(x: Int, y: Int): Int = x + y
```

A function expression looks very much like a regular function declaration, except that its name is omitted. Its body can be either an expression (as shown above) or a block:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

The parameters and the return type are specified in the same way as for regular functions, except that the parameter types can be omitted if they can be inferred from context:

```
ints.filter(fun(item) = item > 0)
```

The return type inference for function expressions works just like for normal functions: the return type is inferred automatically for function expressions with an expression body and has to be specified explicitly (or is assumed to be `Unit`) for function expressions with a block body.

Note that function expression parameters are always passed inside the parentheses. The shorthand syntax allowing to leave the function outside the parentheses works only for function literals.

One other difference between function literals and function expressions is the behavior of non-local returns. A **return** statement without a label always returns from the function declared with the **fun** keyword. This means that a **return** inside a function literal will return from the enclosing function, whereas a **return** inside a function expression will return from the function expression itself.

- Closures

A function literal or expression (as well as a local function and an object expression) can access its *closure*, i.e. the variables declared in the outer scope. Unlike Java, the variables captured in the closure can be modified:

```
var sum = 0
ints.filter { it > 0 }.forEach {
  sum += it
}
print(sum)
```

- Extension Function Expressions

TODO: Move to informal introduction In addition to ordinary functions, Kotlin supports extension functions. Extension function literals and expressions are also supported.

A function expression can have a receiver type specification, in which case it is called an extension function expression. Extension function expression has an additional implicit parameter of the specified receiver type, which has no name but is available within the function body through the `this` keyword.

```
val sum = fun Int.(other: Int): Int = this + other
```

Receiver type may be specified explicitly only in function expressions, not in function literals. Function literals can be used in context where an expression of an extension function type is expected, provided that the receiver type can be inferred from the context.

The type of an extension function expression is a function type with receiver:

```
sum : Int.(other: Int) -> Int
```

The function can be called as if it were a method on the receiver object:

```
1.sum(2)
```

11.4.1. Lambda as an Argument to an Invocation

TODO: Special syntax

11.5. Function Inlining

TODO

11.5.1. inline Functions

An inline function is declared with the annotation inline. Any of type-parameters of an inline

functions can be made reified by declaring them with reified modifier. A virtual function (i.e. non-private, non-final method) cannot be declared inline. An inline function cannot be directly recursive (it also cannot invoke itself inside nested lambdas, and cannot have a callable reference to itself). An inline function cannot be indirectly recursive if at least one recursion cycle contains only inline functions.

The following declarations and expressions are not supported anywhere inside inline functions (including any nested object expressions):

- declarations of local functions
- declarations of local classes
- declarations of inner nested classes
- function expressions (starting with fun keyword)
- default values for optional parameters

A local function cannot be declared inline.

If an `inline` function has a parameter of a functional type without `noinline` annotation, and an argument corresponding to this parameter is an anonymous function (function expression or function literal), then this anonymous function is inlined at each of its usages in the body of the inline function (which is itself is inlined at its call site). The function literal is allowed to have non-local return statements in such cases. There are also certain restrictions on use of such parameters: they cannot be assigned to variables, fields, properties, or array elements or passed as arguments to non-inline functions (or arguments to inline functions that are not inlined, e.g. have non-functional declared type, vararg modifier, or noinline annotation). Unless they are annotated with the annotation `crossinline`, they cannot be used within function literals or object expressions.

A parameters of an `inline` function whose type is a nullable functional types must have `noinline` or `vararg` annotation.

Using higher-order functions brings certain runtime overhead: each function is an object, and it captures a closure, i.e. those variables that are accessed in the body of the function. Memory allocations (both for function objects and classes) and virtual calls introduce runtime overhead.

But in many cases this kind of overhead can be eliminated by inlining the function literals. The functions shown above are good examples of this situation. I.e., the `lock()` function could be easily inlined at call-sites. Consider the following case:

```
lock(l) { foo() }
```

Instead of creating a function object for the parameter and generating a call, the compiler could emit the following code:

```
lock.lock()
try {
  foo()
}
finally {
  lock.unlock()
}
```

To make the compiler do this, the `lock()` function has to be annotated with the `inline` modifier:

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
  // ...
}
```

The `inline` modifier affects both the function itself and the lambdas passed to it: all of those will be inlined into the call site.

- noinline

In case it is intended that only some of the lambdas passed to an inline function are to be inlined, some of the function parameters can be marked with the `noinline` modifier:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
  // ...
}
```

Inlinable lambdas can only be called inside the inline functions or passed as inlinable arguments, but `noinline` ones can be manipulated as any other values: stored in fields, passed around etc.

Note that if an inline function has no inlinable function parameters and no reified type-parameters, the compiler will issue a warning. [Rationale: inlining such functions is very unlikely to be beneficial. End Rationale]

- Non-local returns

To exit a named function or a function expression, the simple unqualified form of the `return` expression is used. But to exit a lambda, the `return` expression with a label has to be used, and an unqualified `return` is forbidden inside a lambda. [Rationale: a lambda can not make the enclosing function return. End rationale]

```
fun foo() {
  ordinaryFunction {
    return // ERROR: can not make `foo` return here
  }
}
```

But if the function the lambda is passed to is inlined, the return can be inlined as well, so it is allowed:

```
fun foo() {
  inlineFunction {
    return // OK: the lambda is inlined
  }
}
```

Such returns (located in a lambda, but exiting the enclosing function) are called **non-local** returns. This sort of constructs are often used in loops, which inline functions often enclose:

```
fun hasZeros(ints: List<Int>): Boolean {
  ints.forEach {
    if (it == 0) return true // returns from hasZeros
  }
  return false
}
```

Note that some inline functions may call the lambdas passed to them as parameters not directly from the function body, but from another execution context, such as a local object or a nested function. In such cases, non-local control flow is also not allowed in the lambdas. To indicate that, the lambda parameter needs to be marked with the `crossinline` modifier:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

- Reified type-parameters

Sometimes it is necessary to access a type passed as a parameter:

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p?.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T
}
```

Here, the function walks up a tree and uses reflection to check if a node has a certain type. Unfortunately, the call site looks overly verbose:

```
myTree.findParentOfType(MyTreeNodeType::class.java)
```

It would be more readable if the type could be provided as a type-argument:

```
myTree.findParentOfType<MyTreeNodeType>()
```

To enable this, inline functions support **reified type-parameters**. Using them, the method can be rewritten as follows:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p?.parent
    }
    return p as T
}
```

Here, the type-parameter is annotated with the `reified` modifier, that makes it accessible inside the function much like a regular class. Since the function is inlined, no reflection is necessary, and operators like `!is` and `as` can be used instead. The call site now takes the desired form: `myTree.findParentOfType<MyTreeNodeType>()`.

Though reflection may not be needed in many cases, it still can be used with a reified type-parameter:

```
inline fun membersOf<reified T>() = T::class.members

fun main(s: Array<String>) {
  println(membersOf<StringBuilder>().joinToString("\n"))
}
```

Functions not marked as `inline` can not have reified parameters. A type that does not have a run-time representation (e.g. a non-reified type-parameter or a fictitious type like `Nothing`) can not be used as an argument for a reified type-parameter.

11.6. Closures

TODO

11.6.1. Instantiation and Lifetime of Captured Variables

TODO

11.6.2. Implications for Strong References and Garbage Collection

TODO

11.6.3. external Modifier

`external` modifier is only applicable to top-level functions, or to class members (not to interface members). Top-level function mush have a body unless it is external. External function cannot be inline. External function cannot have a body. If function is not external and not abstract, it must have a body. A constructor cannot be external.

TODO: external accessors?

11.6.4. tailrec Modifier

`tailrec` modifier on a function indicates that that a function is tail-recursive and its recursive invocations must be optimized by the compiler to loops. If the `tailrec` modifier is applied to a function that is not actually tail-recursive, a compile-time error occurs.

12. Compilation Units

12.1. General

TODO

12.2. File-Level Annotations

TODO

12.3. Package Specification

If a package specification is present, it shall be at the top of the source file. It is not required to match directories and packages: source files can be placed arbitrarily in the file system. All the contents (such as classes and functions) of the source file are contained by the package declared. If the package is not specified, the contents of such a file belong to the default package that has no name.

TODO

12.4. Import Directives

Apart from the default imports declared by the module, each file may contain its own import directives.

An import directive must refer to a fully qualified name. It is an error if two import directives attempt to import types with the same simple name. If several import directives import the same type multiple times, redundant imports are ignored and the result is the same as if the type was

imported only once.

It is an error if two import directives import different symbols but attempt to rename them to the same name. If several import directives import the same symbol multiple times and rename it to the same name N, redundant imports are ignored and the result is the same as if the type was imported by the name N only once.

12.5. Type and Object Declarations

TODO

12.6. Package-Level Functions

Function visible from outside of a module shall have return type explicitly specified. Unit return type can be omitted.

TODO

13. Classes

13.1. General

TODO: more precise class definition.

Class declarations and interface declarations are similar in the sense that they introduce reference types, provide templates partially or completely describing shape and behavior of objects of those types, and define their substyping relations. But only class declarations can describe state of objects and their initialization logic. And only class declarations can introduce concrete types, while interface declarations always introduce abstract types. Only single inheritance is supported for classes, while interfaces support multiple inheritance. A class can be a subtype both of its superclass, and of multiple superinterfaces, while an interterface can only be a subtype of its superinterfaces and cannot specify its superclass (although all interfaces are implicitly subtypes of the ultimate superclass `Any`). This section presents specification of class types and class declarations.

A class can encapsulate state, object initialization logic, public functional contract, possibly providing a complete or partial implementation of it and an implementation of private functional members it may have. A class can be introduced via a class declaration, object declaration or anonymous object expression. Classes introduced via class or object declarations are named (their names always occur explicitly in their declarations, except for companion objects that are allowed to omit it and use the default name `Companion`). Classes introduced via anonymous object expressions do not have a denotable name, but still behave as nominal (not structural) types.

13.2. Class Declarations

A class declaration can appear at the top level,

A body of a class, interface or object declaration is optional. If no explicit body is provided, then an

empty body {} is assumed.

TODO

13.3. Class Modifiers

A class can have zero or more of the following modifiers: visibility modifiers and `abstract`, `enum`, `final`, `inner`, `open`, `sealed`.

An order of modifiers is not significant. The same modifier cannot appear more than once in the same modifier list. Not all combinations of modifiers are valid and not all modifiers are allowed on all class declarations.

A modifier list can contain no more than one modifier from the set: public private protected internal. Only member classes can have protected modifier.

The following pairs of modifiers are incompatible: final open, final abstract, final sealed, sealed open.

Modifier open is redundant if abstract is specified. Modifier abstract is redundant if sealed is specified.

TODO: factor out the common rules for modifiers and visibility modifiers into a separate section.

13.4. Initialization Blocks

Initialization block appears within a class body and must be immediately preceded by the `init` contextual keyword. Initialization block is not a declaration and so does not introduce a symbol. It is evaluated during the primary constructor invocation. A class can have multiple initialization blocks, they are evaluated in their textual order. Primary constructor parameters are in scope throughout all initialization blocks of the class. Variables declared in one initialization block are not in scope in other initialization blocks of the same class.

Initialization blocks can also appear in object declarations and object expressions. They are evaluated during the initialization of the corresponding object in their textual order.

Annotations on init blocks?

TODO

13.5. Class Members

Non-abstract classes cannot declare abstract functional members (but can have abstract member classes), and must override all inherited abstract functional members. TODO

13.6. Constructors

A class in Kotlin can have a primary constructor and one or more secondary constructors. The primary constructor is declared in the class header: it goes after the class name (and optional type-

parameters). Keywords val and var together with accessibility modifiers can be applied to primary constructor parameters to implicitly declare and initialize properties with the same names. They also can have override modifier to override properties from a supertype. The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks, which are prefixed with the `init`. Note that parameters of the primary constructor can be used in the initializer blocks. They can also be used in property initializers declared in the class body.

The class can also declare secondary constructors, which are prefixed with constructor. If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the this keyword.

If a non-abstract class does not declare any constructors (primary or secondary), it will have a default primary constructor with no arguments. The visibility of the constructor will be public. In order to avoid creating a default constructor, an empty private constructor can be declared.

If the class has a primary constructor, the base type can (and must) be initialized right there, possibly using the parameters of the primary constructor.

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the super keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type.

It is an error to access this of the object being created in a constructor delegation. (TODO: access to members or super access).

A class cannot have two or more constructors with the same signature. The @platformName annotation is not applicable to constructors.

A constructor cannot have tailrec modifier.

Constructor cannot have type-parameters, and there is no way to provide type-arguments to the constructor itself at a constructor invocation.

Cycles in constructor delegation are not allowed.

Constructor body is optional, if no explicit body is provided then an empty body { } is assumed. Constructors cannot have an expression body.

A constructor can have only visibility modifiers.

A constructor declaration in class C introduces a method with name C in the same declaration space where the class C is declared. Such method can overload other methods in the same declaration space (possibly including other constructors) subject to normal overloading rules. Such methods are not considered when methods overriding methods from supertypes are being determined.

this of the current class is not available in a superclass constructor invocation.

13.6.1. Secondary constructors

- Examples

With a primary constructor:

```
class Foo(a: Bar): MySuper() {
  // when there's a primary constructor, (direct or indirect) delegation to it is
required
  constructor() : this(Bar()) { ... } // can't call super() here
  constructor(s: String) : this() { ... }
}
```

No primary constructor:

```
class Foo: MySuper { // initialization of superclass is not allowed
  constructor(a: Int) : super(a + 1) { ... } // must call super() here
}
```

No primary constructor + two overloaded constructors

```
class Foo: MySuper { // initialization of superclass is not allowed
  constructor(a: Int) : super(a + 1) { ... }
  constructor() : this(1) { ... } // either super() or delegate to another constructor
}
```

- TODO

  - ☐ is delegation allowed when no primary constructor is present?

  - ☐ Allow omitting parameterless delegating calls?

- Syntax for primary constructor

  - There's a primary constructor if

  - parentheses after class name, or

  - there're no secondary constructors (default primary constructor)

  - No parentheses after name and an explicit constructor present ⇒ no primary constructor

No primary constructor ⇒ no supertype initialization allowed in the class header:

```
class Foo : Bar() { // Error
  constructor(x: Int) : this() {}
}
```

When a primary constructor is present, explicit constructors are called **secondary**.

Every class **must** have a constructor. the following is an error:

```
class Parent
class Child : Parent { }
```

The error is: "superclass must be initialized". This class has a primary constructor, but does not initialize its superclass in the class header.

- Syntax for explicit constructors

```
constructor
  : modifiers "constructor" valueParameters (":" constructorDelegationCall) block
  ;

constructorDelegationCall
  : "this" valueArguments
  | "super" valueArguments
  ;
```

Passing lambdas outside parentheses is not allowed in `constructorDelegationCall`.

- Rules for delegating calls

The only situation when an explicit constructor may not have an explicit delegating call is - when there's no primary constructor **and** the superclass has a constructor that can be called with no parameters passed to it.

```
class Parent {}
class Child: Parent {
  constructor() { ... } // implicitly calls `super()`
}
```

If there's a primary constructor, all explicit constructors must have explicit delegating calls that (directly or indirectly) call the primary constructor.

```
class Parent {}
class Child(): Parent() {
  constructor(a: Int) : this() { ... }
}
```

- Initialization code outside constructors

The primary constructor's body consists of - super class intialization from class header - assignments to properties from constructor parameters declared with `val` or `var` - property initializers and bodies of anonymous initializers following in the order of appearence in the class body

If the primary constructor is not present, property initializers and anonymous initializers are

conceptually "prepended" to the body of each explicit constructor that has a delegating call to super class, and their contents are checked accordingly for definite initialization of properties etc.

- Syntax for anonymous initializers

Anonymous initializer in the class body must be prefixed with the `init` keyword, without parentheses:

```
class C {
  init {
    ... // anonymous initializer
  }
}
```

- Checks for constructors

All constructors must be checked for

- absence of circular delegation
- overload compatibility
- definite initialization of all properties that must be initialized
- absence of non-empty super call for enum constructors

No secondary constructors can be declared for

- interfaces
- objects (named, anonymous, and default)
- bodies of enum literals

A data class shall have a primary constructor.

TODO

13.7. Methods

Functions that declared as members of a type are called methods.

TODO

13.8. Properties

A property is a functional member that is syntactically accessed like a variable, but can be implemented either a simple storage location, or using custom code that is executed when the property is read on written. A property can be read-only or read-write. A read-only property has a single accessor called getter, and a read-write property has a pair of accessors called getter and setter. An accessor is a function that is invoked when the corresponding property is accessed: a getter is invoked on property read, and a setter is invoked or property write.

A property can have an underlying storage location called a backing field. Within accessor bodies, the backing field is available as a variable named `field`. The identifier field is not a reserved keyword, and has a special meaning only within property accessors, and even there can be shadowed according to regular rules. No code outside of a property accessors can name or otherwise access the backing field of the property. The type of the backing field is the same as the type of the property.

The backing field exists if at least one accessor is default (non-abstract, non-external and does not have a body) or at least one accessor refers to the backing field using `field` identifier.

13.8.1. Delegated Properties

There are certain common kinds of properties, that without the language support would be necessary to re-implement each time they are needed. Examples include

- lazy properties: the value gets computed only upon first access,

- observable properties: listeners get notified about changes to this property,

- storing properties in a map, not in separate field each.

To cover these (and other) cases, Kotlin supports *delegated properties*:

```
class Example {
  var p: String by Delegate()
}
```

The syntax is: `val/var <property name>: <Type> by <expression>`. The expression after **by**{:.keyword} is the *delegate*, because `get()` (and `set()`) corresponding to the property will be delegated to its `getValue()` and `setValue()` methods. Property delegates don't have to implement any interface, but they have to provide a `getValue()` function (and `setValue()` --- for **var**{:.keyword}'s). For example:

```
class Delegate {
  operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
    return "$thisRef, thank you for delegating '${property.name}' to me!"
  }

  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
    println("$value has been assigned to '${property.name} in $thisRef.'")
  }
}
```

When the code reads from `p` that delegates to an instance of `Delegate`, the `getValue()` function from `Delegate` is called, so that its first parameter is the object `p` is read from and the second parameter holds a description of `p` itself (e.g. it is possible to take its name). For example:

```
val e = Example()
println(e.p)
```

This prints

```
Example@33a17727, thank you for delegating 'p' to me!
```

Similarly, when a value is assigned to p, the setValue() function is called. The first two parameters are the same, and the third holds the value being assigned:

```
e.p = "NEW"
```

This prints

```
NEW has been assigned to 'p' in Example@33a17727.
```

- Property Delegate Requirements

Here is a summary of requirements to delegate objects.

For a **read-only** property (i.e. a **val**{:.keyword}), a delegate has to provide a function named getValue that takes the following parameters:

- receiver --- must be the same or a supertype of the *property owner* (for extension properties --- the type being extended),
- metadata --- must be of type KProperty<*> or its supertype,

this function must return the same type as property (or its subtype).

For a **mutable** property (a **var**{:.keyword}), a delegate has to *additionally* provide a function named setValue that takes the following parameters:

- receiver --- same as for getValue(),
- metadata --- same as for getValue(),
- new value --- must be of the same type as a property or its supertype.

getValue() and/or setValue() functions may be provided either as member functions of the delegate class or extension functions. The latter is handy when one need to delegate property to an object which doesn't originally provide these functions. Both of the functions need to be marked with the operator keyword.

- Standard Delegates

The Kotlin standard library provides factory methods for several useful kinds of delegates.

- Lazy

`lazy()` is a function that takes a lambda and returns an instance of `Lazy<T>` which can serve as a delegate for implementing a lazy property: the first call to `get()` executes the lambda passed to `lazy()` and remembers the result, subsequent calls to `get()` simply return the remembered result.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

By default, the evaluation of lazy properties is **synchronized**: the value is computed only in one thread, and all threads will observe the same value. If the synchronization of initialization delegate is not required, so that multiple threads can execute it simultaneously, pass `LazyThreadSafetyMode.PUBLICATION` as a parameter to the `lazy()` function. And if it is known for certain that the initialization will always happen on a single thread, one can use `LazyThreadSafetyMode.NONE` mode, which doesn't incur any thread-safety guaratees and the related overhead.

- Observable

`Delegates.observable()` takes two arguments: the initial value and a handler for modifications. The handler gets called every time a value is assigned to the property (*after* the assignment has been performed). It has three parameters: a property being assigned to, the old value and the new one:

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

This example prints

```
<no name> -> first
first -> second
```

If one want to be able to intercept an assignment and "veto" it, use `vetoable()` instead of `observable()`. The handler passed to the `vetoable` is called *before* the assignment of a new property value has been performed.

- Storing Properties in a Map

One common use case is storing the values of properties in a map. [Example: This comes up often in applications like parsing JSON or doing other "dynamic" things. End example] In this case, one can use the map instance itself as the delegate for a delegated property. In order for this to work, one needs to import an extension accessor function `getValue()` that adapts maps to the delegated property API: it reads property values from the map, using property name as a key.

```
import kotlin.properties.getValue

class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

In this example, the constructor takes a map:

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

Delegated properties take values from this map (by the string keys -— names of properties):

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

This works also for **var**{:.keyword}'s properties if a `MutableMap` is used instead of read-only `Map` and an additional extension function is imported: `kotlin.properties.setValue`

```
import kotlin.properties.getValue
import kotlin.properties.setValue

class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

13.8.2. General

Classes in Kotlin can have properties. These can be declared as mutable, using the var keyword or read-only using the val keyword.

13.8.3. Properties without Explicit Accessors

Non-abstract properties must have an initializer.

TODO

13.8.4. Properties with Explicit Accessors

The full syntax for declaring a property is

```
var <propertyName>: <PropertyType> [= <property_initializer>]
   <getter>
   <setter>
```

The initializer, getter and setter are optional. Property type is optional if it can be inferred from the initializer or from the base class member being overridden. Types are not inferred for properties exposed in the public API, i.e. public and protected. If one need to change the visibility of an accessor or to annotate it, but don't need to change the default implementation, it is possible define the accessor without defining its body.

TODO

13.9. Fields

Classes cannot declare fields explicitly. But there is an implicitly declared field for every non-abstract property. It can be accessed using the $ symbol followed by the property name (it's a single token, so no whitespace is allowed in between). A field has private-to-this accessibility, and can only be accessed from inside the class where the corresponding property is defined.

TODO

13.10. Nested and Inner Classes

Inner classes are nested classes declared using inner modifier. Inner classes cannot be declared within interfaces or non-inner nested classes. Inner classes may not contain nested interface declarations or non-inner nested class declarations. Inner classes have access to current instances of their enclosing classes.

TODO

13.11. Sealed Classes

Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type. They are, in a sense, an extension of enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances which can contain state.

To declare a sealed class, one puts the `sealed` modifier before the name of the class. A sealed class can have subclasses, but all of them must be nested inside the declaration of the sealed class itself.

```
sealed class Expr {
    class Const(val number: Double) : Expr()
    class Sum(val e1: Expr, val e2: Expr) : Expr()
    object NotANumber : Expr()
}
```

Note that classes which extend subclasses of a sealed class (indirect inheritors) can be placed anywhere, not necessarily inside the declaration of the sealed class.

The key benefit of using sealed classes comes into play when they are used in a `when` expression. If it is possible to verify that the statement covers all cases, it is not necessary to add an `else` clause to the statement.

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because all cases are already covered
}
```

13.12. Local Classes

A local class is a class declared within a block. A local class cannot have inner modifier. Local classes may not contain nested interface declarations or non-inner nested class declarations. Local classes have access to current instances of their enclosing classes. If a local class is declared within a generic method, then the method's type-parameters are in scope in the local class. A local class is only in scope in its containing block from the beginning of its declaration to the end of the containing block (so, it's in scope throughout its own declaration, and not in scope before its declaration). Local classes has access to outer local variables and parameters.

TODO: specify capture semantics, compare with lambdas

13.13. Anonymous Classes

Anonymous classes are implicitly declared using anonymous class creation expressions (§18.3.10).

The body of an anonymous class is the block of the corresponding anonymous class creation expression.

TODO: Choose what do describe here and what is in the expressions' section.

13.14. Enum Classes

An enum class is declared by specifying enum modifier on a class declaration. A body of enum classes have a syntactic shape (TODO: reference to grammar production) different from regular classes, that is described below. The enum modifier is valid only on class declarations. An enum class cannot be an inner or local class. It cannot be a data class. It cannot have open modifier and other classes cannot inherit from it, except classes corresponding to entries of the same enum class, that always implicitly inherit from it. It cannot have abstract modifier, and is always implicitly abstract. It cannot have annotation or sealed modifier. An enum class declaration cannot have type-parameters (and because it cannot be inner or local, it does not have any type-parameters in scope at all). An enum class declaration cannot specify an explicit superclass, but it can specify superinterfaces. The direct supertype of an enum class E is assumed to be `Enum<E>`. *[Note:* As a consequence of the single instantiation inheritance rule (§TODO), if an enum class type `E` explicitly implements an instantiation of interface `Comparable<T>`, it must be `Comparable<E>`, because this instantiation is implemented by its supertype `Enum<E>`. *End Note]* An enum class can have a companion object.

The only instances of an enum type are the enum entries declared in it. It is not possible to explicitly invoke an enum constructor or create its instance in any other way. All enum entries are distinct objects. A reference equality comparison of an enum entry with an object return true iff the object is that enum entry itself. A constructor of an enum type cannot be invoked explicitly, it can only be implicitly referenced from the declaration of an entry of the same enum. The argument list can be omitted if empty. Trailing lambda, if any, must be provided as a regular explicit argument.

Each enum entry must have a unique name within its declaring enum class. An enum entry cannot have visibility modifiers and is implicitly public. Each enum entry name occupies a slot in the enum member declaration space. A secondary constructor cannot delegate to a constructor of the superclass. An enum entry cannot have type-parameters and never has any other type-parameters in scope.

An enum entry cannot have modifiers.

An enum entry cannot have a companion object. An enum entry can have init blocks. An enum entry cannot have constructors. It cannot specify superinterfaces. Enum classes cannot have abstract modifier, but are implicitly abstract. If an enum class contains at least one abstract functional member, then each its entry must provide implementation for all abstract functional members.

A class nested within an enum entry cannot be an inner class (BUG: KT-9750).

Synthetic method valueOf(value: String) and synthetic property `values` and special overload resolution rules for them (including possible conflicts with identically named functions of a companion object).

TODO: this and super in enum entries.

Enum classes in Kotlin have synthetic methods allowing to list the defined enum constants and to get an enum constant by its name. The signatures of these methods are as follows (assuming the name of the enum class is `EnumClass`):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

[Rationale: The method `values` is declared as a method rather than a property, because such a property could potentially conflict with an enum entry named `values`. *End Rationale]*

The `valueOf()` method throws an `IllegalArgumentException` if the specified name does not match any of the enum constants defined in the class.

Every invocation of the method `values()` returns a new instance of an array filled with enum entries, so a modification made in the result of one invocation does not affect elements in the result of another invocation.

The method `equals()` of enums performs reference equality comparison, and the method `hashCode()` is implemented consistently with it.

Every enum constant has properties to obtain its name and position in the enum class declaration:

```
val name: String
val ordinal: Int
```

The enum constants also implement the `Comparable` interface, with the natural order being the order in which they are defined in the enum class.

Enum classes can be nested.

13.15. Data Classes

A data class is a class declared with a `data` modifier. A data class is always final. It is permitted, but not required to specify a `final` modifier on its declaration. A data class cannot specify its superclass, but can specify zero or more superinterfaces. The direct superclass of a data class is always Any. A data class cannot be inner class, but can be local. A data class can contain inner classes.

A data class must have a (TODO: public?) primary constructor that has at least one parameter. All primary constructor parameters must be val/var. A data class cannot have `enum`, `annotation`, `abstract`, `open`, `sealed` modifiers.

A data class provides default implementation for equals, hashCode and toString method. If a declaration of any of these methods is provided explicitly, then it replaces the corresponding default implementation.

The default implementation for equals method performs first performs reference equality check

for `this` and its arguments, and if this check returns true, then the equals method returns true. Otherwise, it performs component-wise comparison using == operator in the same order in which the components are declared. If any of the comparisons returns false, the method immediately returns false. Otherwise, if all comparisons return true, then the method returns true. [Note: It is possible that this implementation goes into an infinite recursion if a component of a data class is (or refers to) the current instance. End Note] Components that are arrays are compared like all other types using array's equals method (that performs reference equality comparison). [Note: If structural comparison is desired then it is recommended to manually implement equals method using java.util.Arrays.equals or java.util.Arrays.deepEquals and provide a matching implementation of hashCode(). End Note]

It is often necessary to create classes that do nothing but hold data. In such classes some functionality is often mechanically derivable from the data they hold. In Kotlin a class can be marked as `data`:

```
data class User(val name: String, val age: Int)
```

This is called a *data class*. The compiler automatically derives the following members from all properties declared in the primary constructor:

- `equals()`/`hashCode()` pair,
- `toString()` of the form `"User(name=John, age=42)"`,
- `componentN()` functions corresponding to the properties in their order or declaration,
- `copy()` function (described below).

If any of these functions is explicitly defined in the class body or inherited from the base types, it will not be generated.

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfil the following requirements:

- The primary constructor needs to have at least one parameter;
- All primary constructor parameters need to be marked as `val` or `var`;
- Data classes cannot be abstract, open, sealed or inner;
- Data classes may not extend other classes (but may implement interfaces).

On the JVM, if the generated class needs to have a parameterless constructor, default values for all properties have to be specified

```
data class User(val name: String = "", val age: Int = 0)
```

- Copying

It is often necessary to create a copy of an object with *some* of its properties changed, but keeping the rest unchanged. This is what `copy()` function is generated for. For the `User` class above, its

implementation would be as follows:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

This allows us to write

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

- Data Classes and Multi-Declarations

Component functions generated for data classes enable their use in multi-declarations:

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age"
```

- Standard Data Classes

The standard library provides `Pair` and `Triple`. In most cases, though, named data classes are a better design choice, because they make the code more readable by providing meaningful names for properties.

14. Interfaces

14.1. General

An interface represents a contract than can be implemented by multiple classes, and a class can implement multiple interfaces. An interface can also specify zero or more superinterfaces. Thus, interfaces provide a restricted mechanism of multiple inheritance. Interfaces cannot contain any data or initialization logic, they can only declare abstract function members and optionally provide a default implementation for some of them. An interface cannot be instantiated directly, only a class implementing this interface can be instantiated, and an instance of such a class is also considered an instance of the interface.

An interface cannot explicitly declare a superclass. But because an instance of an interface can only exist as an instance of some class that implements it, and every class has the class kotlin.Any as its ultimate superclass, every interface is considered to have `kotlin.Any` as its only superclass, and consequently, it is assignable to that class and inherits its members (unless they are overridden by a superinterface of this interface).

An interface method without a body can be marked `abstract`, but this is redundant.

14.2. Interface Declarations

An interface is declared using interface declaration that can be recognized by presence of `interface` keyword, followed by the interface simple name. An interface name is an identifier. The fully-qualified interface is obtained by appending a dot token followed by the simple name to the fully qualified name of the interface's contained. Each interface has exactly one corresponding interface declaration. It is an error if the program contains two or more interface declarations that attempt to declare interfaces with the same fully qualified name.

TODO: Conflict between declared and imported types.

An interface declaration cannot have an initialization block or a constructor declaration.

Interfaces can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties but these need to be abstract.

An interface is defined using the keyword **interface**

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

- Implementing Interfaces

A class or object can implement one or more interfaces

```
class Child : MyInterface {
    fun bar() {
        // body
    }
}
```

- Properties in Interfaces

Interfaces allow properties as long as these are stateless, that is because interfaces do not allow state.

```
interface MyInterface {
    val property: Int // abstract

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

- Resolving overriding conflicts

If many types are specified in the supertype list, it may happen that the class inherits more than one implementation of the same method. For example,

```
interface A {
  fun foo() { print("A") }
  fun bar()
}

interface B {
  fun foo() { print("B") }
  fun bar() { print("bar") }
}

class C : A {
  override fun bar() { print("bar") }
}

class D : A, B {
  override fun foo() {
    super<A>.foo()
    super<B>.foo()
  }
}
```

Interfaces A and B both declare functions foo() and bar(). Both of them implement foo(), but only B implements bar() (bar() is not marked abstract in A, because this is the default for interfaces, if the function has no body). Now, if we derive a concrete class C from A, we, obviously, have to override bar() and provide an implementation. And if we derive D from A and B, we don't have to override bar(), because we have inherited only one implementation of it. But we have inherited two implementations of foo(), so the compiler does not know which one to choose, and forces us to override foo() and say what we want explicitly.

14.2.1. Methods

TODO

14.2.2. Properties

Properties in interfaces cannot have state, and never have an associated implicitly declared field.

TODO

14.3. Interfaces vs. Classes

TODO

15. Annotations

15.1. General

TODO

15.2. Declarations

An annotation class is declared by providing the `annotation` modifier in a class declaration. It is an error if an annotation class is generic. An annotation class declaration cannot have a body (and so, cannot have a companion object, nested or inner classes or any members other than inherited or introduced in its primary constructor). An annotation class cannot specify its supertypes explicitly. The immediate superclass of every annotation class is `kotlin.Annotation`. An annotation class cannot be an inner class, but can be a local class (TODO: bug?). If no primary constructor is specified, then a primary constructor with an empty parameter list is assumed.

An annotation class is implicitly final. An annotation class cannot have `final`, `open`, `abstract`, `sealed`, `enum` or `data` modifiers. TODO

15.2.1. Primary Constructor Parameters

Parameters of the primary constructor in an annotation class must be declared with `val` or `var` keyword. A type of a parameter cannot be nullable type. A type of a parameter must be one of the following:

- Primitive type: `Byte`, `Short`, `Int`, `Long`, `Char`, `Boolean`, `Float`, `Double`
- `String`
- An instantiation or a projection of `KClass<T>`
- An enum type
- An annotation class
- An array of any of above-mentioned types, or an out-projection of `Array<T>` with any of above-

mentioned types.

[Note: Arrays of arrays are not supported. End Note]

Default values for parameters, if any, must be expressions that would be valid arguments to an annotation application, and they cannot refer to other parameters.

An annotation class cannot be instantiated using a constructor invocation in an executable code. Annotation constructor invocations are only allowed in annotation applications, and are only instantiated by the runtime [TODO: when?]

TODO

15.3. Annotation Targets

A declaration of an annotation specifies to which code elements the annotation can be applied. Those code elements are called targets. The following targets exist:

- method declaration
- type declaration
- file
- type-parameter
- parameter
- TODO

If annotation targets are not specified at its declaration, it has the default set of targets: TODO

15.4. Applying Annotations

An argument to an annotation application must be a compile-time constant, a class expression, an annotation constructor invocation, an invocation of `arrayOf(⋯)` function. In the two latter cases, arguments to an invocation must be expressions that would be valid arguments to an annotation application.

File annotations are only allowed at the top of a source file. The import directives below them, if any, are still in effect for the name resolution in file annotations.

If an annotation is specified in a position where it can possibly apply to several different targets, then the target is selected according to the list of possible targets of the annotations. If several targets match, then the first applicable target from the following list is selected: parameter, property, field.

TODO: @-syntax and simplified syntax.

TODO: Meaning of annotations in functional types. Equality of types that differ in annotations. An argument to an annotation application must be a compile-time constant, a class expression, an annotation constructor invocation, an invocation of arrayOf(...) function. In the two latter cases, arguments to an invocation must be expressions that would be valid arguments to an annotation

application.

15.5. Retention Levels

TODO

15.6. Predefined Annotations Significant for Kotlin Compiler

15.6.1. General

Some annotations defined in the standard library have a special meaning for the compiler and may change meaning of the language, including allowed syntactic shapes of some language constructs. They may cause or suppress compiler diagnostics. Some of them have applicability constraints more strict that would follow from their `target(⋯)` annotation. Some predefined annotations have associated enum classes, whose values can be provided as their arguments and change some details of their meaning.

TODO

[Note: Earlier versions of the Kotlin language and standard library had the following annotations, that are currently superseded by corresponding modifiers: `annotation`, `enum`, `data`, `inline`, `noinline`. *End Note]*

15.6.2. `kotlin.Suppress` Annotation

TODO

15.6.3. `kotlin.Deprecated` Annotation

The first parameter is intended to provide motivation and possible workarounds. Its value is not interpreted by the compiler. The second parameter is IDE-specific.

TODO

15.6.4. `kotlin.annotation.Retention` annotation

TODO

15.6.5. `kotlin.annotation.AnnotationRetention` Enum Class

TODO

15.6.6. `kotlin.annotation.Target` Annotation

TODO

15.6.7. `kotlin.annotation.AnnotationTarget` **Enum Class**

TODO

15.6.8. `kotlin.jvm.Synchronized` **Annotation**

TODO

15.6.9. `kotlin.jvm.Strictfp` **Annotation**

TODO

15.6.10. `kotlin.jvm.Volatile` **Annotation**

TODO

15.6.11. `kotlin.jvm.Transient` **Annotation**

TODO

15.6.12. `kotlin.jvm.JvmName` **Annotation**

TODO

15.6.13. `kotlin.jvm.JvmStatic` **Annotation**

- Annotations
- Annotation Declaration

Annotations are means of attaching metadata to code. To declare an annotation, put the `annotation` modifier in front of a class:

```
annotation class Fancy
```

Additional attributes of the annotation can be specified by annotating the annotation class with meta-annotations:

- `@Target` specifies the possible kinds of elements which can be annotated with the annotation (classes, functions, properties, expressions etc.);
- `@Retention` specifies whether the annotation is stored in the compiled class files and whether it is visible through reflection at runtime (by default, both are true);
- `@Repeatable` allows using the same annotation on a single element multiple times;
- `@MustBeDocumented` specifies that the annotation is part of the public API and is to be included in the class or method signature shown in the generated API documentation.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
public annotation class Fancy
```

- Usage

```
@Fancy class Foo {
  @Fancy fun baz(@Fancy foo: Int): Int {
    return (@Fancy 1)
  }
}
```

To put an annotation on the primary constructor of a class, the explicit `constructor` keyword has to be used in the class header, and the annotation section has to be specified immediately before it:

```
class Foo @Inject constructor(dependency: MyDependency) {
  // ...
}
```

Property accessors can also be annotated:

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

- Constructors

Annotations may have constructors that take parameters.

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

If an annotation is used as a parameter of another annotation, its name is not prefixed with the @ character:

```
public annotation class ReplaceWith(val expression: String)

public annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this ===
other"))
```

- Lambdas

Annotations can also be used on lambdas. They will be applied to the `invoke()` method into which
the body of the lambda is generated.

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

- Annotation Use-site Targets

When annotating a property or a primary constructor parameter, there are multiple Java elements
which are generated from the corresponding Kotlin element, and therefore multiple possible
locations for the annotation in the generated Java bytecode. To specify how exactly the annotation
is to be generated, use the following syntax:

```
class Example(@field:Ann val foo,    // annotate Java field
              @get:Ann val bar,      // annotate Java getter
              @param:Ann val quux)   // annotate Java constructor parameter
```

The same syntax can be used to annotate the entire file. To do this, put an annotation with the
target `file` at the top level of a file, before the package directive or before all imports if the file is in
the default package:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

If multiple annotations with the same target is used together, repeating the target can be avoided
by adding brackets after the target and putting all the annotations inside the brackets:

```
class Example {
    @set:[Inject VisibleForTesting]
    public var collaborator: Collaborator
}
```

The full list of supported use-site targets is:

- `file`
- `property` (annotations with this target are not visible to Java)
- `field`
- `get` (property getter)
- `set` (property setter)
- `receiver` (receiver parameter of an extension function or property)
- `param` (constructor parameter)
- `setparam` (property setter parameter)

To annotate the receiver parameter of an extension function, use the following syntax:

```
fun @receiver:Fancy String.myExtension() { }
```

If a use-site target is not specified, the target is chosen according to the `@Target` annotation of the annotation being used. If there are multiple applicable targets, the first applicable target from the following list is used:

- `param`
- `property`
- `field`
- Java Annotations

Java annotations are fully compatible with Kotlin:

```
import org.junit.Test
import org.junit.Assert.*

class Tests {
  @Test fun simple() {
    assertEquals(42, getTheAnswer())
  }
}
```

Since the order of parameters for an annotation written in Java is not defined, a regular function call syntax cannot be used for passing the arguments. Instead, the named argument syntax has to be used.

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

A special case is the `value` parameter; its value can be specified without an explicit name.

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

If the `value` argument in Java has an array type, it becomes a `vararg` parameter in Kotlin:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

If one need to specify a class as an argument of an annotation, use a Kotlin class `KClass`. The Kotlin compiler will automatically convert it to a Java class, so that the Java code will be able to observe the annotations and arguments normally.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any?>)

@Ann(String::class, Int::class) class MyClass
```

Values of an annotation instance are exposed as properties to Kotlin code.

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

16. Executable Code

16.1. General

16.2. Blocks

A block is a sequence of zero or more statements enclosed in curly braces { ⋯ }. Blocks can appear as bodies of methods, function expressions, loop bodies, or as initialization blocks. Standalone blocks cannot be used as statements to create nested scopes (but a similar effect could be used by run method from the standard library. TODO: Example).

A block is evaluated by sequential evaluation of the statements contained in it.

TODO

16.3. Statements

A statement can have one of the following forms: expression statement, declaration statement, assignment statement, loop statement or empty statement (the latter can be a result of putting multiple semicolons in a row).

TODO

16.4. Expression Statement

An expression statement is an expression followed by a semicolon (the latter can be implicit according to the grammar rules). An expression statement is evaluated by evaluation the expression. A value resulted from evaluation of the expression is discarded.

TODO

16.5. Single-Variable Declaration

A declaration statement can have one of the following forms: a variable declaration, a local function declaration, of a local type declaration. The declaration statement is terminated by a semicolon (the latter can be implicit according to the grammar rules). Local properties and extension properties are not supported. A variable declaration declares a read-only (val) or mutable (var) local variable. It can have an optional variable type specification after color, and an optional variable initializer after the = token. At least one of them must be present. Evaluation of a variable declaration statement without an initializer does nothing. Evaluation of a variable

declaration statement with an initializer consists of evaluation of the initializer and assignment of its result to the declared variable.

16.6. Multi-Variable Declaration

A declaration of the form `val (v1, v2, ···) = expr` is equivalent to

```
val $temp = expr
val v1 = $temp.component1()
val v2 = $temp.component2()
...
```

where method names all begin with the prefix component followed a 1-based component index in decimal form without leading zeros, and `$temp` is a fresh name not visible in the user code. Instead of the `val` keyword, the `var` keyword may be used, in which case all declared variables become mutable. Some or all components of a multi-declaration may have optional type annotations – if present, they are copied in the above expansion to declarations of corresponding variables (and can affect, for example, type inference in the corresponding `$temp.componentN()` invocations). All component names `v1`, `v2`, ... must be distinct. Some collection types in the standard library (e.g. arrays) support a fixed number of `componentN` functions that extract their elements with index (N-1). `componentN` functions are also automatically generated for data classes, where N-th function extracts a value of the property initialized from N-th constructor parameter.

TODO: Components of type Nothing and control flow.

16.7. Local Function Declaration

Evaluation of a local function declaration has no effect at run-time.

TODO

16.8. Local Type Declaration

A local type declaration can be a class declaration or an interface declaration. Evaluation of a local class of interface declaration does nothing.

TODO

16.9. Simple Assignment

[Note: An assignment is a statement. It produces no value and cannot be used as an expression. *End Note]*

TODO: Assignment targets: a variable, property, field, array element. Evaluation of assignment. Multi-component assignment is not supported, only multi-component initialization.

16.10. Compound Assignment

TODO: Evaluation of assignment.

16.11. for Loop

TODO

16.11.1. Multi-Declarations in for Loop

TODO: `for((key, value) in map)`

16.12. while Loop

TODO

16.13. Literals

TODO: `true`, `false`, `null`, integers (dec, hex), floating-point, strings, chars.

16.14. Simple Names

A simple name consists of a single identifier. A simple name resolution can be in general described as follows. Each function body, anonymous function body, accessor body, type body, or file represents a scope. Names imported by `import` directives are placed into several special scopes called "import scopes" that effectively enclose the file scope:

TODO: describe precedence of import directives and what exactly scopes are created.

Scopes can be nested. For each two different scopes M and N in the same file (including inport scopes), there are exactly 3 possibilities: A is nested witin B, B is nested within A, or scopes A and B are disjoint. For each occurrence of a simple name in a file there a hierarchy of nested enclosing scopes, viewed from innermost to outermost. To determine meaning of a simple name, scopes are searched from innermost to outermost, performing a name lookup with a given name in each scope. Once a match is found, the process stops and returns the match as a result. If the result turns out to be not valid in the current context for some reason, the process does not resume. At each step, corresponding to a type body, an additional step is inserted after it, looking for a matching symbol in the type's companion object, if any.

16.15. Invocation Expressions

Invocation expressions has a form

```
PrimaryExpression ( ArgumentList )
```

The PrimaryExpression must either be bound to a method group, or to a value that supports an

invocation operation (e.g. a value of a functional type). Type-arguments can only be provided if the primary expression is a method group containing a generic method. To create an instance of a class, we call the constructor as if it were a regular function. Named arguments. Named argument to a functional type invocation.

TODO

16.16. `this` Expression

TODO

16.17. `super` Access

Super-access is a value of the delegate object if it mentions an interface implemented via delegation. It cannot be used as a target of an assignment. In all other cases it can only be used at the left hand side of member access (dot operator).

There are no super expressions corresponding to this expressions introduced by extension functions or extension anonymous functions.

Super access cannot be used to access an extension.

TODO: Describe expressions of the form `super<A>@B`, where `B` is a containing class such that `this@B` is available, and `A` is a supertype of `B` (where it type-arguments may be omitted).

A super-access without an explicit supertype name can be used only if the current type inherits exactly one non-abstract member with the given name (TODO: method overloading?)

16.18. `class` Expressions

`T::class`

Cannot be used on type-parameters. No type-arguments can be provided even if the type `T` is generic (except for the `Array<T>` type, for which a type-argument must be provided, in/out projections in the argument are ignored). `T` cannot syntactically be a functional type. `dynamic::class` is not supported. The type of the `T::class` expression is `KClass<T>`. `(T)::class` is not supported (BUG?). `java.lang.Object::class` is equivalent to `kotlin.Any::class`. `java.lang.Void::class` is equivalent to `kotlin.Nothing::class`.

16.19. Callable References

Callable references to local variables and parameters are not supported.

TODO: Member, non-member

16.20. Operator Expressions

There are several unary and binary operators in Kotlin. Unary operators are translated into method

invocations on their operand. Binary operators are translated into method invocations on their left operand passing the right operand as the single argument to the invocation. Every operator has a method name associated with it (compound assignment operators can be translated in several possible ways). Overload resolution for translated invocations is performed according the regular rules, except that only those candidates are considered that are declared in Java, or declared with `operator` modifier in Kotlin.

- Unary operators

```
| Expression | Translated to |
|------------|---------------|
| `+a` | `a.plus()` |
| `-a` | `a.minus()` |
| `!a` | `a.not()` |
```

This table says that when the compiler processes, for example, an expression `+a`, it performs the following steps:

- Determines the type of `a`, let it be `T`.

- Looks up a function `plus()` with no parameters for the receiver `T`, i.e. a member function or an extension function.

- If the function is absent or ambiguous, it is a compilation error.

- If the function is present and its return type is `R`, the expression `+a` has type `R`.

```
| Expression | Translated to |
|------------|---------------|
| `a++` | `a.inc()` + see below |
| `a--` | `a.dec()` + see below |
```

These operations are supposed to change their receiver and (optionally) return a value.

`inc()/dec()` shouldn't mutate the receiver object. By "changing the receiver" we mean *the receiver-variable*, not the receiver object.

The compiler performs the following steps for resolution of an operator in the **postfix** form, e.g. `a++`:

- Determines the type of `a`, let it be `T`.

- Looks up a function `inc()` with no parameters, applicable to the receiver of type `T`.

- If the function returns a type `R`, then it must be a subtype of `T`.

The effect of computing the expression is:

- Store the initial value of `a` to a temporary storage `a0`,

- Assign the result of `a.inc()` to `a`,

- Return `a0` as a result of the expression.

For `a--` the steps are completely analogous.

For the **prefix** forms `++a` and `--a` resolution works the same way, and the effect is:

- Assign the result of `a.inc()` to `a`,

- Return the new value of `a` as a result of the expression.

- Binary operations

```
| Expression | Translated to |
| -----------|-------------- |
| `a + b` | `a.plus(b)` |
| `a - b` | `a.minus(b)` |
| `a * b` | `a.times(b)` |
| `a / b` | `a.div(b)` |
| `a % b` | `a.mod(b)` |
| `a..b ` | `a.rangeTo(b)` |
```

For the operations in this table, the compiler resolves the expression in the **Translated to** column.

```
| Expression | Translated to |
| -----------|-------------- |
| `a in b` | `b.contains(a)` |
| `a !in b` | `!b.contains(a)` |
```

For `in` and `!in` the procedure is the same, but the order of arguments is reversed. (TODO: Order of evaluation?)

```
| Expression | Translated to |
|------------|---------------|
| `a == b` | `a?.equals(b) ?: b.identityEquals(null)` |
| `a != b` | `!(a?.equals(b) ?: b.identityEquals(null))` |
```

The operators `===` and `!==` (identity checks) are not overloadable, so no conventions exist for them

The `==` operation is special in two ways:

- It is translated to a complex expression that screens for `null`'s, and `null == null` is `true`.

- It looks up a function with a specific *signature*, not just a specific *name*. The function must be declared as

```
fun equals(other: Any?): Boolean
```

Or an extension function with the same parameter list and return type.

```
| Symbol | Translated to |
|--------|---------------|
| `a > b`  | `a.compareTo(b) > 0` |
| `a < b`  | `a.compareTo(b) < 0` |
| `a >= b` | `a.compareTo(b) >= 0` |
| `a <= b` | `a.compareTo(b) <= 0` |
```

All comparisons are translated into calls to `compareTo`, that is required to return `Int`.

- Indexing and invocations

```
| Symbol | Translated to |
| -------|-------------- |
| `a[i]`  | `a.get(i)` |
| `a[i, j]`  | `a.get(i, j)` |
| `a[i_1, ...,  i_n]`  | `a.get(i_1, ...,  i_n)` |
| `a[i] = b` | `a.set(i, b)` |
| `a[i, j] = b` | `a.set(i, j, b)` |
| `a[i_1, ...,  i_n] = b` | `a.set(i_1, ..., i_n, b)` |
```

Square brackets are translated to calls to `get` and `set` with appropriate numbers of arguments.

```
| Symbol | Translated to |
|--------|---------------|
| `a(i)`  | `a.invoke(i)` |
| `a(i, j)`  | `a.invoke(i, j)` |
| `a(i_1, ...,  i_n)`  | `a.invoke(i_1, ...,  i_n)` |
```

Parentheses are translated to calls to invoke with appropriate number of arguments.

- Assignments

```
| Expression | Translated to |
|------------|---------------|
| `a += b` | `a.plusAssign(b)` |
| `a -= b` | `a.minusAssign(b)` |
| `a *= b` | `a.timesAssign(b)` |
| `a /= b` | `a.divAssign(b)` |
| `a %= b` | `a.modAssign(b)` |
```

For the assignment operations, e.g. `a += b`, the compiler performs the following steps:

- If the function from the right column is available

- If the left-hand side can be assigned to and the corresponding binary function (i.e. `plus()` for `plusAssign()`) is available, report error (ambiguity).

- Make sure its return type is `Unit`, and report an error otherwise.
- Generate code for `a.plusAssign(b)`
- Otherwise, try to generate code for `a = a + b` (this includes a type check: the type of `a + b` must be a subtype of `a`).
- Discussion of the ambiguity rule

We raise an error when both `plus()` and `plusAssign()` are available only if the lhs is assignable. Otherwise, the availability of `plus()` is irrelevant, because we know that `a = a + b` can not compile. An important concern here is what happens when the lhs becomes assignable after the fact (e.g. the user changes **val** to **var** or provides a `set()` function for indexing convention): in this case, the previously correct call site may become incorrect, but not the other way around, which is safe, because former calls to `plusAssign()` can not be silently turned into calls to `plus()`.

16.21. Range Expression

TODO: Describe range expressions 1..10.

16.22. `return` Expression

The type of the return expression is kotlin.Nothing.

TODO: return, non-local return, labelled return.

16.23. `throw` Expression

The type of the throw expression is kotlin.Nothing.

TODO

16.24. `is`, `!is` Operators

The is operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly, see §Smart-cast for exact details.

TODO

16.25. `in`, `!in` Operators

TODO: Usage with range expressions, with collections.

BUG: Order of evaluation of operands is reversed.

16.26. `as`, `as?` Expressions

TODO

16.27. Member Access Operator

TODO: with simple name RHS, and other expressions RHS

16.28. Safe Access Operator ?.

A package name cannot be followed with ?.. A class name cannot be followed with ?. and a nested class name.

16.29. Conditional Expression

TODO: Describe if-else expression.

The condition in the conditional expression has the expected type Boolean. It means that the conditional expression must be assignable to the type Boolean, and in case the conditional expression is a generic function invocation that requires type inference, the type Boolean is used as the expected type in the type inference.

16.30. when Expression

when matches its argument against all branches consequently until some branch condition is satisfied. when can be used either as an expression or as a statement. If it is used as an expression, the value of the selected branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. Each branch can be a block, and its value is the value of the last expression in the block. The else branch is evaluated if none of the other branch conditions are satisfied. If when is used as an expression, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions. If many cases have to be handled in the same way, the branch conditions may be combined with a comma.

Two different syntactic kinds of when expressions exist: * Governed-by-value when expression. In this case, the keyword when is immediately followed by a parenthesized *governing expression*, followed by a sequence of zero or more branches, enclosed in curly braces:

```
when(expr) {
    pattern_list_1 -> branch_1
    pattern_list_2 -> branch_2
    ...
}
```

Each branch is a *pattern list* followed by the → token, followed by a branch body. A branch body is an expression or block. A pattern list is either a sequence of one or more patterns separated by commas, or the else keyword. A *pattern* can be one of the following:

- an expression
- a type test (is, !is)
- a range test (in, !in)

TODO: Evaluation order

- Governed-by-conditions `when` expression. In this case, the keyword `when` is immediately followed by a sequence of zero or more branches, enclosed in curly braces:

```
when(expr) {
    condition_list_1 -> branch_1
    condition_list_2 -> branch_2
    ...
}
```

Each branch is a *condition list* followed by the → token, followed by a branch body. A branch body is an expression or block. A condition list is either a sequence of one or more expressions separated by commas, or the `else` keyword. A governed-by-conditions `when` expression is evaluated as follows: all conditions in all condition branches are evaluated in order (top-to-bottom, left-to-right) until some condition evaluates to `true`. In this case evaluation of conditions stops, and the branch body in the corresponding branch is evaluated. If all conditions are evaluated to `false`, then the branch body following the `else` keyword is evaluated, if there is any. Otherwise, the evaluation of the `when` expression completes.

Every condition has the expected type `Boolean`. It means that the conditional expression must be assignable to the type `Boolean`, and in case the conditional expression is a generic function invocation that requires type inference, the type `Boolean` is used as the expected type in the type inference. (BUG: expected type in type inference is not yet implemented).

TODO

16.31. Object Expressions

An object expression provides a way to declare an anonymous local class and instantiate it within an expression. An object expression can specify the direct superclass and superinterfaces of the declared class, and provide implementations of its members. The declared class does not have a name and is not denotable. Evaluation of an object creation expression involves invocation of the superclass constructor, evaluation of initialization blocks and property initializers within the expression, and returns the created instance. The created instance is also available as this within the expression. If an object expression has an immediately preceding label, then the created instance is also available as labelled this expression that refers to this label.

The declared class cannot be abstract and must override all inherited abstract members. The declaration of an anonymous class cannot have any modifiers.

A body of an anonymous object expression is mandatory (unlike other class and object declarations).

Otherwise, the body of an anonymous class is governed by the same rules as object declarations (see §?).`

16.32. Anonymous Functions

16.32.1. General

An anonymous function can have two different syntactic forms: a function expression, or a function literals. The difference is not purely syntactical – some constructs (e.g. return expressions) have different meaning within them.

TODO: Evaluation rules for anonymous functions. Body is not evaluated immediately. Rather, the whole expression produces a values of a functional type (§Functional Types), whose invocation results in evaluation of the body of the anonymous function. Captured variables, closure, exceptions.

16.32.2. Function Expression

A function expression looks very much like a regular function declaration, except that its name is omitted. (DEVIATION: The current compiler implementation allows to specify an optional name, but it has some unfortunate syntactic interactions with annotations and should be disallowed). It also cannot have a type-parameter list (although can use type-parameters that are already in scope). Its body can be either a block (§block) or the `=` token followed by an expression (return expressions are not allowed in function expressions with expression body). Because a function expression lacks a name, a direct recursive invocation is not possible (a workaround is either to convert the expression into a named local function, or to assign the expression into a named mutable variable). A function expression cannot have @inline annotation. If return type is not specified and the function has a block body, then the return type is assumed to be kotlin.Unit. If return type is not specified and the function has an expression body, then the type can be inferred from the expression.

16.32.3. Function Literal

Similar to a block, parameters are optional, parameter types are optional, the default parameter is `it`, cannot use return statement to exit function literal.

16.33. Static Type Assertion Expression

Deprecated. May cause parsing ambiguities between blocks and function literals.

16.34. Parenthesized Expression

Parenthesized expression has the same value and classification as the expression inside parentheses. Its main use is to change default parsing or grouping of expressions. It is possible to assign a parenthesized variable or property. Parentheses do not change order of evaluation of operands, although they could change order of evaluation of operators by means of changing the shape of a parse tree.

Parentheses has an effect on parsing blocks vs. function literals.

A prefix `super<T>` cannot be parenthesized unless it denotes a delegate.

Every expression can be placed in a context where an atomic expression is required by enclosing it in parentheses (provided it has suitable classification and type). A method group cannot be parenthesized.

16.35. Invocation Expression

TODO

16.35.1. Passing Argument as a Trailing Function Literal

Empty argument list can be omitted in this case: `run { ⋯ }`

TODO

16.35.2. Implicit invoke Method Invocations

`x.foo()` cannot be rewritten multiple times, e.g. to `x.foo.invoke.invoke()`

TODO

16.36. Indexer Access Expression

Translated to an invocation of the method `get`, or the method `set`, or both. Index expressions are evaluated only once, left-to-right. Empty index list is not supported. Named indexes are not supported. Spread operator is supported (TODO: implement). Passing trailing lambda as the last parameter is not supported.

TODO: Do we support further rewriting of `x.get(⋯)` to `x.get.invoke(⋯)`?

16.37. Anonymous Object Creation Expressions

May specify no base class (`object { ⋯ }`), may specify a base class (with a constructor invocation) and zero or more interfaces (`object : Base(⋯), I1, I2 { ⋯ }`). Anonymous class cannot introduce its own type-parameters, but can use type-parameters that are in scope. Every expression declares a distinct anonymous class — even if two expressions are textually and semantically identical, their anonymous classes are unrelated. May not contain constructor declarations, but may contain initialization blocks. May not contain a companion object declaration. May contain nested class declarations, but they must be inner classes. May not contain nested interface declarations. Anonymous classes have access to current instances of their enclosing classes.

16.38. try Expressions

An exception type in a catch clause cannot be a type-parameter (TODO: Implement). An exception type in a catch clause cannot be a nullable type. An exception type in a catch clause cannot be the type `Nothing`.

No initializer is allowed for a catch variable. A catch variable is considered initially assigned. A catch variable must have an explicit type that must be the type `Throwable` or subtype thereof.

TODO: try-catch+, try-catch*-finally.

16.39. Order of Evaluation

Generally, the order of evaluation is left to right, non-lazy (eager). Some expressions have special rules for order of evaluation of their constituent parts (some of them may be not evaluated at all). Order of evaluation of named arguments corresponds to their order at the invocation site, not the declaration site.

TODO

17. Name and Overload Resolution

17.1. General

TODO

17.2. Simple Names

A simple name is a single identifier. Its meaning depends on what symbol with that name are in scope. If only on symbols with that name is in scope, then the simple name refers to it. If there are multiple symbols with this name are in scope, then, informally, the symbol whose declaration is "closest" to the occurrence of the simple name is selected. For more precise rules, see TODO

TODO

17.3. Qualified Names

TODO

Resolution of qualified names always proceeds from left to right, and the meaning of an identifier to the left is always fixed before an indentified to the right starts to being resolved. There is backtracking in this process. [Example: When meaning of the qualified name A.B is resolved, the identifier A may initially have several possible candidates, but exactly one of them is selected according to the rules in this specification. Then an attempt to resolve the identifier B after the dot happends. In case a member with this name cannot be found within A (with already fixed meaning), the compiler does not resume resolution process of the identifier A to find another candidate that initially had less priority, although in principle, it may have a member named B. End Example]

17.4. Name Lookup

TODO

17.5. Argument Lists

An argument list for an invocation is the sequence of zero or more arguments, separated by commas. The argument list itself is enclosed in parentheses. An argument can be one of the following kinds:

- A positional argument (an expression),

- A named argument (an identifier, followed by the `=` token, followed by an expression),

- A positional spread argument (the `*` token, followed by an expression),

- A named spread argument (an identifier, followed by the `=` token, followed by the `*` token, followed by an expression).

[Note: Sometimes the `token in spread arguments is called the spread operator in an informal description of the Kotlin language. But this word usage is only informal, and` is not a unary prefix operator in the usual sense (as used throughout this specification). It can only appear in spread arguments, as specified by their grammar, and cannot be used in other places where usual operators can be used. For example, the purpoted invocation expression with a parenthesized spread operator `f(x = (y))` `is syntactically ill-formed. On the other hand, the invocation expression f(*z is IntArray) is syntactically well-formed and can be well-typed, but it is equivalent to f((z as IntArray)).` Note that if `*` were thought of as a unary prefix operator, it would have an unusually low precedence in this context. *End Note]*

TODO: restrictions on order of arguments.

17.6. Positional Arguments

A positional argument is just an expression. TODO

17.7. Named Arguments

A named argument is an identifier, indicating the corresponding parameter name, followed by the token `=`, followed by an expression. *[Note:* So, a named argument is syntactically similar to an assignment statement with a simple name of the left-hand side. *End Note]* The name in a named argument shall correspond to one of the names of the parameters of a function being invoked. *[Note:* The overload resolution mechanism guarantees that functions without such parameter are not considered applicable in this invocation. *End Note].* A named argument cannot follow a positional argument in an argument list.

Names of all named arguments in the same argument list shall be pairwise distinct.

TODO

17.8. Default Parameter Values

A function declaration can optionally provide default values for some or all its parameters. A default value cannot be provided for the receiver parameter in an extension function. A default value is an expression (TODO: its evaluation context, available parameters) specified after the token

`=` following the parameter type.

[*Example:* TODO *End Example*]

A default value for a parameter has no effect in an invocation were an actual argument was provided for that parameter. For a parameter that has no corresponding actual argument, its default value is used instead of the missing argument. (TODO: order of evaluation).

TODO

17.9. vararg Invocations

At most one parameter in a function declaration can have the `vararg` modifier. This modifier indicates that a variable number (zero, one or more) of arguments corresponding to this parameter can be provided in a function invocation, as in more detail is described below. The type of the parameter annotated with the `vararg` modifier is different from the type specified in its declaration — the type of the parameter is an array type, whose element type is taken to be the type specified in the declaration. More precisely, if the specified type is one of the primitive types, then the corresponding primitive array type is used as shown in the following table:

- `Boolean` — `BooleanArray`
- `Byte` — `ByteArray`
- `Short` — `ShortArray`
- `Int` — `IntArray`
- `Long` — `LongArray`
- `Char` — `CharArray`
- `Float` — `FloatArray`
- `Double` — `DoubleArray`

Otherwise, if the specified type is the special type `Nothing` or the corresponding nullable type `Nothing?`, then a compile-time error occurs.

Otherwise, let the specified type be some type `T`. In this case the type of the parameter is the projection type `Array<out T>`.

TODO: Consider cases when `T` is a type-parameter (possibly, bounded) as is substituted with primitive types, or special types `Nothing` or `Nothing?`.

Arguments provided to a `vararg` parameter shall be either:

- a single named argument whose name matches the name of the vararg parameter, or
- a (possibly empty) sequence of positional arguments or spread arguments intermixed in any order.

Arguments provided to a `vararg` parameter are consolidated into a single array instance (TODO: of type ...). In particular:

- If no arguments are passed, then an empty array is created and used as the parameter value;

- If a named argument or exactly one positional argument is provided, then a singleton array is created, whose only element is the argument provided, and this array is used as the parameter value;

- If one spread argument `*a` (where `a` can be any expression) is provided, then the type of `a` shall be the same as the type of the parameter (or a subtype thereof) — in which case a copy of the array `a` is created and is used as the parameter value (copying of the array can be elided by the compiler or runtime environment if this would not have any observable effects); TODO: define "observable effects" (excludes performance and memory consumption, and any effects depending on them explicitly or through race conditions or asynchronous exceptions).

- If multiple spread arguments `*ai` provided, then the type of every `ai` be the same as the type of the parameter (or a subtype thereof) — in which case a new array instance of the length equal to the sum of lengths of all arrays in spread arguments is created, whose content is a concatenation of contents of all arrays in spread arguments, in the same order as the arguments are specified in source; [Note: The creation of the array could fail with an exception if its lengths exceeds the maximum array length supported by the platform, or because of insufficient memory available to allocate the array. End Note]

- In a general case, a mixture of positional and spread arguments can be given in any order — in which case the effective behavior is as if every positional argument is replaced with a spread argument with a singleton array whose only element is that positional argument, and then the spread arguments are processed as specified in the previous bullet.

[Note: If the specified type of a `vararg` parameter is a primitive type (say, `Int`, so the actual type of the parameter is `IntArray`) and a spread argument `*a` is provided for that parameter, then the type of `a` shall be the corresponding primitive array type (`IntArray` in this case), and not the generic `Array<T>` class instantiated with the primitive type (`Array<Int>` in this case). CONSIDER: relax this requirement. *End Note]*

If a generic class or interface with a type-parameter `T` (and, possibly, other type-parameters) declares a method whose parameter list has a `vararg` parameter whose specified type is `T`, then it is not possible to override that method in an instantiation of that generic class or interface where `T` is instantiated with a primitive type. *[Rationale:* Suppose the primitive type used as the type-argument is `Int`. The type of the parameter in the original declaration is `Array<T>`, and its type in the generic instantiation is `Array<Int>`. In a purpoted override the corresponding parameter would be declared as `vararg x : Int`, then would result in the `IntArray` type, not matching the type `Array<Int>` in the method being overridden. *End Rationale]*

TODO

17.10. Matching Arguments with Parameters

TODO

17.11. Candidate Method Search

TODO

17.12. Potential Applicability

TODO

17.13. Actual Applicability

TODO

17.14. Overload Resolution

17.14.1. General

TODO

17.14.2. Better Conversion

TODO

17.14.3. Better Candidate

TODO

17.14.4. Best Candidate

TODO

17.15. Type Inference

TODO

Expression Typing Facade Computes a type for an expression.

Input:

- expression
- context (ExpressionTypingContext)

Output:

- TypeInfo for the expression

Code: ExpressionTypingFacade.getTypeInfo()

- Type Info

For each expression we collect type info, that means "what type does this expression have independently of context". By context dependency we call the dependency of the expected type for this expression. After the expected type for the expression becomes known, we complete analysis of

the expression according to it.

[Example.

```
fun emptyList<E>(): List<E>
val list: List<Int> = emptyList()
```

Here the type of the expression emptyList() is List<Int>, but we know it because we declared it explicitly as a type of a variable (and therefore it became an expected type for the expression). If we examine this expression independently of context, it'll have the type info List<E> where E is unknown.

Another example:

```
fun listOf<E>(vararg elements: E): List<E>
val list: List<Int> = listOf(1, "a")
```

Type info of the expression listOf(1, "a") is List<E> where there are constraints on type E derived from actual arguments (1, "a"): - E is a supertype of Int - E is a supertype of String

But if we consider this expression in its context with expected type List<Int>, it occurred to be not typeable, because expected type adds a constraint E is a subtype of Int that makes the constraints incompatible. End Example]

TypeInfo for an expression is: – type that might have unknown yet type-arguments (T1...Tn); – data flow info; – constraint system that imposes constraints on type-arguments (T1...Tn) as well as some other variables; system might be empty.

Code: JetTypeInfo Todo: now a constraint system corresponds to the resolved call, not to the expression.

Call Resolver

Input: – call r.foo(a1...an, fl1...flm) – where fl1... flm are function literal arguments; – context (ResolutionContext) including context dependency flag (DEPENDENT/INDEPENDENT)

Output: – typeInfo – where constraint system is solved and type is known for INDEPENDENT mode – resolution results

Code: CallResolver.doResolveCall()

The process of the call resolution should go in the following steps:

- Resolve receiver and arguments type infos.

Analyze receiver and arguments in DEPENDENT mode: r → typeInfo ai → typeInfo

Analyze explicitly-typed function literal arguments (and their bodies). For implicitly-typed function literal argument we'll be satisfed with it's shape. A shape of the function literal argument is: - a type

if it's explicitly-typed, - a special placeholder denoting it's a function type and containing value parameter types and it's number info (if there's any) for implicitly-typed function literal. For implicitly-typed function literal { (x, y) → x + y } a shape is (???, ???) → ???. Here ??? means "unknown type".

- build and prioritize tasks / candidates (see Task Prioritizer)

Input: – receiver type info – function name – context

Output: – resolution task (actually OrderedMultiTask)

The candidates are grouped and inside each group they can be ordered and unordered. The following interfaces describe this:

```
interface ResolutionTask {
 fun getStatus(): ResolutionStatus
 fun getResultingCall(): ResolvedCall?
}

interface SingleCandidateTask : ResolutionTask {
 val candidate: ResolutionCandidate
 val resolvedCall: ResolvedCall
}

interface OrderedMultiTask : ResolutionTask {
 val tasks: Iterable<ResolutionTask>
}

interface UnorderedMultiTask : ResolutionTask {
 val tasks: Set<ResolutionTask>
}
```

- Analyze tasks

All candidates that are stored in a task and its subtasks are resolved (by Resolve candidate function). Tasks should be built and analyzed lazily. The SingleCandidateTask is successful if the candidate is successful. The OrderedMultiTask is successful if it contains a successful task. The resulting candidate is the one of the frst successful task. The UnorderedMultiTask is successful if it contains successful tasks (not all of them must be successful) and there is a maximally specifc one among their candidates.

- Resolve (try) each candidate (see Resolve candidate)

Input: – arguments and receiver type infos – shape of each function literal argument – resolution candidate

Output: – status – typeInfo

Candidate resolution consists of two steps: - map arguments to parameters (see Value Arguments to Parameters mapping); - check that constraint system built on assumptions "argument is a subtype

of a formal parameter type" isn't failure.

- Resolve overloading conflcts (find the most specifc candidate)

Input: the first task with successful candidates Output: a candidate or an error

See Choosing the most specifc candidate.

- Complete resolution if the mode is INDEPENDENT (see Solve constraint system)

Input: – typeInfo where – constraint system is non-trivial – all nested implicitly-typed function literal arguments were not analyzed – expected type

Output: – substitution for all type variables in the constraint system – analyzed bodies of implicitly-typed function literal arguments – result type of the expression

Resolution Candidate Sometimes function foo requires more than one receiver to be correctly resolved:

```
class A {
  fun B.foo() {}
}
```

In such cases when we resolve it two receivers should be available in the context. In the following example there are an explicit receiver b and an implicit receiver this of the type A:

```
fun A.test1(b: B) {
  b.foo()
}
```

One more example with two implicit receivers of types A and B:

```
fun test2(a: A, b: B) {
  with (a) {
    with (b) {
      foo()
    }
  }
}
```

We will name a receiver that function is an extension to as extension receiver (B in the first example) and a receiver that function is a member of as dispatch receiver (A in the example).

Let's introduce the following notation

```
   dispatch receiver
-------------------.foo
   extension receiver
```

to denote the dispatch receiver and the extension receiver for concrete function invocation of foo. In this notation foo denotes function descriptor.

For our examples it's:

```
   this
------.foo
   b
```

for invocation in test1 and

```
   a
---.foo
   b
```

for invocation in test2.

Actually, for second example it's

```
   this@A
---------.foo
   this@B
```

where this@A, this@B – implicit extension receivers for corresponding function literals, but for simplicity we'll sometimes write a, b. We write dash symbol to indicate the absence of dispatch receiver or extension receiver. For example,

```
   a
---.foo
   -
```

describes an invocation a.foo where a has type A and foo is a member of A:

```
class A { fun foo() {} }
```

```
   -
---.foo
   a
```

describes an invocation a.foo where foo is declared as extension function to A:

```
class A
fun A.foo() {}
```

To shorten notation in complicated cases, let's denote a

```
  a
---.foo
  -
```

as `a.foo^m` and

```
  -
---.foo
  a
```

as `a.foo^e`.

So Resolution Candidate is:

- function descriptor
- dispatch receiver
- extension receiver
- Task Prioritizer

To be able to resolve `invoke` convention simultaneously with the usual invocation we have to resolve properties beforehand. Because `invoke` in invocation `r.foo()` can be extension to `R` or extension to (or member of) `Foo`, we should resolve two properties: with explicit receiver `r` and without. We'll denote them as `r.foo` and `foo`. The expression `r.foo` can indicate a class object (or object) as well.

Input:

- invocation `r.foo(args){fl}` where
- `foo` is the name of the function by which all possible candidates will be found;
- receiver `r` denotes any kind of expression and has the type `R<T1···Tn>` (some of `Ti` may be unknown yet) or may be absent;
- `args` are value arguments (including function literal arguments that are invoked as usual, without convention), `fl` – the function literal argument (if any) that is invoked with special syntax.
- implicits - all implicit receivers that are available in the context.

Output:

- prioritized resolution candidates for foo.

The candidates for the simple invocation r.foo(x) with the explicit receiver r and the argument list x should be built by the following scheme:

1. Members. Will be denoted as r.foo^m

```
( x) :
r
 -
. foo(x)
```

- Extensions. Will be denoted as r.foo^e

```
( x) :
implicits ∪ { - }
r
. foo(x)
```

Which means

```
for (implicit in implicits) {
 add
implicit
r
. foo(x)
}
add -
r
. foo(x)
```

The extension candidates should be prioritized by implicit receiver. The union of members and extensions will be denoted simply as `r.foo* ( x)`. Note that all members and extensions are found with respect to available smart cast types of receiver `r`. The candidates within one group but found from diferent smart cast types have the same priority. It might turn out later there is the most specifc candidate, so several candidates don't mean ambiguity at once. The call `r.foo(args){fl}` might contain implicit `invoke` invocations. Note that `invoke` members have more priority than `foo` extensions, which means the candidates for `invoke` calls should be built together with the usual ones.

[Note. (from the discussion 28.01.15) If the receiver type is unknown yet (`T` where there are some constrains on `T`), it should be completed immediately. We don't want to collect all possible extension functions. End Note]

If only type-parameters are unknown in the receiver type (e.g. `List<T>`), the candidates might be found for incomplete type, it's important for the Immutable map builder case support).

One should be very careful collecting all possible extensions of `List<T>`. On the one hand, all `T` substitutions should be considered (both extensions to `List<String>` and `List<Int>` may apply). On the other hand, some candidates might be thrown away using the current `T` substitution. (Say, if `T` is a subtype of `Number`, extension to `List<String>` is defnitely inappropriate).

- The receiver r exists.

Add candidates for the explicit receiver:

1. Add members matching by all arguments: r.foo^m(args, fl).

2. Add members for invoke. The following cases have the same priority. a) 'Invoke' is positioned after foo: r.foo^m

```
.invoke^m(args , fl) .
```

b) 'Invoke' is positioned before the last function literal argument:

```
r.foo^m(args).invoke^m(fl) .
```

c) Both invocations are made through `invoke`:

```
r.foo^m.invoke^m(args).invoke^m(fl).
```

Note that it is possible to find appropriate `invoke` candidates only when resolution of receiver call is completed. This means that the computation of the last `invoke(fl)` should be deferred.

- Add extensions (member extensions and pure extensions) matching by all arguments: `r.foo^e(args, fl)`.

- Add extensions for `invoke`. The following groups of candidates have the same priority, but inside each group the candidates should be prioritized by an implicit receiver. The same logic applies as in item 2, but now both members and extensions are under consideration:

a) `invoke` is positioned after foo: `r.foo*.invoke*(args , fl)`. b) `invoke` is positioned before the last function literal argument: `r.foo*(args).invoke*(fl)`. c) Both invocations are made through `invoke`: `r.foo*.invoke*(args).invoke*(fl)`.

The `invoke' function can be an extension to the explicit receiver `r` as well:

d) foo

```
r
.invoke(args , fl)
```

e) foo

```
r
.invoke(args).invoke*
( fl)
```

In the cases d and e `invoke` is a member extension, which means it is an extension to `R` declared in the class `Foo`. So `foo` is 'dispatch receiver' for such invocation, `r` is an 'extension receiver'. The case 4 is illustrated with the examples below.

II. The receiver r doesn't exist.

1. At frst, we fnd all local declarations named `foo`:

add locals That means functions -

```
 -
. foo and properties foo
 -
.invoke and -
foo .invoke
```

where the function or the property `foo` is declared locally.

- We repeat the process defined in a stage I trying all implicit receivers that are available in our scope:

```
for (implicit in implicits) {
 add_candidates_for_explicit_receiver (implicit)
}
```

That means we sequentially imply that $foo(\cdots)$ is:

```
this.foo(...)
this@A.foo(...)
this@B.foo(...)
```

where `this`, `this@A`, `this@B` are implicit receivers. Thus we have candidates sorted by implicit receiver.

- At last we consider the case when there is no explicit receiver and `foo` is declared not locally: add non-locals. Which means the same as case 1 but the function or the property foo is declared not locally.

[Note. (from the discussion 29.01.15) The direct loop through implicit receivers takes square time, because for each implicit receiver (as candidate for 'dispatch' receiver) we check all implicit receivers (as candidates for 'extension' receiver). But actually we need to collect only all members named 'foo' of implicit receivers. So some preliminary work might be done to simplify answering

whether given implicit receiver have a potentially applicable member-extension function with given name or not. End Note]

[Example

- The most non-trivial case is when there are extensions to invoke.

```
r.foo
 -
.invoke
```

```
interface R
val R.foo: Foo
interface Foo {
 fun invoke()
}
fun test(r: R) {
 r.foo()
}
```

```
implicits ∪ { - }
r.foo .invoke
```

That consists of the cases:

```
r.foo .invoke
```

```
interface Foo
interface R { val foo: Foo } //or property can be an extension
fun Foo.invoke() {}
fun test(r: R) {
 r.foo()
}
```

```
implicit(b)
r.foo .invoke
```

```
interface Foo
interface R { val foo: Foo } //or property can be an extension
interface B {
 fun Foo.invoke()
 fun test(r: R) {
 r.foo()
 }
}
```

```
foo
r
.invoke
```

```
interface R
interface Foo {
 fun R.invoke()
}
fun test(r: R, foo: Foo) {
 r.foo()
}
```

The real example for the last case is ExtensionFunction interfaces:

```
public interface ExtensionFunction0<in T, out R> {
 public fun T.invoke() : R
}
fun test(f: Int.()->Unit) {
 1.f()
}
```

- More real examples.
- String.plus in html builders.

```
fun foo() =
 html {
 head {
 title { +"Foo" }
 }
 }
fun html(init: HTML.() -> Unit): HTML
abstract class Tag(val name: String) {
 fun String.plus() {
 }
}
class HTML() : Tag("html") {
 fun head(init: Head.() -> Unit)
}
class Head() : Tag("head") {
 fun title(init: Title.() -> Unit)
}

class Title() : Tag("title")
```

The following candidates are built for the +"Foo" invocation:

```
1. "Foo"
 -
fun plus(other: Any?): String defined in jet.String
2. this@title
 "Foo" fun String.plus(): Unit defined in Title
3. this@head
 "Foo" fun String.plus(): Unit defined in Head
4. this@html
 "Foo" fun String.plus(): Unit defined in HTML
5.
 -
"Foo"
fun String?.plus(Any?)
fun IntArray.plus(Int)
...
fun Array<T>.plus(T)
fun Array<T>.plus(Iterator<T>)
...
fun Iterable<T>.plus(T)
The frst candidate is a member in a class String ( r
 -
. foo(x) ).
```

The candidates 2-4 are member extensions sorted by implicit receivers

```
( implicits ∪ { - }
⌐
. foo (x) )
```

The last candidates are extensions, most of which are incompatible by receiver type, but in the absence of more appropriate candidates the corresponding error will be generated. All 2-4 candidates are successful (appropriate), but the candidate list is sorted, and the result is the second one.

```
2. 'Invoke' vs usual invocation
class A {
 fun bar(foo: (Int)->Int) {
 foo(42)
 }

 fun foo(p: Int) {}
}
```

There are two candidates for the `foo(42)` invocation:

```
1. -
 -
 local val foo: (Int)->Int
&
foo
 -
 fun invoke(Int) defined in Function1<Int, Int>
2. this
 -
 fun foo(Int) defined in A
```

The frst candidate is successful, that means foo is resolved to the local variable.

Todo: Now 'invoke' candidates are resolved a bit diferently. In the proposed approach for `foo(42)` we have a bunch of candidates - functions:

```
{ fun foo(Int), fun invoke(Int) }
```

But now we collect both functions and variables with the same name together:

```
{ fun foo(Int), val foo: Function<Int, ...> }
```

And for variables the corresponding `invoke` is resolved later in `CallTransformer`.

Code: TaskPrioritizer.computePrioritizedTasks.

Constraint system The most controversial idea of the proposing approach is that the constraint system should store references to implicitly-typed function literal arguments. They depend on unknown type variables and so cannot be analyzed at once, but can add some additional information after being analyzed.

The constraint system is:

- map: type variable → type bounds;

- type bounds consist of upper, lower and exact bounds on type variable; these bounds might equal to or contain other type variables;

- references to implicitly-typed function literal arguments (knowing which type variables depend on function literal argument, and which function literal argument depend on);

- information about errors.

Let's look at the example from the kotlin standard library:

```
fun <E> newHashSet(): HashSet<E>
fun <T, C: MutableCollection<T>>
 Iterable<T>.toCollection(result: C) : C
fun test(list: List<Int>) {
 list.toCollection(newHashSet())
}
```

Constraint system for this example is:

```
type variables: E, T, C
type bounds: T → { lower bound: Int }
 C →
 { upper bound: MutableCollection<T>,
 lower bound: HashSet<E> }
```

We'll see below how we're going to solve it. The following interface describes the operations that we want to do with the constraint system:

```
interface ConstraintSystem {
 fun registerTypeVariables(typeVariables)
 fun addSubtypeConstraint(constrainingTypeInfo, subjectType)
 fun addSupertypeConstraint(constrainingTypeInfo, subjectType)
 fun addFunctionLiteralArgument(fl)
 private fun addSimpleConstraint(Constraint)
 private fun addConstraintSystem(ConstraintSystem)
}
interface Constraint {
 val kind // SUBTYPE, SUPERTYPE or EQUAL
 val constrainingType
 val subjectType
}
interface VariableConstraint : Constraint {
 override val subjectType //is a type variable
}
```

When we register type variable, we add all upper bounds for it as constraints. When we add subtype or supertype constraint, we have to separately add `typeInfo.type` and `typeInfo.constraintSystem` to our system:

```
addSimpleConstraint(
  Constraint(kind, constrainingTypeInfo.type, subjectType)
addConstraintSystem(constrainingTypeInfo.constraintSystem)
fun addSimpleConstraint(Constraint)
```

When we add a new simple constraint we reduce it. Reduction of the constraint includes the following:

- fnding a common supertype of type constructors of subject type and constraining type; it's an error if it's not found;

- generating new constraints for corresponding type-arguments of subject type and constraining type.

Later we'll need a function that in addition to generating new constraints takes a handler for processing every new one (that constraints a type variable, otherwise is can be reduced):

```
fun reduceConstraint(c: Constraint,
  handler: (VariableConstraint)->Unit)
```

Adding a new `Constraint` may produce new type bounds for type variables. The process of updating the dependent type variables after generating constraints will be described below (see Constraint system incorporation).

```
fun addConstraintSystem(ConstraintSystem)
```

When we add the whole system, we add all data in it (type variables, type bounds and function literal arguments) to our system. The detail of implementation whether the reference to constraint system is stored or the data is copied is irrelevant here. The constraint for `constrainingTypeInfo.type` that we add before adding constraint system often ties together type variables from both systems (current and the one we add).

Code: ConstraintSystem, ConstraintSystemImpl.

Todo: Now interdependence for type variables from diferent (outer and inner) calls is not supported.

Resolve candidate

```
Input:
— invocation r.foo(a1
... an
, fl 1
... flm
)
— where fl1
... flm
 are function literal arguments;
— TypeInfo for receiver r ;
— TypeInfo for arguments a1
...an
 where all arguments were successfully mapped to
foo function's value parameters before;
— resolution candidate dispatch _ receiver
extension _ receiver
. foo (either dispatch_receiver, either
extension_receiver is r);
— dataFlowInfo that was obtained after analyzing arguments.
Output:
— status (SUCCESS or a specifc error)
— TypeInfo for invocation (if success)
```

If in the invocation some type-parameters are specifed, we consider a substituted function descriptor for foo instead. We build a constraint system that stores information about all unknown type-parameters (of function `foo`, receiver and function arguments `a1...an`). The following describes the constraints that are added to the system:

```
val s = ConstraintSystem()
s.registerTypeVariables(foo.typeParameters)
s.addSubtypeConstraint(
 extension_receiver.typeInfo, foo.getExtensionReceiver.type)
s.addSubtypeConstraint(
 dispatch_receiver.typeInfo, foo.getDispatchReceiver.type)
for (arg_i in arguments) {
 s.addSubtypeConstraint(
 arg_i.typeInfo, foo.valueParameter_i.type)
}
for (fl in function_literal_arguments) {
 if (fl.isExplicitlyTyped)
 s.addSubtypeConstraint(
 fl.type, foo.valueParameter_for_fl.type)
 else
 s.addFunctionLiteralArgument(fl)
}
```

If the constraint system isn't incompatible (it'll be discussed later), we return SUCCESS status and type info:

```
TypeInfo(foo.getReturnType, dataFlowInfo, s)
```

Code: `CandidateResolver.performResolutionForCandidateCall()`.

Todo: Now function literals are not the part of the constraint system. Thus we can't defnitely say the right moment to analyze each literal (when all argument types are inferred), so we analyze some in `CandidateResolver.addConstraintForFunctionLiteral()` and complete analyzing all in `CallCompleter`.

Smart casts

If the receiver or the value argument has smart cast types, its type info is taken as the intersection of possible types. That means an intersection type may be added to the constraint system (which is a nondenotable type). But as it may be added only as a lower bound, the result (the supertype of lower bounds) will always be denotable.

Code: `CandidateResolver.addConstraintForValueArgument()`;

`updateResultTypeForSmartCasts()`

- Value Arguments to Parameters mapping

The complexity goes only when named arguments are mixed with positioned or some arguments with default values are omitted. The main rule is that positioned argument is always located at its position. Named arguments that go before the last positioned argument should be used at the right position as well. It might be useful to place named argument before positioned, for example to name a boolean flag. If the mapping isn't successful (there is unnecessary argument or a parameter without default value doesn't have a corresponding argument), a special error is generated. But the analysis doesn't stop (to be able to infer the resulting type and avoid generating too much red code).

Code: `ValueArgumentsToParametersMapper.`

Todo: Now we don't allow to write a positioned argument after a named one.

- Choosing the most specifc candidate

The informal intuition is that one candidate is more specifc than another if any invocation handled by the frst candidate could be passed on to the other one without a compile-time error.

Todo: In Java type inference is applied here (we might want the same approach).

Code: OverloadingConflictResolver.

- Solve constraint system

The reminder: the constraint system consists of:

- map: type variable → type bounds;
- references to function literal arguments;
- information about errors.

We have to find type substitution for each type variable that satisfes all its bounds. As variables may depend on each other, we have to find the right order to fix the value of a variable to avoid changing it later (that might make the process infnite). Let's emphasize several phases of the solving process (that can be repeated and go one after the other in any order):

- Adding new constraint and infating the system with all possible new bounds that can be derived from this new constraint and current bounds.
- Fixing type variable.
- Analyzing function literal argument body (that may add new type variables and constraints to our system).

Let's say that a variable α depends on a variable β if there is a type bound on α that contains β as a type-argument or is β itself.

A function literal `fl` of the type `(A1,··· An)-> R` depends on a variable β if its value argument type `Ai` mentions β . A variable α depends on a function literal `fl` if the result type `R` mentions α. With "depends on" relation we can divide our set of variables and function literals into connected components. If there is a relation " α is subtype of β ", then variables α and β are in the same connected component, because they depend on each other. Let's divide some components even more to reach the state when the component is either a set of variables, either a function literal. To achieve this, we cut the dependencies of the last function literal in the component containing both variables and function literals.

The example:

```
fun <T, R> foo(···, f: (T)->R, g: (R)->T)
```

For the concrete invocation

```
foo(⋯, { x -> ⋯ }, { y -> ⋯ })
```

we'll have the following dependencies:

```
T ← f ← R ← g ← T
```

There is one dependency component, but we'd like to divide it. Therefore we don't add the dependency for the function literal if it creates a connected component with both variables and function literals, in our case g ← T, and have: T ← f ← R ← g. Repeating this process we separate each function literal into its own connected component. Note that we generate constraints for the arguments in an order they are written in the code, so in an example above we say that the second function literal depends on return type of the frst one (not vice versa).

Another example:

```
fun <T>foo(f: (T)->T)
```

We don't add the last dependency f ← T, having only T ← f. Having done this, we can consequently fix the values of the type variables and analyze function literal bodies, adding new constraints to the system. The "expected type" constraint is the last one to add (all constraints from the value arguments we've added already).

```
fun solve(s: ConstraintSystem, expectedType) {
 s.addSupertypeConstraint(expectedType, foo.returnType)
 while (s.hasUnknownVariables()) {
 val e = s.getIndependentConnectedComponent()
 if (e is VariableSet) {
 fixVariables(e)
 }
 if (e is FunctionLiteral) {
 analyzeFunctionLiteralBody({ returnedExpression →
 s.addSubtypeConstraint(fl.type, returnedExpression.typeInfo)
 })
 }
 }
}
```

The easiest way to fix the variables that comes to mind is:

```
fun fixVariables(set: Set<Variable>) {
 for (v in set) {
 fixVariable(v)
 }
}
```

But this method won't work for some corner cases:

- variables: `U`, `V`, `W`

- constraints: `U <: V`, `V <: W`, `W >: Int`

We try to fix `U` first, and generate an error "not enough information to infer type variable value". But we could get this info generating the value for `W` beforehand. So fixing variables should work like this:

```
fun fixVariables(set: Set<Variable>) {
 for (v in set)
 fixWithDependent(v)
 for (v in set)
 if (!fixed(v))
 status ← error
 }
fun fixWithDependent(v) {
 if (hasNoProperTypeBounds(v) || fixed(v)) return
 fixVariable(v)
 for (u in v.dependentVariables)
 fixWithDependent(u)
 }
fun fixVariable(v)
```

Fixing a variable means fnding the type that satisfes all bounds on this variable. The result may depend on variance of type variable as well: subtype of all possible answers for covariant and invariant occurrences and supertype for contravariant (see the example below). Type bounds can be weak and strong. Weak bounds are derived only from type-parameter bounds, and all other bounds are strong. If a type variable has only weak bounds, we consider it's value as undefned and generate an error "not enough information to infer type-parameter". If a type variable has both weak and strong bounds, all of them are used to infer the result value. When we fx the variable, we try several suggestions and return the suggestion if it satisfes all bounds. The suggestions are:

- exact bound,

- supertype of lower bounds except number lower bounds,

- supertype of number lower bounds,

- supertype of all lower bounds,

- intersection of upper bounds (if it's denotable).

Here only proper bounds are considered, meaning only those which don't depend on other type

variables. The number lower bounds are processed separately, because an algorithm to fnd their supertype difers from the one for usual bounds. Let's consider two number value types `NVT(1)` and `NVT(10000000000)`. The supertypes of `NVT(1)` are `Int`, `Byte`, `Short`, `Long`. `NVT(10000000000)` has only `Long` as its supertype (the number is too large to be stored in `Int`). We have to take intersection of supertypes (`Long` in the example) to find the correct value.

Code: `TypeBoundsImpl.computeValues()`

For the type-parameter that can be found only in CONTRAVARIANT position in return type, the order of suggestions to try could be changed (try first intersection of upper bounds) to provide having the upper bound of possible answers as a result. That means maximal versus minimal type of the solution space. In the most cases for type-parameter in contravariant position maximal type equals minimum type, because only lower bounds are generated. It fails only with the type-parameter bounds:

```
fun foo<T: R>(T, Comparator<R>): Comparator<T>
interface A; interface B : A
foo(b, comparatorForA)
T >: B,
Comparator<R> >: Comparator<A> => R <: A => T <: A (with the use of 'T <: R')
```

Thus the solution space is {A, B}. Maximal type is `A`, and returning of `Comparator<A>` is better, because it's a more precise type. However, that means in the following case we may want to return more precise type as well:

```
fun bar<T: A>(T): Comparator<T>
bar(b)
```

But returning `Comparator<A>` may surprise one, because if `A` is just implicit `Any?`, to have always `Comparator<Any?>` is a bit odd. Note that using upper bounds in inference (not just checking them) is important for some cases (see KT-3372).

Note. (from discussion 03.03.15) Let's assume we use type information for one 'elvis' or 'if' branch from another branch (if we represent special constructions 'if', 'elvis', '!!' as calls, it's difcult not to do so). Then some unexpected behavior may occur like in KT-6694:

```
interface PsiElement {
 fun <T: PsiElement> findChildByType(i: Int): T?
}
interface JetSimpleNameExpression: PsiElement {
 open fun getReferencedNameElement(): PsiElement
}
interface JetLabelReferenceExpression : JetSimpleNameExpression {
 public override fun getReferencedNameElement() =
findChildByType(42) ?: this
}
```

Here the expression `findChildByType(42)` has the type `JetLabelReferenceExpression`, which can lead to class cast exception in runtime (see the comment in the task for more detailed example). The proposed solution is described below. It might also be noted that for now we'd like not to infer type-parameter without constraints on it, like (KT-5464):

```
val l = listOf() //compiler error
```

In Scala `List<Nothing>` is inferred for such a case. We defnitely want to infer type parameter if it's not used in return type (KT-2656), but there are doubts whether to infer `List<Nothing>` as a type of `l` variable or not. If we infer `Nothing` by default, the problem KT-6694 can be fixed by not using the type information for one branch from another, because the following case will work anyway: `if (true) listOf(1, 2, 3) else listOf()`. However in Scala it's not the case: information from another `if` branch is used for inference, in spite of inferring `Nothing` by default in many cases (see the comment to KT-6901). The proposed solution is to consider a constraint from expected type with the highest priority. For the example above there are two constraints:

```
T >: JetLabelReferenceExpression, T <: PsiElement
```

`PsiElement` can be inferred because it's a constraint from expected type. However if we use the information for one 'elvis' or 'if' branch from another branch, the following declarations will inevitably lead to class cast exception:

```
val x = findChildByType(42) ?: this //exception
```

With explicit variable type, we infer `PsiElement`, because it's the constraint from expected type and it has the highest priority:

```
val x: PsiElement = findChildByType(42) ?: this // ok
```

To support initial request (an example above), we should use a return type of overridden function as expected type for function body (see KT-6901), with the same highest priority as every expected type.

- Constraint system incorporation

TODO: integrate the description of the algorithm from another document

18. Threads and Concurrency

18.1. General

Not all platforms may provide multithreading. TODO

18.2. Memory Model

TODO

18.3. Race Conditions

TODO

18.4. Synchronization

TODO: synchronized annotation

18.5. Thread Creation and Termination

TODO

19. Java Interoperability

19.1. General

One of design goals of the Kotlin language is to provide a seamless interoperability with Java code, both when invoking Java code from Kotlin, and when invoking Kotlin code from Java. In this section we describe some details about calling Java code from Kotlin.

- Calling Java code from Kotlin

Most Java code can be used from Kotlin without any issues

```
import java.util.*

fun demo(source: List<Int>) {
  val list = ArrayList<Int>()
  // 'for'-loops work for Java collections:
  for (item in source)
    list.add(item)
  // Operator conventions work as well:
  for (i in 0..source.size() - 1)
    list[i] = source[i] // get and set are called
}
```

- Getters and Setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with `get` and single-argument methods with names starting with `set`) are represented as properties in Kotlin. For example:

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY        // call setFirstDayOfWeek()
    }
}
```

If the Java class only has a setter, it will not be visible as a property in Kotlin, because Kotlin does not support set-only properties.

• Methods returning void

If a Java method returns void, it will return `Unit` when called from Kotlin. If that return value is used, it will be assigned at the call site by the Kotlin compiler, since the value itself is known in advance (being object `Unit`).

• Escaping for Java identifiers that are keywords in Kotlin

Some of the Kotlin keywords are valid identifiers in Java: **in**, **object**, **is**, etc. If a Java library uses a Kotlin keyword for a method, it still can be called escaping it with the backtick (`` ` ``) character

```
foo.`is`(bar)
```

• Null-Safety and Platform Types

Any reference in Java may be **null**, which makes Kotlin's requirements of strict null-safety impractical for objects coming from Java. Types of Java declarations are treated specially in Kotlin and called **platform types**. Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java (see more [below](#mapped-types)).

Consider the following examples:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size() // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

When we call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```
item.substring(1) // allowed, may throw an exception if item == null
```

Platform types are **non-denotable**, meaning that one can not write them down explicitly in the

language. When a platform value is assigned to a Kotlin variable, we can rely on type inference (the variable will have an inferred platform type then, as `item` has in the example above), or we can choose the type that we expect (both nullable and non-null types are allowed):

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```

If we choose a non-null type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-null variables from holding nulls. Assertions are also emitted when we pass platform values to Kotlin functions expecting non-null values etc. Overall, the compiler does its best to prevent nulls from propagating far through the program (although sometimes this is impossible to eliminate entirely, because of generics).

- Notation for Platform Types

As mentioned above, platform types cannot be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (in error messages, parameter info etc), so we have a mnemonic notation for them:

- `T!` means "T` or `T?`",
- `(Mutable)Collection<T>!` means "Java collection of `T` may be mutable or not, may be nullable or not",
- `Array<(out) T>!` means "Java array of `T` (or a subtype of `T`), nullable or not"
- Nullability annotations

Java types which have nullability annotations are represented not as platform types, but as actual nullable or non-null Kotlin types. The compiler supports the JetBrains flavor of the nullability annotations (its description can be found at http://www.jetbrains.com/idea/help/nullable-and-notnull-annotations.html). TODO: copy relevant information here (`@Nullable` and `@NotNull` from the `org.jetbrains.annotations` package).

- Mapped types

Kotlin treats some Java types specially. Such types are not loaded from Java "as is", but are *mapped* to corresponding Kotlin types. The mapping only matters at compile time, the runtime representation remains unchanged. Java's primitive types are mapped to corresponding Kotlin types (keeping [platform types](#platform-types) in mind):

129

```
| **Java type** | **Kotlin type**  |
|---------------|------------------|
| `byte`        | `kotlin.Byte`    |
| `short`       | `kotlin.Short`   |
| `int`         | `kotlin.Int`     |
| `long`        | `kotlin.Long`    |
| `char`        | `kotlin.Char`    |
| `float`       | `kotlin.Float`   |
| `double`      | `kotlin.Double`  |
| `boolean`     | `kotlin.Boolean` |
```

Some non-primitive built-in classes are also mapped:

```
| **Java type** | **Kotlin type**  |
|---------------|------------------|
| `java.lang.Object`       | `kotlin.Any!`           |
| `java.lang.Cloneable`    | `kotlin.Cloneable!`     |
| `java.lang.Comparable`   | `kotlin.Comparable!`    |
| `java.lang.Enum`         | `kotlin.Enum!`          |
| `java.lang.Annotation`   | `kotlin.Annotation!`    |
| `java.lang.Deprecated`   | `kotlin.Deprecated!`    |
| `java.lang.Void`         | `kotlin.Nothing!`       |
| `java.lang.CharSequence` | `kotlin.CharSequence!`  |
| `java.lang.String`       | `kotlin.String!`        |
| `java.lang.Number`       | `kotlin.Number!`        |
| `java.lang.Throwable`    | `kotlin.Throwable!`     |
```

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package `kotlin`):

```
| **Java type** | **Kotlin read-only type**  | **Kotlin mutable type** | **Loaded
platform type** |
|---------------|------------------|----|----|
| `Iterator<T>`        | `Iterator<T>`        | `MutableIterator<T>`           |
`(Mutable)Iterator<T>!`          |
| `Iterable<T>`        | `Iterable<T>`        | `MutableIterable<T>`           |
`(Mutable)Iterable<T>!`          |
| `Collection<T>`      | `Collection<T>`      | `MutableCollection<T>`           |
`(Mutable)Collection<T>!`          |
| `Set<T>`            | `Set<T>`            | `MutableSet<T>`           |
`(Mutable)Set<T>!`          |
| `List<T>`           | `List<T>`           | `MutableList<T>`           |
`(Mutable)List<T>!`          |
| `ListIterator<T>`    | `ListIterator<T>`    | `MutableListIterator<T>`           |
`(Mutable)ListIterator<T>!`          |
| `Map<K, V>`          | `Map<K, V>`          | `MutableMap<K, V>`           |
`(Mutable)Map<K, V>!`          |
| `Map.Entry<K, V>`    | `Map.Entry<K, V>`    | `MutableMap.MutableEntry<K,V>` |
`(Mutable)Map.(Mutable)Entry<K, V>!` |
```

Java's arrays are mapped as mentioned below:

```
| **Java type** | **Kotlin type**  |
|---------------|------------------|
| `int[]`       | `kotlin.IntArray!` |
| `String[]`    | `kotlin.Array<(out) String>!` |
```

- Java generics in Kotlin

Kotlin's generics are a little different from Java's. When importing Java types to Kotlin we perform some conversions:

- Java's wildcards are converted into type projections
- `Foo<? extends Bar>` becomes `Foo<out Bar!>!`
- `Foo<? super Bar>` becomes `Foo<in Bar!>!`
- Java's raw types are converted into star projections
- `List` becomes `List<*>!`, i.e. `List<out Any?>!`

Like Java's, Kotlin's generics are not retained at runtime, i.e. objects do not carry information about actual type-arguments passed to their constructors, i.e. `ArrayList<Integer>()` is indistinguishable from `ArrayList<Character>()`. This makes it impossible to perform **is**-checks that take generics into account. Kotlin only allows **is**-checks for star-projected generic types:

```
if (a is List<Int>) // Error: cannot check if it is really a List of Ints
// but
if (a is List<*>) // OK: no guarantees about the contents of the list
```

- Java Arrays

Arrays in Kotlin are invariant, unlike Java. This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure. Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed (though [platform types](#platform-types) of the form `Array<(out) String>!`).

Arrays are used with primitive datatypes on the Java platform to avoid the cost of boxing/unboxing operations. As Kotlin hides those implementation details, a workaround is required to interface with Java code. There are specialized classes for every type of primitive array (`IntArray`, `DoubleArray`, `CharArray`, and so on) to handle this case. They are not related to the `Array` class and are compiled down to Java's primitive arrays for maximum performance.

Suppose there is a Java method that accepts an int array of indices:

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

To pass an array of primitive values one can do the following in Kotlin:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // passes int[] to method
```

Java classes sometimes use a method declaration for the indices with a variable number of arguments (varargs).

```
public class JavaArrayExample {

    public void removeIndices(int... indices) {
        // code here...
    }
}
```

In that case the spread operator * can be used to pass the `IntArray`:

```
val javaObj = JavaArray()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

It is not possible to pass **null** to a method that is declared as varargs.

When compiling to JVM byte codes, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = array(1, 2, 3, 4)
array[x] = array[x] * 2 // no actual calls to get() and set() generated
for (x in array) // no iterator created
  print(x)
```

Even when we navigate with an index, it does not introduce any overhead

```
for (i in array.indices) // no iterator created
  array[i] += 2
```

Finally, **in**-checks have no overhead either

```
if (i in array.indices) { // same as (i >= 0 && i < array.size)
  print(array[i])
}
```

- Operators

Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (`invoke()` etc.). Calling Java methods using the infix call syntax is not allowed.

- Checked Exceptions

In Kotlin, all exceptions are unchecked, meaning that there is no requirement to catch any of them. So, when a Java method that declares a checked exception is called from Kotlin, a try/catch statement is not required:

```
fun render(list: List<*>, to: Appendable) {
  for (item in list)
    to.append(item.toString()) // Java would require us to catch IOException here
}
```

- Object Methods

When Java types are imported into Kotlin, all the references of the type `java.lang.Object` are turned

into `Any`. Since `Any` is not platform-specific, it only declares `toString()`, `hashCode()` and `equals()` as its members, so to make other members of `java.lang.Object` available, Kotlin uses extension functions.

19.1.1. wait()/notify()

Methods wait() and notify() declared in java.lang.Object are not available on references of type `Any`.

[Rationale: [Effective Java](http://www.oracle.com/technetwork/java/effectivejava-136174.html) Item 69 recommends to prefer concurrency utilities to `wait()` and `notify()`. End rationale] [Note: to workaround this restriction, an object can be explicitly cast to `java.lang.Object`:

```
(foo as java.lang.Object).wait()
```

End Note]

- getClass()

To retrieve the type information from an object, we use the javaClass extension property.

```
val fooClass = foo.javaClass
```

Instead of Java's `Foo.class` use Foo::class.java.

```
val fooClass = Foo::class.java
```

- clone()

To override `clone()`, a class needs to extend `kotlin.Cloneable`:

```
class Example : Cloneable {
  override fun clone(): Any { ... }
}
```

- finalize()

To implement a finalizer, a class has to declare a protected method named `finalize()`, without using the **override**{:.keyword} keyword:

```
class C {
  protected fun finalize() {
    // finalization logic
  }
}
```

- Inheritance from Java classes

At most one Java-class (and arbitrary number of Java interfaces) can be a supertype for a class in Kotlin.

- Accessing static members

Static members of Java classes form "companion objects" for these classes. We cannot pass such a "companion object" around as a value, but can access the members explicitly, for example

```
if (Character.isLetter(a)) {
  // ...
}
```

- Java Reflection

Java reflection works on Kotlin classes and vice versa. Expressions `instance.javaClass` or `ClassName::class.java` can be used to to enter Java reflection through `java.lang.Class`.

Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, a `KProperty` for a Java field, a Java method or constructor for a `KFunction` and vice versa.

- SAM Conversions

Just like Java 8, Kotlin supports SAM conversions. This means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

This can be used for creating instances of SAM interfaces:

```
val runnable = Runnable { println("This runs in a runnable") }
```

...and in method calls:

```
val executor = ThreadPoolExecutor()
// Java signature: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

If the Java class has multiple methods taking functional interfaces, a particular one can be choosed by using an adapter function that converts a lambda to a specific SAM type. Those adapter functions are also generated by the compiler when needed.

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

Note that SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Also note that this feature works only for Java interop; since Kotlin has proper function types, automatic conversion of functions into implementations of Kotlin interfaces is unnecessary and therefore unsupported.

- Calling Kotlin code from Java

Kotlin code can be called from Java easily.

- Properties

Property getters are turned into **get**-methods, and setters – into **set**-methods.

- Package-Level Functions

All the functions and properties declared in a file `example.kt` inside a package `org.foo.bar` are put into a Java class named `org.foo.bar.ExampleKt`.

```
// example.kt
package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

The name of the generated Java class can be changed using the `@JvmName` annotation:

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

Having multiple files which have the same generated Java class name (the same package and the same name or the same @JvmName annotation) is normally an error. However, the compiler has

the ability to generate a single Java facade class which has the specified name and contains all the declarations from all the files which have that name. To enable the generation of such a facade, use the @JvmMultifileClass annotation in all of the files.

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() {
}
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() {
}
```

```
// Java
demo.Utils.foo();
demo.Utils.bar();
```

- Fields

To expose a Kotlin property as a field in Java, it has to be annotates with the `@JvmField` annotation. The field will have the same visibility as the underlying property. A property with `@JvmField` can be created if it has a backing field, is not private, does not have `open`, `override` or `const` modifiers, and is not a delegated property.

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

- Static Methods and Fields

As mentioned above, Kotlin generates static methods for package-level functions. On top of that, it also generates static methods for functions defined in named objects or companion objects of classes and annotated as `@JvmStatic`. For example:

```
class C {
  companion object {
    @JvmStatic fun foo() {}
    fun bar() {}
  }
}
```

Now, `foo()` is static in Java, while `bar()` is not:

```
C.foo(); // works fine
C.bar(); // error: not a static method
```

Same for named objects:

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

In Java:

```
Obj.foo(); // works fine
Obj.bar(); // error
Obj.INSTANCE.bar(); // works, a call through the singleton instance
Obj.INSTANCE.foo(); // works too
```

Also, public properties defined in objects and companion objects, as well as top-level properties annotated with `const`, are turned into static fields in Java:

```
// file example.kt

object Obj {
  val CONST = 1
}

const val MAX = 239
```

In Java:

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
```

- Handling signature clashes with @JvmName

Sometimes we have a named function in Kotlin, for which we need a different JVM name the byte code. The most prominent example happens due to **type erasure**:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

These two functions can not be defined side-by-side, because their JVM signatures are the same: filterValid(Ljava/util/List;)Ljava/util/List;. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with @JvmName and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

From Kotlin they will be accessible by the same name filterValid, but from Java it will be filterValid and filterValidInt.

The same trick applies when we need to have a property x alongside with a function getX():

```
val x: Int
  @JvmName("getX_prop")
  get() = 15

fun getX() = 10
```

- Overloads Generation

Normally, when a Kotlin method is declared with default parameter values, it will be visible in Java only as a full signature, with all parameters present. To expose multiple overloads to Java callers, the @JvmOverloads annotation can be used:

```
@JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
    ...
}
```

For every parameter with a default value, this will generate one additional overload, which has this parameter and all parameters to the right of it in the parameter list removed. In this example, the following methods will be generated:

```
// Java
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

The annotation also works for constructors, static methods etc. It can't be used on abstract methods, including methods defined in interfaces.

Note that if a class has default values for all constructor parameters, a public no-argument constructor will be generated for it. This works even if the @JvmOverloads annotation is not specified.

- Checked Exceptions

Kotlin does not have checked exceptions. So, normally, the Java signatures of Kotlin functions do not declare exceptions thrown. Thus if a function is declared in Kotlin like this:

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

Consider a scenario when it is necessary to call it from Java and catch the exception:

```
// Java
try {
    demo.Example.foo();
}
catch (IOException e) { // error: foo() does not declare IOException in the throws
list
    // ...
}
```

This code will result in an error message from the Java compiler, because `foo()` does not declare `IOException`. To work around this problem, the `@Throws` annotation in Kotlin can be used:

```
@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

- Null-safety

When calling Kotlin functions from Java, there is no mechanism to prevent Java code from passing

null as a non-null parameter. Therefore, Kotlin generates runtime checks for all public functions that expect non-nulls. In case a null argument is passed, the `NullPointerException` will be thrown immediately upon entry to the function, before any statements in it are executed.

Generic types whose type-arguments is Nothing are represented as raw type in Java. TODO: what happens if arity > 1?

- Function Types in Kotlin on JVM

- Brief overview

- Treat extension functions almost like non-extension functions with one extra parameter, allowing to use them almost interchangeably.

- Introduce a physical class `Function` and unlimited number of **fictitious** (synthetic) classes `Function0`, `Function1`, … in the compiler front-end

- On JVM, introduce `Function0`..`Function22`, which are optimized in a certain way, and `FunctionN` for functions with 23+ parameters. When passing a lambda to Kotlin from Java, one will need to implement one of these interfaces.

- Also on JVM (under the hood) add abstract `FunctionImpl` which implements all of `Function0`..`Function22` and `FunctionN` (throwing exceptions), and which knows its arity. Kotlin lambdas are translated to subclasses of this abstract class, passing the correct arity to the super constructor.

- Provide a way to get arity of an arbitrary `Function` object (pretty straightforward).

- Hack `is`/`as Function5` on any numbered function in codegen (and probably `KClass.cast()` in reflection) to check against `Function` and its arity.

- Extension functions

Extension function type `T.(P) → R` is now a shorthand for `@kotlin.extension Function2<T, P, R>`. `kotlin.extension` is a **type annotation** defined in built-ins. So effectively functions and extension functions now have the same type, which means that everything which takes a function will work with an extension function and vice versa.

To avoid ambiguities, the following restrictions are in effect:

- A value of an extension function type cannot be called as a non-extension function, and a value of a non-extension function type cannot be called as an extension. This requires an additional diagnostic which only occurs when a call is resolved to the `invoke` with the wrong extension-ness.

- Shape of a function literal argument or a function expression must exactly match the extension-ness of the corresponding parameter. An extension function literal or an extension function expression cannot be passed where a function is expected and vice versa. [Note: To workaround this restriction, it is possible to change the shape, assign literal to a variable or use the `as` operator. End Note]

So, it is possible safely coerce values between function and extension function types, but they still have to be invoked in a way which specified in their type (with or without `@Extension`).

- `Function0`, `Function1`, … types

The arity of the functional interface is not limited by the language, but can be limited by the platform. *[Example:* a typical limitation on JVM is 255 on JVM. *End example]*

These interfaces are named `Function0<R>`, `Function1<P0, R>`, ..., `Function42<P0, P1, ···, P41, R>`, ... in the package `kotlin`. They are **fictitious**, which means they have no sources and no runtime representation. Type checker creates the corresponding descriptors on demand, IDE creates corresponding source files on demand as well. Each of them inherits from `kotlin.Function` (described below) and contains only two functions, both of which are synthetically produced by the compiler:

- (declaration) `invoke` with no receiver, with the corresponding number of parameters and return type.

- (synthesized) `invoke` with first type-parameter as the extension receiver type, and the rest as parameters and return type.

Call resolution shall use the annotations on the type of the receiver of the call to select the correct `invoke` and to report the diagnostic if the `invoke` is ill-formed (see the previous block).

On JVM function types are erased to the physical classes defined in package `kotlin.jvm.internal`: `Function0`, `Function1`, ..., `Function22`, and `FunctionN` for 23 parameters and above.

- Function interface

There is also the empty interface `kotlin.Function<R>` which is a supertype for all functions.

```
package kotlin

interface Function<out R>
```

It is a physical interface, declared in platform-agnostic built-ins. [Example: On JVM it is typically located in `kotlin-runtime.jar`. End example]. However its declaration is empty. [Rationale: Because every physical JVM function class `Function0`, `Function1`, ... inherits from it (and adds `invoke()`), it is desirable that a user is not required to provide overrides for anything beside `invoke()` when doing it from Java code. End rationale]

- Functions with 0..22 parameters at runtime

There are 23 function interfaces in `kotlin.jvm.functions`: `Function0`, `Function1`, ..., `Function22`. Here's `Function1` declaration, for example:

```
package kotlin.jvm.functions

interface Function1<in P1, out R> : kotlin.Function<R> {
    fun invoke(p1: P1): R
}
```

These interfaces are supposed to be inherited from by Java classes when passing lambdas to Kotlin. They shouldn't be used from Kotlin however, because normally one would use a function type

there, most of the time even without mentioning built-in function classes: (P1, P2, P3) → R.

- Translation of Kotlin lambdas

There's also `FunctionImpl` abstract class at runtime which helps in implementing `arity` and vararg-invocation. It inherits from all the physical function classes, unfortunately (more on that later).

```
package kotlin.jvm.internal;

// This class is implemented in Java because supertypes need to be raw classes
// for reflection to pick up correct generic signatures for inheritors
public abstract class FunctionImpl implements
    Function0, Function1, ..., ..., Function22,
    FunctionN   // See the next section on FunctionN
{
    public abstract int getArity();

    @Override
    public Object invoke() {
        // Default implementations of all "invoke"s invoke "invokeVararg"
        // This is needed for KFunctionImpl (see below)
        assert getArity() == 0;
        return invokeVararg();
    }

    @Override
    public Object invoke(Object p1) {
        assert getArity() == 1;
        return invokeVararg(p1);
    }

    ...
    @Override
    public Object invoke(Object p1, ..., Object p22) { ... }

    @Override
    public Object invokeVararg(Object... args) {
        throw new UnsupportedOperationException();
    }

    @Override
    public String toString() {
        // Some calculation involving generic runtime signatures
        ...
    }
}
```

Each lambda is compiled to an anonymous class which inherits from `FunctionImpl` and implements the corresponding `invoke`:

```
{ (s: String): Int -> s.length }

// is translated to

object : FunctionImpl(), Function1<String, Int> {
    override fun getArity(): Int = 1

    /* bridge */ fun invoke(p1: Any?): Any? = ...
    override fun invoke(p1: String): Int = p1.length
}
```

- Functions with more than 22 parameters at runtime

To support functions with many parameters there's a special interface in JVM runtime:

```
package kotlin.jvm.functions

interface FunctionN<out R> : kotlin.Function<R> {
    val arity: Int
    fun invokeVararg(vararg p: Any?): R
}
```

TODO: usual hierarchy problems: there are no such members in `kotlin.Function42` (it only has `invoke()`), so inheritance from `Function42` will need to be hacked somehow

And another type annotation:

```
package kotlin.jvm.functions

annotation class arity(val value: Int)
```

A lambda type with 42 parameters on JVM is translated to `@arity(42) FunctionN`. A lambda is compiled to an anonymous class which overrides `invokeVararg()` instead of `invoke()`:

```
object : FunctionImpl() {
    override fun getArity(): Int = 42

    override fun invokeVararg(vararg p: Any?): Any? { ... /* code */ }
    // TODO: maybe assert that p's size is 42 in the beginning of invokeVararg?
}
```

Note that `Function0`..`Function22` are provided primarily for **Java interoperability** and as an **optimization** for frequently used functions.

So when a large function is passed from Java to Kotlin, the object will need to inherit from `FunctionN`:

```
    // Kotlin
    fun fooBar(f: Function42<*,*,...,*>) = f(...)
```

```
    // Java
    fooBar(new FunctionN<String>() {
        @Override
        public int getArity() { return 42; }

        @Override
        public String invokeVararg(Object... p) { return "42"; }
    }
```

Note that `@arity(N) FunctionN<R>` coming from Java code will be treated as `(Any?, Any?, ···, Any?)` → `R`, where the number of parameters is `N`. If there's no `@arity` annotation on the type `FunctionN<R>`, it won't be loaded as a function type, but rather as just a classifier type with an argument.

- Arity and invocation with vararg

There's an ability to get an arity of a function object and call it with variable number of arguments, provided by extensions in **platform-agnostic** built-ins.

```
package kotlin

@intrinsic val Function<*>.arity: Int
@intrinsic fun <R> Function<R>.invokeVararg(vararg p: Any?): R
```

But they don't have any implementation there. The reason is, they need **platform-specific** function implementation to work efficiently. This is the JVM implementation of the `arity` intrinsic (`invokeVararg` is essentially the same):

```
fun Function<*>.calculateArity(): Int {
    return if (function is FunctionImpl) {  // This handles the case of lambdas
created from Kotlin
        function.arity  // Note the smart cast
    }
    else when (function) {  // This handles all other lambdas, i.e. created from Java
        is Function0 -> 0
        is Function1 -> 1
        ...
        is Function22 -> 22
        is FunctionN -> function.arity  // Note the smart cast
        else -> throw UnsupportedOperationException()  // TODO: maybe do something
funny here,
                                                        // e.g. find 'invoke'
reflectively
    }
}
```

- is/as

The newly introduced `FunctionImpl` class inherits from all the `Function0`, `Function1`, ..., `FunctionN`. This means that `anyLambda is Function2<*, *, *>` will be true for any Kotlin lambda.

```
package kotlin.jvm.internal

// This is the intrinsic implementation
// Calls to this function are generated by codegen on 'is' against a function type
fun isFunctionWithArity(x: Any?, n: Int): Boolean = (x as? Function).arity == n
```

[Note: `as` should check if `isFunctionWithArity(instance, arity)`, and checkcast if it is or throw exception if not. A downside is that `instanceof Function5` obviously won't work correctly from Java. *End Note]*

Also a warning is issued on `is Array<Function2<*, , *>>` and `as Array<Function2<*, *, *>>`. [Rationale: There is no reasonable way how it could work for empty arrays, because there is no single instance of `FunctionImpl` to reach out and ask the arity. End rationale]

- How this will help reflection

`KFunction*` interfaces are synthesized at compile-time identically to functions. The compiler shall resolve `KFunction{N}` for any `N`, IDEs should synthesize sources when needed, `is/as` should be handled similarly etc.

However, at runtime there will not be multitudes of `KFunction`s.

So for reflection there will be:

- **fictitious** interfaces `KFunction0`, `KFunction1`, ..., `KFunction42`, ... (defined in `kotlin.reflect`)

- **physical** interface `KFunction` (defined in `kotlin.reflect`)
- **physical** JVM runtime implementation class `KFunctionImpl` (defined in `kotlin.reflect.jvm.internal`)

As an example, `KFunction1` is a fictitious interface (in much the same manner that `Function1` is) which inherits from `Function1` and `KFunction`. The former lets one call a type-safe `invoke` on a callable reference, and the latter allows to use reflection features on the callable reference.

```
fun foo(s: String) {}

fun test() {
    ::foo.invoke("")  // ok, calls Function1.invoke
    ::foo.name        // ok, calls KFunction.name
}
```

19.2. Mixed Projects

A Kotlin compiler is able to co-operate with a Java compiles to enable building of mixed projects, where Java source files and Kotlin source files co-exist and are able to reference each other.

TODO

19.3. Platform Types

Because Java lacks built-in support for non-nullable types, some types exposed from Java to Kotlin need to be handled in a special way.

TODO: Flexible types.

19.4. SAM Types

A SAM type is an abstract class or interface with a single abstract method (SAM stands for "Single Abstract Method"). TODO: Conversions from lambda expressions to SAM types.

19.5. Optional Parameters

On the JVM, if all of the parameters of the primary constructor have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors.

19.6. Using predefined types from java.lang package

In general, a predefined type of Java language (e.g. `java.lang.Object` or `java.lang.Integer`) should not be used in Kotlin code. The Kotlin standard library provides conterparts for them (e.g. kotlin.Any or kotlin.Int) that should be used instead. The counterparts are not exact clones of their

Java prototypes. They may have different set of methods, generic types may declare variance on their type paramteres — they are designed to better fit Kotlin programming paradigm. The compiler shall issue a warning if a predefined Java type is used rather than its Kotlin counterpart.

If a Java API is used from Kotlin, and it exposes some of those basic types, then signatures in that API are automatically adjusted to replace Java types with their Kotlin counterparts.

TODO

20. Reflection

TODO

21. Standard Library Overview

TODO

22. Documentation Comments

TODO

23. Miscellaneous

ALL RULES FROM THIS SECTION SHOULD BE MOVED TO OTHER SECTIONS WHERE THEY BELONG

An interface method without a body can be marked abstract, but this is redundant. Property initializers are not allowed in interfaces. Override member must have the same or wider visibility as the overridden member. Annotations on type-parameters are not supported yet. Spread operator – can pass an existing array, can join to create a new array. Syntax for floating-point literals (including hex), round to zero, round to infinity, suffixes. Generic properties. Escape \$ Capitalizing first letter of a property in its accessors' names. Within default value for an optional parameter, the parameter is not definitely assigned (TODO: except within lambdas). Visibility of generated componentN() methods in data classes is the same as visibility of the corresponding property. Block cannot contains property declaration with setters, extension property declaration, or generic property declaration. It is an error if right operand of / or % operator for any integral type is constant expression with zero value. Operators / and % for any integral type throw ArithmeticException if right operand is zero at runtime. Arithmetic operators overflow silently. Operator > implements strict linear order on integral types. m<n is equivalent to n>m, m⇐n is equivalent to m<n || m==n, m>=n is equivalent to m>n || m==n for integral types and type Char. The type Char supports binary operators +, - with the right operand of type Int, relational operators >,<,>=,⇐. Every type (including nullable types) supports operators ==, !=, ===, !==, they are not mentioned specifically for each type. NaN != NaN, if both operands of the same non-nullable floating-point type (so reflexivity and trichotomy properties for equality and comparison operators are violated). This does not apply for method equals invocations. @external annotation is only allowed for top-level functions. An enum constructor cannot explicitly delegate to super . return expression are not allowed in functions with expression body. return expression without label

return from nearest enclosing named function or anonymous function (TODO: it is more complicated). Anonymous function can use expression body. Type-parameter list is not allowed for anonymous functions. A package name cannot be followed with `?..`. A class name cannot be followed with `?.` and a nested class name. { ... } is parsed as block or as function literal depending on context (e.g. function body, class body, after `for`, `if`, `else`, `do`, `while`, `try`, `catch`, `finally`, after → in a when branch). To specify a function literal in a context where { ... } would be parsed as a block, it can be parenthesized ({ ... }) or explicit parameter list can be provided { → ... } (even if it's empty). (TODO: block/f. literal can be preceded by annotations and at most one label; providing several labels can force it to be parsed as f. literal) TODO: empty or semicolon branches in if/else An object declaration introduces both a type name and an object name. `else` branch must be the last branch in `when` expression. Exhaustive when check. Soft keywords: abstract annotation by catch companion constructor dynamic enum file final finally get import init inner internal open out override private protected public reified sealed set vararg where http://youtrack.jetbrains.com/issue/KT-2877 Relax this rule, but take care about identifier not supported by the platform. Space?

Left names in dotted names can be merged symbols (even of different kinds, e.g. class and package), whose eventual meaning is determined by which names to the right are looked up in them.

Overloading is possible based on constraints only, provided that erased signatures are different.

To access a member of an enclosing class using a simple name it is necessary to have an access to `this` instance of an enclosing class. In some contexts (e.g. within object declarations) `this` instances of enclosing classes are not available, and hence an access to a member of an enclosing class is not possible using a simple name (unless there is an implicit receiver of the corresponding type introduced by other means, e.g. through an extension function, or by deriving the object from an enclosing class).

@synchronized and default parameter values @synchronized and inline annotations on annotation parameters priority list for annotation targets: param, field, property, setter, getter investigate: non-nullable types T! investigate: import com.acme.A.foo where A is an object

import directives can import functional members only if they are top-level. Secondary constructors cannot have val/var keywords in parameter declarations. Recursive generic constraints. annotations on init blocks. Nested classes are not inherited, and need to be qualified with their declaring type when references from typed derived form the declaring type (TODO: consider enabling inheritance for inner classes, or an equivalent to bring their constructors in scope in derived classes).

initializer block can have annotations with EXPRESSION target.

TODO: Do we support inheritance from inner classes outside of their declaring classes, and do we have any syntax similar to Java qualified superclass constructor invocation (JLS 8, §8.8.7.1 Explicit Constructor Invocations)?

At most 1 parameter of a function can be vararg, but not necessarily the last one.

```
interface A : C { // Error
    public interface B {}
}

interface C : A.B { }
```

It is not allowed to manually inherit from Enum<E> type (TODO: implement).

Do we need `constructor` for primary constructors?

what is a non-empty package (in binary form and in source form)? no backtracking not including the last identifier try to find visible type with next identifier we depend on order of dependencies, and select the 1st type all except last identifier resolve to a type all import * directives create all unified scope enclosing the file scope import * may import ambiguous members import * may import nothing inner classes from generic classes? package members do not include subpackages package members include type, props, funs (including extensions) TODO: import from singletons?

import A — TODO: error? Do we resolve simple names to top-level symbols in root package? Repeat steps from the last case. 2 imports ends on the same identifier - error if class found in 2 modules - select one, and import its constructors all props and funs imported all with the same priority

ambiguous classes in type position? fqn vs. canonical in java stop package at generic arguments

TODO: members from default package

How do we place annotations for the element type int in the array type int[]?

If overriding member does not specify visibility and all overridden members have the same visibility (or there is only one overridden member), then the overriding method inherits that visibility. If not all overridden members have the same visibility, then the overriding method must specify it visibility explicitly, otherwise a compile-time error occurs. (TODO: verify this)

23.1. Import Directive Priorities

Import directives priorities (from highest to lowest):

1. Explicit user imports

2. Current package symbols

3. Top-level packages

4. Explicit default imports

5. User imports with "*"

6. Default imports with "*"

23.2. Anonymous Types in Public API

Methods and properties can return anonymous type originating from object literals, or generic

types constructed with anonymous types. These type are not denotable. In Java interop scenarios anonymous types are replaced with their first declared supertype, or Any if no supertype is specified. It potentially can result in type safety violations and heap pollution if, for example, Java overrides a method returning an anonymous type, and because the override return type is Any, it can return an object of unrelated type, that will be visible in Kotlin as an object of the anonymous types. Because JVM enforces type safety on a lower level, such scenarios will usually eventually result in CastClassException or ArrayStoreException, but it can occur in a distant position in code and may be difficult to debug.

23.3. Scripting

TODO

Importing a script means running it (if it has not been run before) and importing its symbols into the current scope. Imports may be allowed not only on the top of the file, but in other places of the program in the program. Script is similar to normal sequence of top-level declarations as they appear on top-level in a file, but may also contain executable statements between them.

23.4. Misc

TODO: `@setparam:`

The declared visibility of an override must be not less than the declared visibility of the overridden member (TODO: declared includes default).

Local functions and local classes cannot have visibility modifiers.

- 9.4.1.3 Inheriting Methods with Override-Equivalent Signatures
- `operator` is inherited (at least one inheritance path is enough)
- TODO: parameter names in functional types, no vararg, no duplicate parameter names in functional types, do we resolve to them in annotations?
- local functions can be operators
- TODO: no non-public set in interfaces
- TODO: named arguments are not allowed for function types (both in () and .invoke() invocations)
- TODO: top-level non-extension invoke cannot be operator (packageName() is not allowed)
- TODO: x[i] is not translated to x.get.invoke(), because property get cannot be operator; function parameter or local cannot be operator
- TODO: super.foo cannot refer to abstract member
- TODO: arguments can be renamed in overrides, invocation uses name from static type of the target
- TODO: an overriding function cannot provide default values for its parameters, they are always inherited. No more than 1 overridden declaration is allowed to provide a default value (even if values are identical)

- TODO: order of evaluation for constants: topo sorting, no cycles
- TODO: val property can be overridden by var property, but not vice versa
- TODO: no private abstract accessors in properties

A class can inherit multiple methods with the same signature, but unrelated return types, provided that it is abstract or overrides them with a method that is a subtype of all those return types (there is always at least one such type, namely Nothing).

When compiling to JVM, the name DefaultImpls in reserved in interfaces that provide default implementations for functional members. This name is used for nested class that is exposed to Java and contains the default implementations.

TODO: Spec DefaultImpls nested class, including representation of methods in generic interfaces (dependencies between type-parameters).

TODO: Spec `lateinit` properties (only var). There is no direct way to check if a lateinit property is already initialized (a workaround it to try to read from it and catch an exception), and there is no way to uninitialize it (set it to null to release a reference to an object). `kotlin.UninitializedPropertyAccessException`. The property type cannot be primitive or nullable type. TODO: Spec null safety and possible violations

TODO: Spec heap pollution

TODO: Do we copy annotation from default implementations in interfaces to corresponding classes?

TODO: Synchronized in interfaces?

TODO: Spec const vals

TODO: Generic anonymous objects / local classes, out-projections?

```
fun main(args: Array<String>) {
    fun f<T>(x : T) = object { val X = x };
    val s = listOf(f(1), f(""))
    val y = s[0].X
}
```

TODO: pre- and postfix increment/decrement operators

TODO: (x is Int?) Nullable mark is redundant here and results in a warning.

TODO: @KotlinOperator and @KotlinInfix annotations for Java interop

Properties declared in parameter declarations cannot be abstract or external.

Modifier `abstract` is not applicable to property accessors.

Cannot have constraints both in <...> and in where.

object instance is exposed to Java as INSTANCE field, for nested classed, also as class name

Only is Array<*>

It is a suppressable error if an override changes parameter name from a superclass.

TODO: precedence between synthetic and extension member (e.g. Runnable vs. Functional); discrimination levels? TODO: spec Runnable { ... }

TODO: vararg can declare Array<Xxx> or XxxArray.

TODO: It is an error, if return type of ++, — operators is not a subtype of their argument variable: [Example:

```
operator fun Any?.inc() : Any? = null

fun f() {
    var x = ""
    x++
}
```

End Example]

TODO: it is an error if a signature with vararg modifier overrides a signature without vararg modifier, or a signature without vararg modifier overrides a signature with vararg modifer. It is an error to inherit override-equivalent signatures that differ in vararg modifiers.

23.4.1. Method Overriding

[Examples

```
interface A {
    fun foo()
}

interface B : A // OK
```

The interface B inherits abstract method foo from its superinterface A.

```
interface A {
    fun foo() { println() }
}

interface B : A // OK
```

The interface B inherits concrete method foo from its superinterface A.

```
interface A {
    fun foo()
}

interface B {
    fun foo()
}

interface C : A, B // OK
```

The interface C inherits 2 abstract methods foo with the same signature from difference superinterfaces A and B.

```
interface A {
    fun foo() { println() }
}

interface B {
    fun foo()
}

interface C : A, B // ERROR
```

The interface C inherits multiple methods foo with the same signature from difference superinterfaces A and B, and there is at least 1 concrete method among them. This results in an error.

The error can be resolved by declaring in the interface C an abstract or concrete method foo with the same signature that overrides all inherited methods:

```
interface A {
    fun foo() { println() }
}

interface B {
    fun foo()
}

interface C : A, B {
    override fun foo(); // OK
}
```

```
interface A {
    fun foo() { println() }
}

interface B : A {
    override fun foo()
}

interface C {
    fun foo()
}

interface D : A, B // OK
```

The interface D has superinterfaces that declare multiple methods foo with the same signature, but all concrete methods are already overridden in the superinterfaces.

```
interface A {
    fun foo() { println() }
}

interface B {
    fun foo()
}

interface C : A, B {
    override fun foo() { println() }
}

interface D : A, B, C // OK
```

The interface D has superinterfaces that declare multiple methods foo with the same signature, but there is a single concrete method that already overrides all other methods.

```
interface A {
    fun foo(x: String)
}

interface B<T> : A {
    fun foo(x: T) { println() }
}

interface C : A, B<String> // OK
```

This is a more tricky example involving generics. Although the declaration of the method foo in B<T> does not override the method foo from A at its declaration, it nevertheless overrides it in the particular instantiation B<String> that is inherited by C, so the interface C does not inherit conflicting methods.

```
interface A {
    fun foo()
}

open class B {
    fun foo() { println() }
}

class C : B(), A // OK
```

Rules for inheritance from superclass are different. The class C inherits the single concrete method foo from its superclass, and also have a superinterface (or multiple superinterfaces) that declares abstract methods with the same signature. In this case no error occurs, and the concrete method from the superclass overrides all abstract methods from superinterfaces.

```
interface A {
    fun foo() { println() }
}

abstract class B {
    abstract fun foo()
}

abstract class C : B(), A // OK
```

In this case the class C inherits the abstract method from its superclass, and it also overrides all methods from superiterfaces (abstract or concrete). The class C must have abstract modifier, because it contains an abstract method.

```
interface A {
    fun foo() { println() }
}

abstract class B {
    abstract fun foo()
}

class C : B(), A // ERROR
```

In this case an error occurs, because the class C is not marked abstract.

```
interface A {
    fun foo()
}

abstract class B : A

interface C : A {
    override fun foo() {
        println("C.foo")
    }
}

class D : B(), C // OK
```

The class B inherits abstract method foo from the interface B. The interface C overrides that method with a concrete method foo. The class D does not inherited the abstract method foo from its superclass, because it is already overridden by its superinterface C. Instead, the class D inherits the implementation of foo from C.

End Examples]

@JvmField can be applied to properties with backing fields (in particular, non-extensions). Not in an interface companion object.

- TODO: infix decl restrictions: member/extension, 1 parameter, non-default, non-vararg
- TODO: sting intrepolation, last character can be $. Intepolation starts after unescaped $ followed by identifier, reserved keyword, full ⋯ identifier (not just ``) or {.
- TODO: no super for extension member invocations
- TODO: @JvmName is not applicable to non-final functions.

TODO: Locals win over members, even if members are declared closer to the usage (a class declaration can be nested within a method body). [Example:

```
fun foo(bar: Bar) = object {
    val bar = bar // The bar on the right-hand side refers to the parameter bar
}
```

End Example]

- TODO: intersection types and captured types leaking from (possibly nested) generic invocations to local variables.

- TODO: (possible generic) extension method invocations on numeric literals 3.foo()

- TODO: operands of compound assignments are evaluated only once.

- TODO: invokeExtension convention.

- TODO: conflict between nested constructor and synthetic valuesOf()

- TODO: `public fun foo() = if(true) this else null;` — ERROR. type is private anonymous

A function declaration can omit a specification of the return type, if the return type can be inferred from the function body, and the function body can be resolved without using any information about the return type. In particular, the return type specification cannot be omitted if the function is implemented recursively.

TODO: Comma-separated conditions are not allowed in `when` without argument. [Rationale: Commas can be confused with logical AND, but would mean OR if were allowed in this context. So, we require to use explicit `&&` or `||` operators if needed. End Rationale]

TODO: All token consisting of dots only (`.`, `..`, `···`, etc) are reserved. Currently only `.` and `..` can be used in valid programs. [Rationale: Without this rule expressions like `1···2` would be parsed as `1 .. .2` — that has been proven to be confusing for some users. It also looks plausible that the token `···` might be used in future versions of the language. End Rationale]

TODO: $ has special meaning in strings if both of the following are true: * it does not immediately follow an unescaped backslash * it is immediately followed by an identifier or keyword (in case the identifier is escaped, it must have opening and closing backtick and non-empty body.)

TODO: Java package-local can be used in signatures of internal members declared in Kotlin (https://youtrack.jetbrains.com/issue/KT-9623).

TODO: @JvmSuppressWildcards(true).

TODO: @JvmStatic for overrides?

TODO: Parameter name vs. property name in init blocks and property initializers TODO: Do not create raw type when Nothing TODO: Array's type-parameter is not reified anymore. What do we do with Array<Nothing>.

When applicable candidate members are compared for "betterness", for each explicitly provided argument, types of parameters corresponding to that argument are compared (irrespectively to whether they appear on the same position in method signatures). Type of parameters for which default values where automatically provided do not participate in "betterness" comparison.

Extension method with a receiver of numeric type is applicable on a literal receiver, even if the literal type does not match the received type exactly, but could be converted to it if it appeared in a regular argument position. E.g. fun Long.foo() can be applicable in 4.foo() invocation.

TODO: rename multi-declaration to desctructuring declarations, spec desctructuring declarations in for loops.

TODO: inline public API:

```
object X {
    private fun foo() {}
    public inline fun bar(x : () -> Unit)
    {
        foo() // Public-API inline function cannot access non-public-API 'private
final fun foo(): Unit defined in X'
    }
}
```

This rule also extends to any nested object literals and default values of parameters.

TODO: companion object inherirs from a class with companion object — it not transitive. TODO: equals(null) ⇒ true

```
fun f() : Unit {
    return 1 // Error
}
```

```
Out<In<Out<Open>>>>
```

- TODO: Scala rules for primary ctors scopes
- TODO: Accessibility constrains for delegated interfaces
- TODO: order of evaluation between delegation expression and superclass arguments

TODO: val properties must be assigned on all control-flow paths:

```
class C {
    val a: Int
    init {
        if(true) {
            a = 3
        }
        else {
            a = 5
        }
    }
}
```

If a class declares or inherits at least one abstract functional member, the class must be declared abstract.

23.4.2. Intersection types

An intersection type is a special type that is not syntactically denotable in the language, but can arise during typechecking or type inference for certain expressions. We will denote an intersection type as T1 & T2 & ⋯ & Tn in the specification (but note that this is not a proper language syntax).

A particular method to be invoked is determined not only by the method name specified in an invocation, but is also affected by two mechanisms: method overloading and method overriding. Method overloading is an ability to declare multiple methods with the same name in the same declaration space that differ by their signatures, and to select at compile-time a particular signature to invoke based on arguments specified in an invocation. Method overriding is an ability to provide multiple implementations for the same method signature at different levels of inheritance (or interface implementation) hierarchy, and to select at run-time a particular implementation to invoke bases on the exact run-time type of the object whose method is being invoked.

Method overloading happens an compile-time and consists of several steps as described below. First, a set of visible methods with the same name is formed by a name lookup. Visibility of a particular method is determined based on its effective visibility and the location is source where the method invocation occurs. Second, applicable

23.4.3. Priority of candidates

In general a method invocation expression is enclosed in a linear hierarchy of nested scopes, each of those can introduce methods, extension methods and implicit receivers. In addition, each level of the hierarchy can introduce multiple methods, overloaded by signature. When a method invocation has a form of a simple name followed by an argument list, the simple name can, in general, denote either a local function, a member function or an exension function (in the later case the extension function can be local or member). Before an overload resolution can be performed, a set of candidate functions has to be formed. Not all potentially applicable functions from all level are collected together. Instead, functions are grouped by their priority, and the groups are considered from highest priority to lowest, until a group that has at least one potentially applicable candidate is found. The hierarchy of priorities is not a simple copy of the hierarchy of nested scopes. Instead, it may contatin multiple subhierarchies, each of them reflecting the hierarchy of nested scopes. The

reason for that is an extension function invocation requires both a dispatch receiver and an extension receiver (TODO: check terminology) to be determined, and this may require multiple traversals through the hierarchy of scopes. TODO: fill details from https://jetbrains.quip.com/BfYcA0ITQ9FM

https://youtrack.jetbrains.com/issue/KT-11128 SAM-converted signatures are internally represented as extension methods on classes whose member their prototype is. So, they lose in overload resolution to members with less specific signatures. SAM conversion only happens to members declared in Java (not in Kotlin), so they never can be extensions by themselves. Generics methods are not SAM-convertible (CHECK).

TODO: https://youtrack.jetbrains.com/issue/KT-7052 (check corresponding Java rules).

"multi-declaration" ⇒ "destructuring"

[Note: In a situation when a method declared in Java has reference types in its signature, and needs to be overridden in Kotlin, in general it is not possible for the compiler to know whether any of those types were intended to have the `null` reference as one of their valid values. Still, this information may be known to the programmer from the documentation or by other means. So, the intended type in the overriding method's signatue may be either a non-nullable reference type or the corresponding nullable type. The programmer should be allowed to chose either of these alternative. Because the compiler cannot check the null-safety of the code in such cases, the responsibility for the correct choice lies with the programmer. End note] If a method declared in Java is overridden in Kotlin, and the original method declaration has reference types in its signature, then the overriding method is allowed to specify either non-nullable or nullable type at each occurrence of a reference type provided that, ignoring possible nullability, the specified type is the same type as the type in the corresponding position in the signature if the original method. If the method is further overridden in more derived types in Kotlin, those overriding methods must use the same types that where specified in the least derived override in written Kotlin.

TODO: discuss non-transitivity in subtyping in case { expr } ⇒ Unit ⇒ Unit?. See https://youtrack.jetbrains.com/issue/KT-7882

The instance of a companion object is available within the immediately enclosing type as an implicit receiver, but not as a qualified `this`. [Example: this@Companion.foo() is an error. End example]

Locals win. [Example:

```
fun bar(){
    fun foo() {
        class A {
            fun foo() {
                foo() // local foo() declared in top-level bar()
            }
        }
    }
}
```

End example]

A local variable can shadow a name from an outer scope. The shadowing does not occur in its own initializer.

A label that is an argument to a function invocation can use the function name as a label for a return expression in its body.

Safe call operator e?.foo() normally promotes the result type of foo() to the corresponding nullable type, but if the type of e is non-nullable type or e is known to be not null due to a smart-cast, then a warning is issued that the . operator could be used instead of ?., and the result type of foo() is not promoted to the nullable type.

Parameters of a function (method, constructor, standalone function or an anonymous function) are always immutable variables. Catch parameters are always immutable variables

Values captured into local class declaration at the point of its declaration, rather than in the point of construction. [Example:

```
fun main(args: Array<String>) {
    var obj : Any? = null
    for (x in arrayOf(1, 2, 3)) {
        class C() {
            fun foo() = x
        }

        if(obj is C){
            println("$x, ${obj.foo()}")
        }
        obj = C()
    }
}
```

This program prints:

```
2, 1
3, 2
```

End example]

TODO: Order of evaluation top-level property initializers across multiple source files, effects of `inline`: https://jetbrains.slack.com/archives/kotlin-team/p1463993052000897

An upper bound of a type parameter cannot be an array, or an array projection. [Example:

```
interface C<A, E> where A : Array<out E> { } // Error
```

End example]

- TODO: coroutine modifier. Parameter must be of type Controller.() → kotlin.coroutines.Coroutine<Unit>

- `coroutine` parameter of an `inline` function must be `noinline`

- `coroutine` parameter cannot be `vararg`

- A signature can contain multiple `coroutine` parameters

- A signature cannot contain more than 1 `coroutine` parameter

- A non-function expression can be passed to a `coroutine parameter`

- A controller can be any type, including nullable type

`Nothing` has the same members as `Any`. None of these members can be actually invoked at run-time, and any such invocation is an unreachable code at compile-time.

If a type S is a supertype of a type T, then the nullable type S? is a supertype of the nullable type T?. `Any` is a supertype of every class, object and interface type.

No class type except `Any` can be a supertype of an interface.

Subtyping relation is transitive: If a type S is a supertype of a type T, and the type T is a supertype of a type U, then the tyoe S is a supertype of the type U. Subtyping relation is irreflexive (strict): no type can be a supertype of itself. The corresponding non-strict reflexive relation is assignability. A type T is assignable to type S if either S is a supertype of T, or S is the same type as T.

TODO: define "sameness" for types. This is an equivalence relation, but unfortunately it is neither syntactically-based nor set-of-values-and-operations-on-them-based. The same type can have different syntactic forms. Two distinct types can have the same set of values and allow the same operations on those values.