Type system

TODO(Add examples)

TODO(Add grammar snippets?)

Glossary

T
: Type

$T!!$
: Non-nullable type

$T?$
: Nullable type

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

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

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

Γ
: Type context

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

$A <:> B$
: 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 argument of a type constructor

Type argument
: Actual type argument in a parameterized type

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

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

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

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

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

$A \vert B$	Union type of A and B
`GLB`	Greatest lower bound
`LUB`	Least upper bound

TODO(Not everything is in the glossary, make some criteria of what goes where)

TODO(Cleanup glossary)

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][Exceptions].

Kotlin has a type system with the following main properties.

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

TODO(static type checking, gradual type checking)

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][Cast expression]. 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][Smart casts].

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

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
- TODO(Error / invalid 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 internally by the compiler.

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.

TODO(`kotlin.Any` members?)

`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.

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

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

Additional details about how `kotlin.Nothing` should be processed are available [here][Control- and data-flow analysis].

`kotlin.Unit`

`kotlin.Unit` is a unit type, i.e., a type with only one value `kotlin.Unit`; all values of type `kotlin.Unit` should reference the same underlying `kotlin.Unit` object.

TODO(Compare to `void`?)

`kotlin.Function`

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

Classifier types

Classifier types represent regular types which are declared as [classes][Classes], [interfaces][Interfaces] or [objects][Objects]. As Kotlin supports generics, 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

  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`

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
- $\forall i \in [1, n] : K_T(F_i, A_i)$ is a well-formed captured type, where K is a type capturing operator

  Example:

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

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

  // A well-formed PACT with a single iPACT 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 PACT,
//   as an abstract type Generic
//   cannot be used as a supertype
interface Invalid<P> : Generic


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


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

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

// An ill-formed type,
//   as NumberWrapper<String> is an ill-formed iPACT
//   (String <:> Number)
interface InvalidWrapper : NumberWrapper<String>
```

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 $F <:$ `kotlin.Any`?.

A bounded type parameter additionally specify 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 PACT 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; as any type may be a subtype of no more than one class type, it makes no sense to support several class type bounds. 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).

- Bounded type parameter with type parameter bound:
  - F is a type parameter of PACT 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 $K <: U$ where $U = B_1 \,\&\, \ldots \,\&\, B_n$.

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 types $A <: B$, subtyping between `T<A>` and `T<B>` depends on the variance of type parameter F of 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> <:> 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 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>`
- `T<in A> <:> T<out A>`

> Note: Kotlin does not support specifying both co- and contravariance at the same time, i.e., it is impossible to have `T<in A out B>` neither on declaration- nor on use-site.

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

TODO(Fix formatting here)

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., type parameters of functions 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.

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.

> Note: in some cases, Kotlin prohibits certain combinations of declaration- and use-site variance, i.e., which type arguments can be used in which type parameters. These rules are covered in more detail [here][TODO()].

In case one cannot specify any well-formed type argument, but still needs to use a parameterized type in a type-safe way, one 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 and generics).

TODO(Specify how this combination of co- and contravariant parameters works from the practical PoV)

Important: use-site variance cannot be used when declaring a supertype.

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> <:> T<out Int>
        }
    }
```

```
}
```

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: $\exists X : X <: T.\mathtt{A} < \mathtt{X} >$. 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 $T[\sigma]$ uses the following rules to create captured type K_i from the type parameter F_i and type argument A_i.

> 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 $F_i <: B_1, \ldots, B_m$, if $\exists j \in [1, m] : \neg(A_i <: B_j)$, K_i is an ill-formed type. Otherwise, $\forall j \in [1, m] : K_i <: \sigma B_j$.
- 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 $\star$, $\mathtt{kotlin.Nothing} <: K_i <: \mathtt{kotlin.Any?}$.
- 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 $L = L_1 | \ldots | L_p$ and $U = U_1 \,\&\, \ldots \,\&\, U_q$.

> 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][Type inference] may approximate a captured type K to a concrete type $K^{\approx}$; in our case, it would be that $K_i^{\approx} \equiv K_j^{\approx}$.

TODO(Need to think more about this part)

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

$$FT(A_1, \ldots, A_n) \rightarrow R$$

consists of

- argument types A_i
- return type R

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

$$FT(A_1, \ldots, A_n) \rightarrow R \equiv FunctionN[A_1, \ldots, A_n, R]$$

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

A function type with receiver FTR

$$FTR(TH, A_1, \ldots, A_n) \rightarrow R$$

consists of

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

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

$$FTR(TH, A_1, \ldots, A_n) \rightarrow R \equiv FT(TH, A_1, \ldots, A_n) \rightarrow R$$

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

> Note: this means that, for example, these two types are equivalent
>
> - `Int.(Int) -> String`
> - `(Int, Int) -> String`

Furthermore, all function types $FunctionN$ are subtypes of a general argument-agnostic type [`kotlin.Function`][kotlin.Function] for the purpose of unification.

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

TODO(We already have `kotlin.Function` settled in this spec earlier. The reason for this is that overloading needs it)

> 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
> ```

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

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

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

Array type specialization `ATS`(T) 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][Variable length parameters] are handled.

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
- $L <: U$
- $\neg(L <: U)$
- 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 type will be a specific type between L and 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.

TODO(Details of assertion generation?)

Dynamic type

Kotlin includes a special *dynamic* type, which is 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.

TODO(We should reconsider defining `dynamic` as a flexible type, cause it doesn't behave like one in many situations)

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:

- [Platform types for Java][TODO(need a way to have same section names in different parts of the spec)]

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

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

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][Navigation operators]
2. Downcast from $T?$ to $T!!$
   - [unsafe cast][Cast expression]
   - [type check][Type-checking expression] combined with [smart casts][Smart casts]
   - null check combined with [smart casts][Smart casts]
   - [not-null assertion operator][Not-null assertion expression]
3. Supply a default value to use if `null` is present
   - [elvis operator][Elvis operator expression]

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 $\texttt{GLB}(A, B)$. 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][Smart casts].

Integer literal types

TODO(Think this through)

An integer literal type containing types $T_1, \ldots, T_N$, denoted $\texttt{LTS}(T_1, \ldots, T_N)$ 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][Built-in integer types]

Integer literal types are the types of [integer literals][Integer literals].

TODO(Consult with the team)

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 $A|B$ and is equivalent to the least upper bound of its components $\texttt{LUB}(A, B)$. 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 approximation.

Type context

TODO(Type contexts and their relation to scopes) TODO(Inner vs nested type contexts)

Subtyping

TODO(Need to change the way we think about 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 ($A <: A$)
- transitive ($A <: B \land B <: C \Rightarrow A <: C$)

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 **not** transitive (see here for more details).

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 $\widehat{T} = T[\sigma] : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : \widehat{T} <: \sigma S_i$
- For any two parameterized types $\widehat{T}$ and $\widehat{T'}$ with captured type arguments K_i and K_i' it is true that $\widehat{T} <: \widehat{T'}$ if $\forall i \in [1, n] : K_i <: K_i'$

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

- $\forall T : \texttt{kotlin.Nothing} <: T <: \texttt{kotlin.Any}$
- For any type constructor $\widehat{T} = T(F_1, \ldots, F_n) : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : \widehat{T} <: S_i$
- For any two type constructors $\widehat{T}$ and $\widehat{T'}$ with type parameters F_i and F_i' it is true that $\widehat{T} <: \widehat{T'}$ if $\forall i \in [1, n] : F_i <: F_i'$

Subtyping for type parameters uses the following rules.

- $\forall F : \texttt{kotlin.Nothing} <: F <: \texttt{kotlin.Any}?$
- For any two type parameters F and F', it is true that $F <: F'$, if all of the following hold
  - variance of F matches variance of F'
    - `out` matches `out`
    - `in` matches `in`
    - `inv` matches any variance
  - for $F <: B$ and $F' <: B'$, $B <: B'$

Subtyping for captured types uses the following rules.

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

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

Subtyping for flexible types

Flexible types (being flexible) follow a simple subtyping relation with other inflexible types. Let T, A, B, L, U be inflexible 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:

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

However, $A \not\equiv B$.

Subtyping for intersection types

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

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

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

Subtyping for integer literal types

Every integer literal type is equivalent with w.r.t. subtyping, meaning that for any sets $T_1, \ldots, T_K$ and $U_1, \ldots, U_N$ of builtin integer types:

- $\texttt{LTS}(T_1, \ldots, T_K) <: \texttt{LTS}(U_1, \ldots, U_N)$

- $\texttt{LTS}(U_1, \ldots, U_K) <: \texttt{LTS}(T_1, \ldots, T_K)$
- $\forall T_i \in \{T_1, \ldots, T_K\}. T_i <: \texttt{LTS}(T_1, \ldots, T_K)$
- $\forall T_i \in \{T_1, \ldots, T_K\}. \texttt{LTS}(T_1, \ldots, T_K) <: T_i$

Subtyping for nullable types

TODO(Why can't we just say that $\forall T : T <: T?$ and $\forall T : T!! <: T$ and be done with it?)

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

- Regular subtyping $<:$ for non-nullable types $A!!$ and $B!!$
- Subtyping by nullability $\overset{null}{<:}$

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

- $A!! \overset{null}{<:} B$
- $A \overset{null}{<:} B$ if $\exists T!! : A <: T!!$
- $A \overset{null}{<:} B?$
- $A \overset{null}{<:} B$ if $\nexists T!! : B <: T!!$

TODO(How the existence check works)

Generics

TODO(How are generics different from type parameters? Or are we going to get into deep technical detail?)

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.

Least upper bound

The *least upper bound* $\texttt{LUB}(A, B)$ 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., $\texttt{LUB}(A, B) = \texttt{LUB}(B, A)$. 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.

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

- $\texttt{LUB}(A, A) = A$
- if $A <: B$, $\texttt{LUB}(A, B) = B$
- if A is nullable, $\texttt{LUB}(A, B)$ is also nullable
- if both A and B are nullable, $\texttt{LUB}(A, B) = \texttt{LUB}(A!!, B!!)?$
- if A is nullable and B is not, $\texttt{LUB}(A, B) = \texttt{LUB}(A!!, B)?$
- if $A = T\langle K_{A,1}, \ldots, K_{A,n}\rangle$ and $B = T\langle K_{B,1}, \ldots, K_{B,n}\rangle$, $\texttt{LUB}(A, B) = T\langle \phi(K_{A,1}, K_{B,1}), \ldots, \phi(K_{A,n}, K_{B,n})\rangle$, where $\phi(X, Y)$ is defined as follows:
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ X) = X$
  - $\phi(\texttt{out}\ X, \texttt{out}\ Y) = \texttt{out}\ \texttt{LUB}(X, Y)$
  - $\phi(\texttt{out}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{out}\ X, \texttt{in}\ Y) = \star$
  - $\phi(\texttt{inv}\ X, \texttt{out}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{inv}\ X, \texttt{in}\ Y) = \phi(\texttt{out}\ X, \texttt{out kotlin.Any}?) = \texttt{out kotlin.Any}?$
  - $\phi(\texttt{in}\ X, \texttt{out}\ Y) = \star$
  - $\phi(\texttt{in}\ X, \texttt{inv}\ Y) = \phi(\texttt{out kotlin.Any}?, \texttt{out}\ Y) = \texttt{out kotlin.Any}?$
  - $\phi(\texttt{in}\ X, \texttt{in}\ Y) = \texttt{in}\ \texttt{GLB}(X, Y)$
  - TODO(we may also choose the `in` projection for `inv` parameters, do we wanna do it though?)
- if $A = (L_A..U_A)$ and $B = (L_B..U_B)$, $\texttt{LUB}(A, B) = (\texttt{LUB}(L_A, L_B)..\texttt{LUB}(U_A, U_B))$
- if $A = (L_A..U_A)$ and B is not flexible, $\texttt{LUB}(A, B) = (\texttt{LUB}(L_A, B)..\texttt{LUB}(U_A, B))$

TODO(prettify formatting)

TODO(actual algorithm for computing LUB)

TODO(LUB for 3+ types)

TODO(what do we do if this procedure loops?)

TODO(Why do we need union types again?)

Greatest lower bound

The *greatest lower bound* $\texttt{GLB}(A, B)$ 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: enumerating all subtypes of a given type is impossible in general, but in the presence of intersection types, $GLB(A, B) \equiv A \,\&\, B$.

TODO(It's not if types are related)

> Note: GLB is commutative, i.e., $\texttt{GLB}(A, B) = \texttt{GLB}(B, A)$. 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.

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

- $\texttt{GLB}(A, A) = A$
- if $A <: B$, $\texttt{GLB}(A, B) = A$
- if A is non-nullable, $\texttt{GLB}(A, B)$ is also non-nullable
- if both A and B are nullable, $\texttt{GLB}(A, B) = \texttt{GLB}(A!!, B!!)$?
- if A is nullable and B is not, $\texttt{GLB}(A, B) = \texttt{GLB}(A!!, B)$
- if $A = T\langle K_{A,1}, \ldots, K_{A,n}\rangle$ and $B = T\langle K_{B,1}, \ldots, K_{B,n}\rangle$, $\texttt{GLB}(A, B) = T\langle\phi(K_{A,1}, K_{B,1}), \ldots, \phi(K_{A,n}, K_{B,n})\rangle$, where $\phi(X, Y)$ is defined as follows:
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ X) = X$
  - $\phi(\texttt{out}\ X, \texttt{out}\ Y) = \texttt{out}\ \texttt{GLB}(X, Y)$
  - $\phi(\texttt{out}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$

- $\phi(\texttt{out}\ X, \texttt{in}\ Y) = \star$
- $\phi(\texttt{inv}\ X, \texttt{out}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
- $\phi(\texttt{inv}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
- $\phi(\texttt{inv}\ X, \texttt{in}\ Y) = \phi(\texttt{out}\ X, \texttt{out kotlin.Any}?) = \texttt{out kotlin.Any}?$
- $\phi(\texttt{in}\ X, \texttt{out}\ Y) = \star$
- $\phi(\texttt{in}\ X, \texttt{inv}\ Y) = \phi(\texttt{out kotlin.Any}?, \texttt{out}\ Y) = \texttt{out kotlin.Any}?$
- $\phi(\texttt{in}\ X, \texttt{in}\ Y) = \texttt{in LUB}(X, Y)$
- TODO(we may also choose the `in` projection for `inv` parameters, do we wanna do it though?)

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

TODO(prettify formatting)

TODO(actual algorithm for computing GLB)

TODO(GLB for 3+ types)

TODO(what do we do if this procedure loops?)

Type approximation

TODO()

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.

TODO(the big TODO for the whole chapter: we need to clearly decide what kind of type system we want to specify: an algo-driven ts vs a full declarational ts, operation-based or relation-based. An example of the second distinction would be difference between $(A?)!!$ and $((A!!)?)!!$. Are they the same type? Are they different, but equivalent? Same goes for $(A..B)?$ vs $(A?..B?)$ and such.)

TODO(another big question is: do we want to formally prove all the different thing here?)