Generics

generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

To reference the generic Box class from within your code, you must perform a generic type invocation, which replaces T with some concrete value, such as Integer:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — Integer in this case — to the Box class itself.

A raw type is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of Box<T>:

```
Box rawBox = new Box();
```

Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is not a raw type.

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

Wildcard

In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound.

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the Object class.
- When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Consider the following method, printList:

```
public static void printList(List<Object> list) {

    for (Object elem : list)

        System.out.println(elem + " ");

    System.out.println();

}
```

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>. To write a generic printList method, use List<?>

Diferencia entre wildcard y type parameter

When instantiating, we need to provide a concrete type (type argument). We can't use wildcards to define a generic class or interface.

To write a method with a generic type argument, we should use a type parameter. So, let’s create a method that prints a given item:

```
public static <T> void print(T item){

    System.out.println(item);

}
```

In this example, we aren't able to replace the type argument T with the wildcard. We can't use wildcards directly to specify the type of a parameter in a method.

If a type parameter appears only once in the method declaration, we should consider replacing it with a wildcard.

When a generic method returns a generic type, we should use a type parameter instead of a wildcard.

We can't use type parameters with the lower bound. Furthermore, type parameters can have multiple bounds, while wildcards can’t.

When dealing with collections, a common rule for selecting between upper or lower bounded wildcards is PECS. **PECS stands for producer extends, consumer super. Producers to objects you only read from and Consumers to those you only write to.**

You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

An "In" Variable

- An "in" variable serves up data to the code. Imagine a copy method with two arguments: copy(src, dest). The src argument provides the data to be copied, so it is the "in" parameter.

An "Out" Variable

- An "out" variable holds data for use elsewhere. In the copy example, copy(src, dest), the dest argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

Wildcard Guidelines:

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

Generics Kotlin

1. **A subtype must accept at least the same range of types as its supertype declares.**
2. **A subtype must return at most the same range of types as its supertype declares.**

out produces T and preserves subtyping

When you declare a generic type with an out modifier, it's called covariant. A covariant is a producer of T, that means functions can return T but they can't take T as arguments

List<out T> in Kotlin is equivalent to List<? extends T> in Java.

in consumes T and reverses subtyping

When you declare a generic type with an in modifier, it's called contravariant. A contravariant is a consumer of T, that means functions can take T as arguments but they can't return T

List<in T> in Kotlin is equivalent to List<? super T> in Java

UNIT, NOTHING, ANY

Unit эквивалентен void в Java. В этом выражении возвращаемый тип можно не указывать, если функция ничего не возвращает.

Nothing — класс, который является наследником любого класса в Kotlin, даже класса с модификатором final. При этом Nothing нельзя создать — у него приватный конструктор. (Es el usado por TODO).

Класс Any находится на вершине иерархии — все классы в Kotlin являются наследниками Any. Any — это аналог Object в Java, но с меньшим количеством методов.

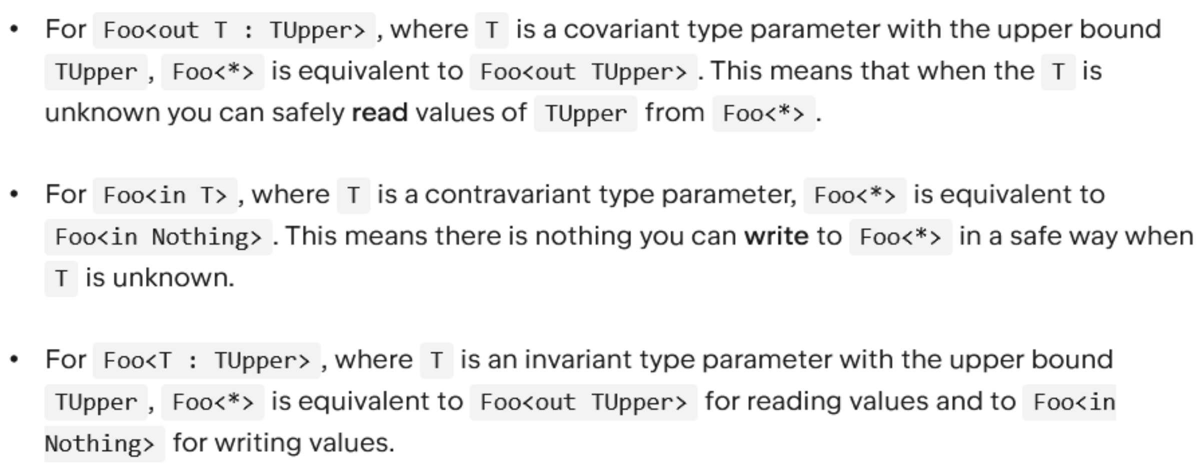
Star-projections

There are times when you want a function that can accept *any* kind of a particular generic.

1. Every kind of Group must accept at least the same set of types as its SuperGroup declares.
2. Every kind of Group must return at most the same set of types as its SuperGroup declares.

We’ve concluded that, for every function that accepts or returns the type parameter, our SuperGroup will need to:

- Accept Nothing
- Return Any?



- For `Foo<out T : TUpper>`, where `T` is a covariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>`. This means that when the `T` is unknown you can safely **read** values of `TUpper` from `Foo<*>`.

- For `Foo<in T>`, where `T` is a contravariant type parameter, `Foo<*>` is equivalent to `Foo<in Nothing>`. This means there is nothing you can **write** to `Foo<*>` in a safe way when `T` is unknown.

- For `Foo<T : TUpper>`, where `T` is an invariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>` for reading values and to `Foo<in Nothing>` for writing values.

Inline classes

Sometimes it is necessary for business logic to create a wrapper around some type. However, it introduces runtime overhead due to additional heap allocations. Moreover, if the wrapped type is primitive, the performance hit is terrible, because primitive types are usually heavily optimized by the runtime, while their wrappers don't get any special treatment.

To solve such issues, Kotlin introduces a special kind of class called an inline class. Inline classes are a subset of value-based classes. They don't have an identity and can only hold values.

To declare an inline class, use the `value` modifier before the name of the class:

```
value class Password(private val s: String)
```

To declare an inline class for the JVM backend, use the `value` modifier along with the `@JvmInline` annotation before the class declaration:

```
// For JVM backends
@JvmInline
value class Password(private val s: String)
```

⚠ The `inline` modifier for inline classes is deprecated.

An inline class must have a single property initialized in the primary constructor. At runtime, instances of the inline class will be represented using this single property (see details about runtime representation below):

```
// No actual instantiation of class 'Password' happens
// At runtime 'securePassword' contains just 'String'
val securePassword = Password("Don't try this in production")
```

Inline classes support some functionality of regular classes. In particular, they are allowed to declare properties and functions, and have the init block

Inline class properties cannot have backing fields. They can only have simple computable properties (no lateinit/delegated properties).

Inline classes are allowed to inherit from interfaces

It is forbidden for inline classes to participate in a class hierarchy. This means that inline classes cannot extend other classes and are always final.

Inline functions

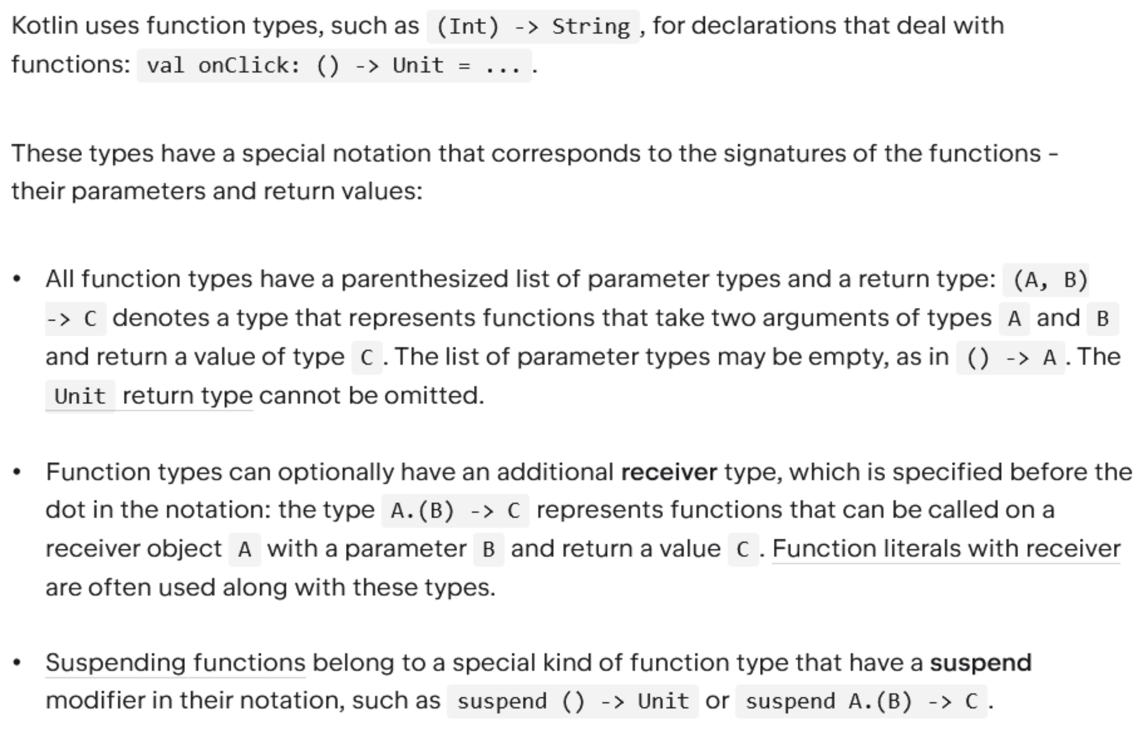
В Kotlin существует модификатор inline, которым можно пометить функцию. Основное его предназначение - повысить производительность. Чтобы понять за счёт чего она повышается, нужно вспомнить лямбда-выражения.

Как правило, лямбда-выражения компилируются в анонимные классы. То есть каждый раз, когда используется лямбда-выражение, создаётся дополнительный класс. Отсюда вытекают дополнительные накладные расходы у функций, которые принимают лямбду в качестве аргумента. Если же функцию отметить модификатором inline, то компилятор не будет создавать анонимные классы и их объекты для каждого лямбда-выражения, а просто вставит код её реализации в место вызова. Или другими словами встроит её.

Kotlin functions are first-class, which means they can be stored in variables and data structures, and can be passed as arguments to and returned from other higher-order functions. You can perform any operations on functions that are possible for other non-function values.

A higher-order function is a function that takes functions as parameters, or returns a function.

Function types



Kotlin uses function types, such as `(Int) -> String`, for declarations that deal with functions: `val onClick: () -> Unit = ...`.

These types have a special notation that corresponds to the signatures of the functions - their parameters and return values:

- All function types have a parenthesized list of parameter types and a return type: `(A, B) -> C` denotes a type that represents functions that take two arguments of types `A` and `B` and return a value of type `C`. The list of parameter types may be empty, as in `() -> A`. The `Unit` return type cannot be omitted.

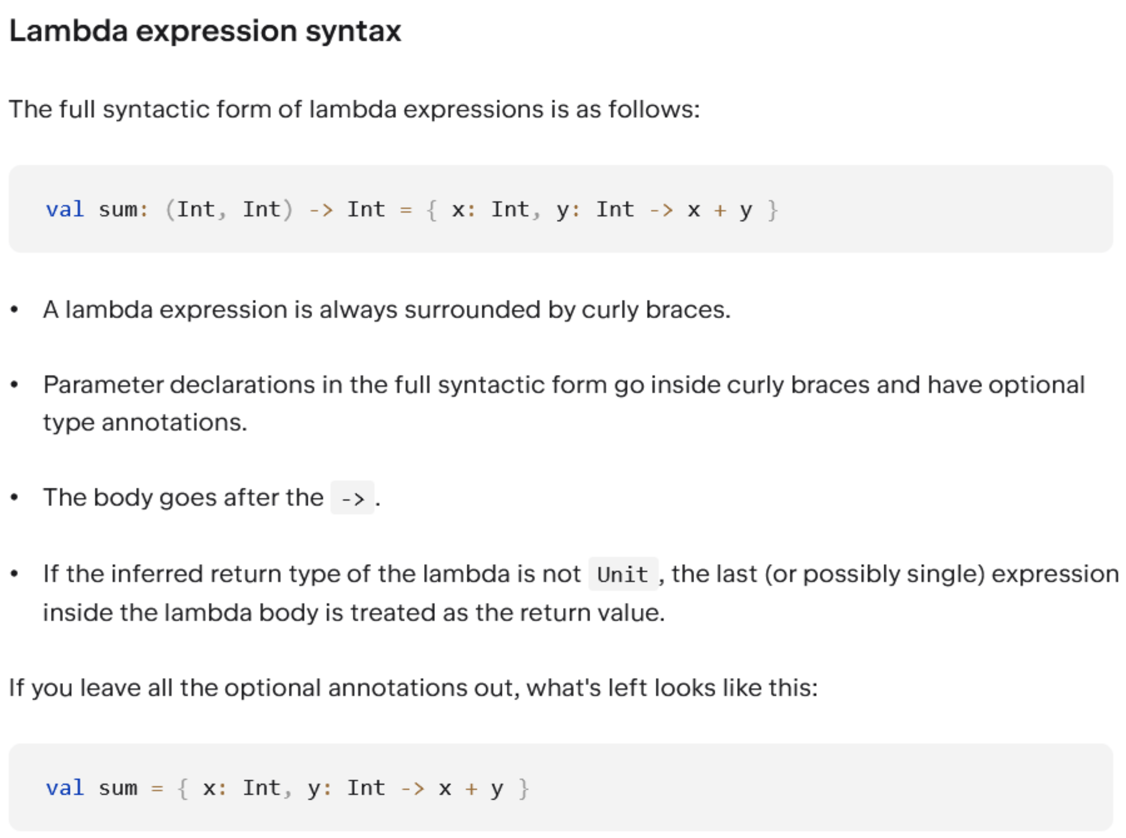
- Function types can optionally have an additional **receiver** type, which is specified before the dot in the notation: the type `A.(B) -> C` represents functions that can be called on a receiver object `A` with a parameter `B` and return a value `C`. Function literals with receiver are often used along with these types.

- Suspending functions belong to a special kind of function type that have a **suspend** modifier in their notation, such as `suspend () -> Unit` or `suspend A.(B) -> C`.

Lambda expressions

are functions that are not declared but are passed immediately as an expression

Для выхода только из лямбды используется `label`.



Lambda expression syntax

The full syntactic form of lambda expressions is as follows:

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

- A lambda expression is always surrounded by curly braces.
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
- The body goes after the `->`.
- If the inferred return type of the lambda is not `Unit`, the last (or possibly single) expression inside the lambda body is treated as the return value.

If you leave all the optional annotations out, what's left looks like this:

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

Serialization

Serialization is the process of converting data used by an application to a format that can be transferred over a network or stored in a database or a file. In turn, deserialization is the opposite process of reading data from an external source and converting it into a runtime object.

kotlin core

infix fun

Functions marked with the infix keyword can also be called using the infix notation (omitting the dot and the parentheses for the call). Infix functions must meet the following requirements:

- They must be member functions or extension functions.
- They must have a single parameter.
- The parameter must not accept variable number of arguments and must have no default value.

```
infix fun Int.shl(x: Int): Int { ... }

// calling the function using the infix notation
1 shl 2

// is the same as
1.shl(2)
```

Infix function calls have lower precedence than arithmetic operators, type casts, and the `rangeTo` operator. The following expressions are equivalent:

- `1 shl 2 + 3` is equivalent to `1 shl (2 + 3)`
- `0 until n * 2` is equivalent to `0 until (n * 2)`
- `xs union ys as Set<*>` is equivalent to `xs union (ys as Set<*>)`

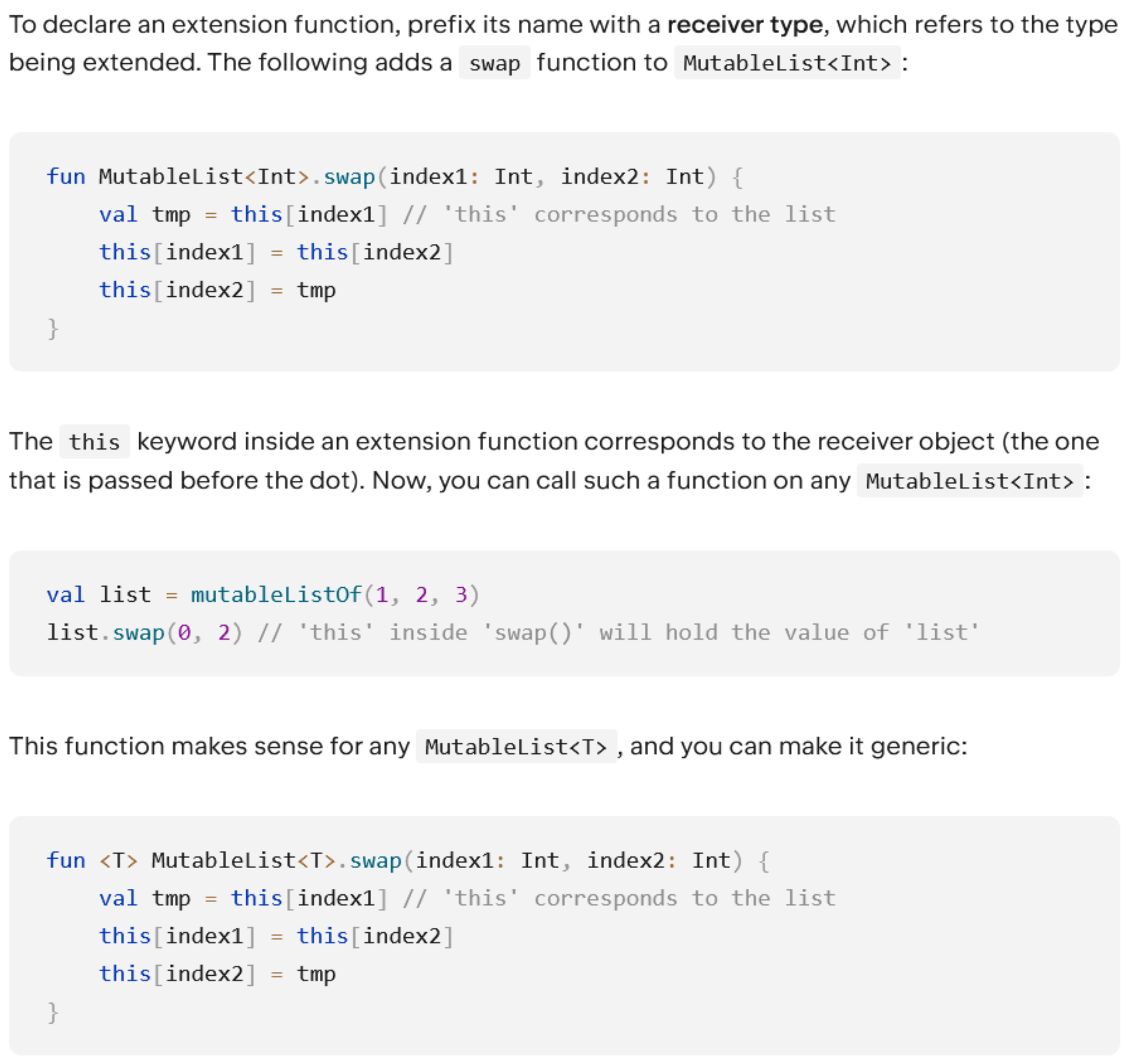
On the other hand, an infix function call's precedence is higher than that of the boolean operators `&&` and `||`, `is`- and `in`-checks, and some other operators. These expressions are equivalent as well:

- `a && b xor c` is equivalent to `a && (b xor c)`
- `a xor b in c` is equivalent to `(a xor b) in c`

extension fun

Kotlin provides the ability to extend a class or an interface with new functionality without having to inherit from the class or use design patterns

For example, you can write new functions for a class or an interface from a third-party library that you can't modify. Such functions can be called in the usual way, as if they were methods of the original class. This mechanism is called an extension function. There are also extension properties that let you define new properties for existing classes.



To declare an extension function, prefix its name with a **receiver type**, which refers to the type being extended. The following adds a `swap` function to `MutableList<Int>`:

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

The `this` keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot). Now, you can call such a function on any `MutableList<Int>`:

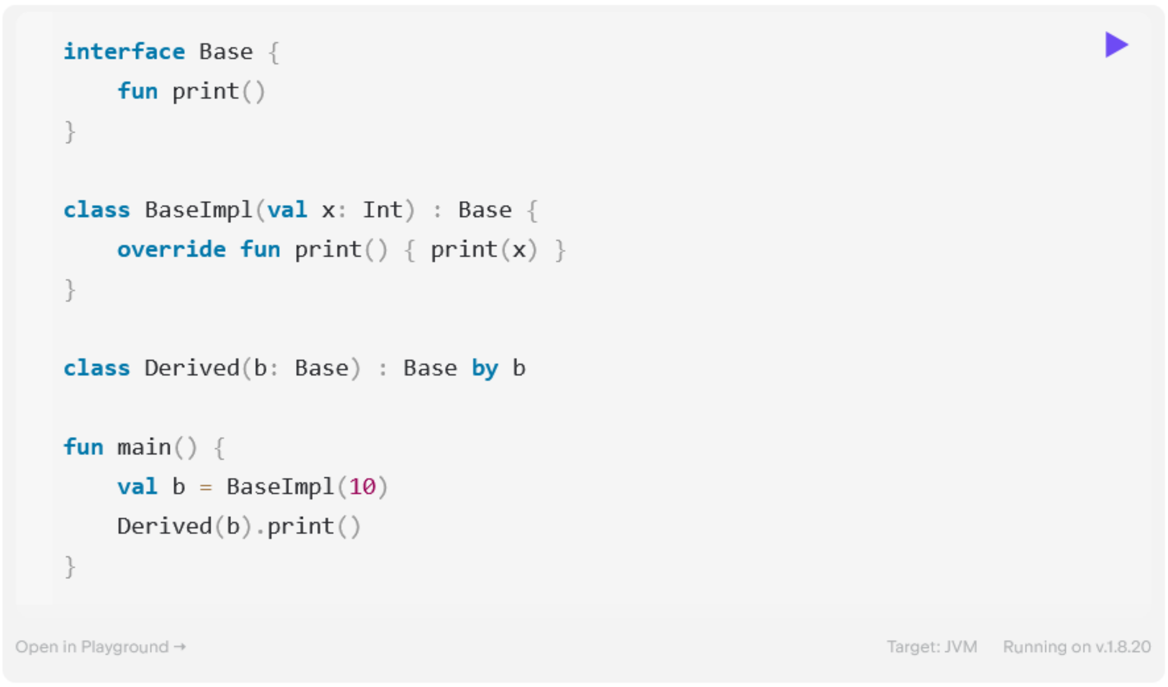
```kotlin
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list'
```

This function makes sense for any `MutableList<T>`, and you can make it generic:

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

delegates

A class Derived can implement an interface Base by delegating all of its public members to a specified object:



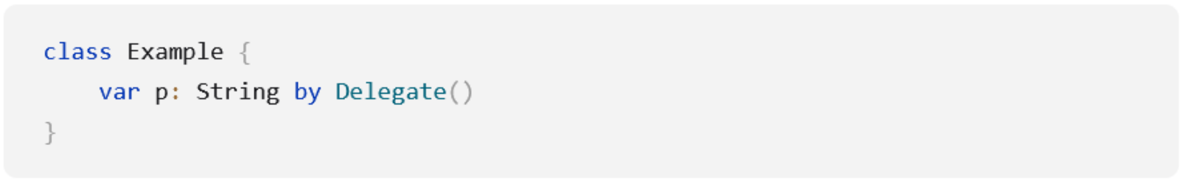
```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Open in Playground → Target: JVM Running on v.1.8.20

The `by` -clause in the supertype list for `Derived` indicates that `b` will be stored internally in objects of `Derived` and the compiler will generate all the methods of `Base` that forward to `b` .

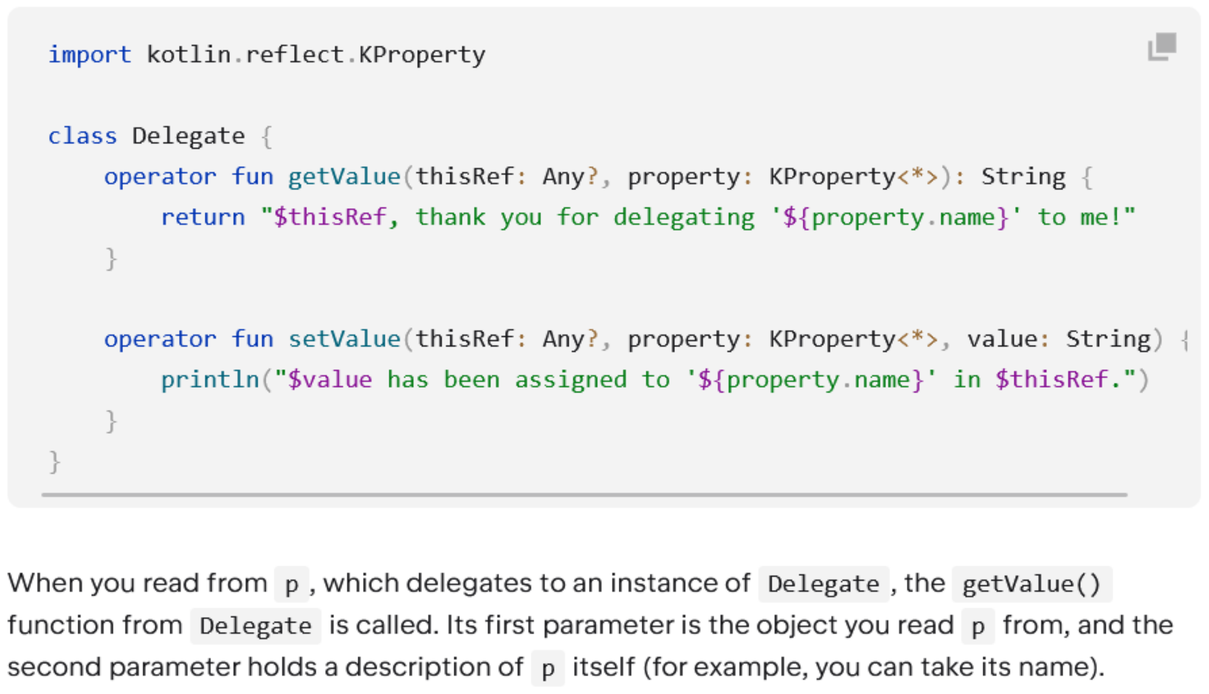
Delegated properties

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



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

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



```kotlin
import kotlin.reflect.KProperty

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

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

When you read from `p`, which delegates to an instance of `Delegate`, the `getValue()` function from `Delegate` is called. Its first parameter is the object you read `p` from, and the second parameter holds a description of `p` itself (for example, you can take its name).

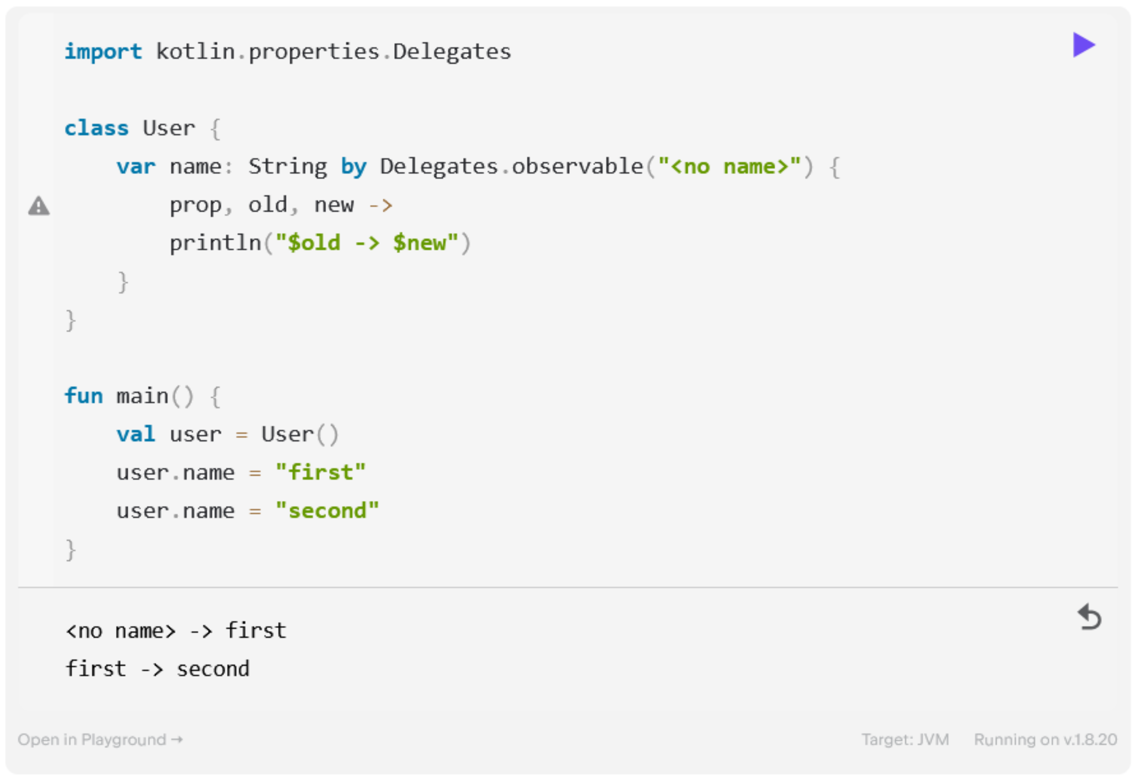
Standard Delegates

Lazy properties

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

Observable properties

The handler is called every time you assign to the property (**after** the assignment has been performed). It has three parameters: the property being assigned to, the old value, and the new value:



```kotlin
import kotlin.properties.Delegates

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

fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

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

Open in Playground → Target: JVM Running on v.1.8.20

If you want to intercept assignments and **veto** them, use `vetoable()` ↗ instead of `observable()`. The handler passed to `vetoable` will be called **before** the assignment of a new property value.

Delegating to another property

A property can delegate its getter and setter to another property. Such delegation is available for both top-level and class properties (member and extension). The delegate property can be:

- A top-level property
- A member or an extension property of the same class
- A member or an extension property of another class

To delegate a property to another property, use the :: qualifier in the delegate name, for example, this::delegate or MyClass::delegate.

Storing properties in a map

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

In this example, the constructor takes a map:

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

Delegated properties take values from this map through string keys, which are associated with the names of properties:

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

Open in Playground → Target: JVM Running on v.1.8.20