Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 2

Type system

Glossary

T Type (with unknown nullability)

$T!!$ Non-nullable type

$T?$ Nullable type

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

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

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

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

Γ Type context

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

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

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

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

Type parameter
: Formal type parameter of a type constructor

Type argument
: Actual type argument in a parameterized type

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

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

substitution $\sigma : F_1 = A_1, \ldots, F_n = A_n$

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

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

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

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

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

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

GLB
Greatest lower bound

LUB
Least upper bound

Introduction

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

Kotlin has a type system with the following main properties.

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

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

Null safety is enforced by having two type universes: *nullable* (with nullable types $T?$) and *non-nullable* (with non-nullable types $T!!$). A value of any non-nullable

type cannot contain `null`, meaning all operations within the non-nullable type universe are safe w.r.t. empty values, i.e., should never result in a runtime error caused by `null`.

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

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

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

2.1 Type kinds

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

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

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

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

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

We also distinguish between *denotable* and *non-denotable* types. The former are types which are expressible in Kotlin and can be written by the end-user. The

latter are special types which are *not* expressible in Kotlin and are used by the compiler in type inference, smart casts, etc.

2.1.1 Built-in types

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

`kotlin.Any`

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

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

`kotlin.Nothing`

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

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

`kotlin.Function`

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

Built-in integer types

Kotlin supports the following signed integer types.

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

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

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

Array types

Kotlin arrays are represented as a parameterized type `kotlin.Array`(T), where T is the type of the stored elements, which supports `get`/`set` operations. The

`kotlin.Array`(T) type follows the rules of regular type constructors and parameterized types w.r.t. subtyping.

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

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

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

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

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

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

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

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

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

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

2.1.2 Classifier types

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

Simple classifier types

A simple classifier type

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

consists of

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

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

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

  Example:

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

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

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

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

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

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

Parameterized classifier types

A classifier type constructor

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

describes an abstract type and consists of

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

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

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

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

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

which consists of

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

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

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

  Example:

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

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

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

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


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


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

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

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

2.1.3 Type parameters

Type parameters are a special kind of types, which are introduced by type constructors. They are considered well-formed concrete types only in the type context of their declaring type constructor.

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

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

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

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

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

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

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

```
interface Foo
interface Bar

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

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

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

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

Function type parameters

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

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

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

Mixed-site variance

To implement subtyping between parameterized types, Kotlin uses *mixed-site variance* — a combination of declaration- and use-site variance, which is easier

to understand and reason about, compared to wildcards from Java. Mixed-site variance means you can specify, whether you want your parameterized type to be co-, contra- or invariant on some type parameter, both in type parameter (declaration-site) and type argument (use-site).

> Info: *variance* is a way of describing how subtyping works for *variant* parameterized types. With declaration-site variance, for two non-equivalent types $A <: B$, subtyping between `T<A>` and `T<B>` depends on the variance of type parameter F for some type constructor T.
>
> - if F is covariant (`out` F), `T<A>` $<:$ `T<B>`
> - if F is contravariant(`in` F), `T<A>` $:>$ `T<B>`
> - if F is invariant (default), `T<A>` $\not<:\not:>$ `T<B>`
>
> Use-site variance allows the user to change the type variance of an *invariant* type parameter by specifying it on the corresponding type argument. `out` A means covariant type argument, `in` A means contravariant type argument; for two non-equivalent types $A <: B$ and an invariant type parameter F of some type constructor T, subtyping for use-site variance has the following rules.
>
> - `T<out A>` $<:$ `T<out B>`
> - `T<in A>` $:>$ `T<in B>`
> - `T<A>` $<:$ `T<out A>`
> - `T<A>` $<:$ `T<in A>`
>
> Important: by the transitivity of the subtyping operator these rules imply that the following also holds:
>
> - `T<A>` $<:$ `T<out B>`
> - `T<in A>` $:>$ `T<B>`
>
> Note: Kotlin does not support specifying both co- and contravariance at the same time, i.e., it is impossible to have `T<out A in B>` neither on declaration- nor on use-site.
>
> Note: informally, covariant type parameter `out` A of type constructor T means "T is a producer of As and gets them out"; contravariant type parameter `in` A of type constructor T means "T is a consumer of As and takes them in".

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

Declaration-site variance

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

By default, all type parameters are invariant.

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

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

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

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

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

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

    // Invariant type parameters do not create subtyping
}

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

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

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

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

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

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

Use-site variance

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

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

By default, all type arguments are invariant.

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

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

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

In case one cannot specify any well-formed type argument, but still needs to use a parameterized type in a type-safe way, they may use *bivariant* type argument $\star$, which is roughly equivalent to a combination of `out kotlin.Any?` and `in kotlin.Nothing` (for further details, see subtyping).

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

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

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

    when (random) {
```

```
            1 -> {
                inInt = invInt    // OK
                // T<in Int> :> T<Int>

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

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

2.1.4 Type capturing

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

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

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

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

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

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

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

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

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

which can be represented as

$$L <: K <: U$$

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

> Note: for implementation reasons the compiler may approximate L and/or U; for example, in the current implementation L is always approximated to be a single type.
>
> Note: as every captured type corresponds to a fresh type variable, two different captured types K_i and K_j which describe the same set of possible types (i.e., their constraint sets are equal) are *not* considered equal. However, in some cases type inference may approximate a

captured type K to a concrete type $K^{\approx}$; in our case, it would be that $K_i^{\approx} \equiv K_j^{\approx}$.

Examples: also show the use of type containment to establish subtyping.

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

interface Root<T>

interface A
interface B : A
interface C : B

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

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

fun test01() {

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

    // Bounded<in B> <: Bounded<KB> where B <: KB <: A
    //   (from type capturing)
    // Bounded<KB> <: Root<KB>
    //   (from supertype relation)

    val test: Root<in C> = bounded

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

```
}

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

fun test02() {

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

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

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

    // ?- Foo<out B> <: Root<out Out<B>>
    //
    // Root<Out<KB>> <: Root<out Out<B>> where KB <: B
    //   (from above facts)
    // Out<KB> ⪯ out Out<B>
    //   (from subtyping for parameterized types)
    // Out<KB> <: Out<B>
    //   (from type containment rules)
    // Out<out KB> <: Out<out B>
    //   (from declaration-site variance)
    // out KB ⪯ out B
    //   (from subtyping for parameterized types)
    // out KB ⪯ out KB' where B <: KB' <: B
    //   (from type containment rules)
    // KB <: KB'
    //   (from type containment rules)
    // (KB :< B) <: (B <: KB' <: B)
    //   (from subtyping for captured types)
    // B <: B
    //   (from subtyping definition)
    // True

}

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

fun test03() {

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

    // Bar<out B> <: Bar<KB> where KB <: B
```

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

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

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

}

interface Recursive<T : Recursive<T>>

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

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

    // Recursive<*> <: Recursive<KS> where KS <: Recursive<KS>
    //   (from type capturing)
    // Recursive<KS> <: Root<KS>
    //   (from supertype relation)

    val root: Root<*> = rec

    // ?- Recursive<*> <: Root<*>
    //
    // Root<KS> <: Root<KT>
    //     where Nothing <: KS <: Recursive<KS>
    //           Nothing <: KT <: Any?
    //   (from above facts and type capturing)
```

```
    // KS ⪯ KT
    //   (from subtyping for parameterized types)
    // KS ⊆ KT
    //   (from type containment rules)
    // (Nothing <: KS <: Recursive<KS>) ⊆ (Nothing <: KT <: Any?)
    //
    // True

    val rootRec: Root<Recursive<*>> = rec

    // ?- Recursive<*> <: Root<Recursive<*>>
    //
    // Root<KS> <: Root<Recursive<*>>
    //     where Nothing <: KS <: Recursive<KS>
    //   (from above facts)
    // KS ⪯ Recursive<*>
    //   (from subtyping for parameterized types)
    // KS ⪯ KT where Recursive<*> <: KT <: Recursive<*>
    //   (from type containment rules)
    // KS ⊆ KT
    //   (from type containment rules)
    // (Nothing <: KS <: Recursive<KS) ⊆ (Recursive<*> <: KT <: Recursive<*>)
    //
    // False

}
```

2.1.5 Type containment

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

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

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

$\preceq$ is defined as follows.

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

- `in` $A \preceq$ `in` B if $A :> B$

Rules for captured types follow the same structure.

- $K_A \preceq K_B$ if $K_A \subseteq K_B$
- $K_A \preceq$ `out` K_B if $K_A <: K_B$
- $K_A \preceq$ `in` K_B if $K_A :> K_B$
- `out` $K_A \preceq$ `out` K_B if $K_A <: K_B$
- `in` $K_A \preceq$ `in` K_B if $K_A :> K_B$

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

2.1.6 Function types

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

A function type FT

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

consists of

- argument types A_i
- return type R

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

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

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

A function type with receiver FTR

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

consists of

- receiver type RT
- argument types A_i
- return type R

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

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

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

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

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

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

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

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

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

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

Suspending function types

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

For the purposes of type system, a suspending function has a *suspending* function type $\texttt{suspend}\,\mathrm{FT}(A_1, \ldots, A_n) \to R$, which is **unrelated by subtyping** to any

non-suspending function type. This is important for overload resolution and type inference, as it directly influences the types of function values and the applicability of different functions w.r.t. overloading.

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

Example:

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

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

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

2.1.7 Flexible types

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

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

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

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

As the name suggests, flexible types are flexible — a value of type $(L..U)$ can be used in any context, where one of the possible types between L and U is needed (for more details, see subtyping rules for flexible types). However, the actual

runtime type T will be a specific type satisfying $\exists S : T <: S \wedge L <: S <: U$, thus making the substitution possibly unsafe, which is why Kotlin generates dynamic assertions, when it is impossible to prove statically the safety of flexible type use.

Dynamic type

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

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

Platform types

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

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

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

2.1.8 Nullable types

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

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

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

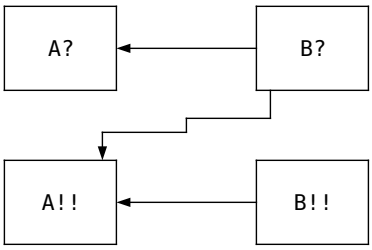
To represent a well-formed nullable type, $T?$ should satisfy the following conditions.

- T is a well-formed concrete type

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

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

Nullability lozenge



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

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

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

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

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

Example: if we have `kotlin.Int`? <: T?, we also have `kotlin.Int`!! <: T? and `kotlin.Int`!! <: T!!, meaning we can establish `kotlin.Int`!! <: $T \equiv$ `kotlin.Int` <: T.

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

Definitely non-nullable types

As discussed here, type variable types have unknown nullability, e.g., a type parameter T may correspond to either nullable version T?, or non-nullable version T!!. In some cases, one might need to specifically denote a nullable/non-nullable version of T.

Note: for example, it is needed when overriding a Java method with a `@NotNull` annotated generic parameter.

Example:

```
public interface JBox {
    <T> void put(@NotNull T t);
}

class KBox : JBox {
    override fun <T> put(t: T/* !! */) = TODO()
}
```

To denote a nullable version of T, one can use the nullable type syntax T?.

To denote a non-nullable version of T, one can use the definitely non-nullable type syntax T & Any.

To represent a well-formed definitely non-nullable type, T & Any should satisfy the following conditions.

- T is a well-formed type parameter with a nullable upper bound
- Any is resolved to `kotlin.Any`

Example:

```
typealias MyAny = kotlin.Any

fun <T /* : Any? */, Q : Any> bar(t: T?, q: Q?, i: Int?) {
    // OK
    val a: T & Any = t!!
    // OK: MyAny is resolved to kotlin.Any
    val b: T & MyAny = t!!
    // ERROR: Int is not kotlin.Any
    val c: T & Int = t!!
```

```
        // ERROR: Q does not have a nullable upper bound
        val d: Q & Any = q!!

        // ERROR: Int? is not a type parameter
        val e: Int? & Any = i!!
    }
```

One may notice the syntax looks like an intersection type T & *Any*, and that is not a coincidence, as an intersection type with *Any* describes exactly a type which cannot hold `null` values. For the purposes of the type system, a definitely non-nullable type T & *Any* is consider to be the same as an intersection type T & *Any*.

2.1.9 Intersection types

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

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

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

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

One of the main uses of intersection types are smart casts. Another restricted version of intersection types are definitely non-nullable types.

2.1.10 Integer literal types

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

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

2.1.11 Union types

> Important: Kotlin does **not** have union types in its type system. However, they make reasoning about several type system features

easier. Therefore, we decided to include a brief intro to the union types here.

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

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

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

2.2 Type contexts and scopes

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

2.2.1 Inner and nested type contexts

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

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

Example:

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

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

    // Cannot use type parameter T as a type
    // in a nested class
    class Error(val t: T)
}

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

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

2.3 Subtyping

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

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

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

2.3.1 Subtyping rules

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

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

Subtyping for captured types uses the following rules.

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

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

2.3.2 Subtyping for flexible types

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

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

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

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

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

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

However, $A \not\equiv B$.

2.3.3 Subtyping for intersection types

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

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

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

2.3.4 Subtyping for integer literal types

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

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

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

  Example:

```
interface In<in T>

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

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

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

```
    fun foo(a: Short) {}

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

fun iltAsUnion() {
    val a: Short = 42

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

2.3.5 Subtyping for nullable types

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

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

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

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

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

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

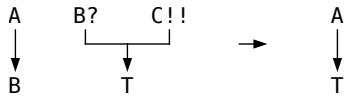
   Note: these rules follow the structure of the nullability lozenge and check the absence of nullability violation $A? \overset{\text{null}}{<:} B!!$ via underapproximating it using the *supertype* relation (as we cannot enumerate the *subtype* relation for B).

Example:

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

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

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

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

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

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

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

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

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

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

Example:



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

2.4 Upper and lower bounds

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

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

2.4.1 Least upper bound

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.2 Greatest lower bound

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5 Type approximation

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

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

Type approximation function α is defined as follows.

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

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

2.6 Type decaying

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

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

Type decaying function δ is defined as follows.

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

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

References

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