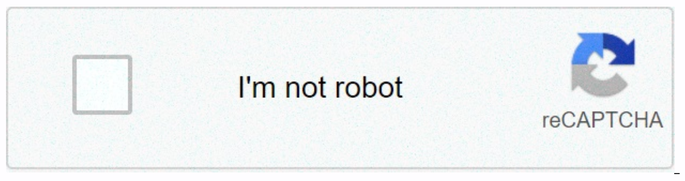

Continue

Android studio convert kotlin code to java

Google is committed to promoting racial equity for black communities. See how. [This codeplate is also available in Chinese and Brazilian Portuguese] In this codeplate, you'll learn how to convert your code from Java to Kotlin. You will also learn what Kotlin language conventions are and how to ensure that the code you write follows them. This codeplate is suitable for all developers using Java who are considering transferring the project to Kotlin. We start with a couple of Java classes that you convert to Kotlin using IDE. Then we take a look at the converted code and see how we can improve it by making it more idiomatic and avoiding common pitfalls. What you learn You will learn how to convert Java to Kotlin. By doing this you will learn the following Kotlin language features and concepts: Handling nullability Implementing singletons Data classes Handling strings Elvis operator Destructuring Properties and backing properties Standard arguments and named parameters Work with collections Extension functions Top level functions and parameters let, search, with and run keyword prerequisites You should already be familiar with Java. What you need Android Studio 4.0 or IntelliJ IDEA Note that automatic conversion in future versions of Android Studio can create slightly different results. If you are using IntelliJ IDEA, create a new Java project with Kotlin/JVM. If you're using Android Studio, create a new project with no activity. The minimum SDK can be of any value, it will not affect the outcome. Code We create a user model object and a Repository singleton class that works with user objects and displays lists of users and formatted user names. Create a new file called User.java under app /java /<yourpackagename> and paste into the following code: Public Class User { @Nullable private String first name; @Nullable private String last name; public user(string first name, string last name) { this.firstName = first name; this.last name = last name; } common string getFirstName() { return first name; } public invalid setFirstName(String first name) { this.firstName = firstname, } common string getLastName() { return last name; } public void setLastName(String last name) { this.lastName = last name} } You will notice your IDE tells you @Nullable is not defined. So import androidx.annotation.Nullable if you are using Android Studio, or org.jetbrains.annotations.Nullable if you are using IntelliJ. Create a new file called Repository.java and paste into the following code: import java.util.ArrayList; import java.util.List; public class Repository { private static repository INSTANCE = null; private<User> List users = null; public static repository getInstance() { if (INSTANCE == null) { synchronized (Repository.class) { if (INSTANCE == zero) { INSTANCE = new repository(); } } return INSTANCE; } // keep the constructor private to enforce the use of private Repository() { Bruker bruker1 = ny bruker (Jane,); Bruker2 = ny bruker(John, null); </User></yourpackagename> </User></yourpackagename> user3 = new user(Anne, Doe); users = new ArrayList(); users.add (user1); users.add (user2); users.add (user3); } public list<User> getUsers() { return users; } public<String> list getFormattedUserNames() { List<String> username = new ArrayList <> (users.size()); for (User User: Users) { String Name, if (user.getLastName() != null) { if (user.getFirstName() != null) { name = user.getFirstName() + + user.getLastName(); } other { name = user.getLastName(); } other if (user.getFirstName() != null) { name = user.getFirstName(); } else { name = Unknown, } userNames.add(name); } return username; } } Our IDE can do a pretty good job of automatically converting Java code to Kotlin code, but sometimes it needs some help. Let's let our IDE make a first pass on the conversion. Then we go through the resulting code to understand how and why it has been converted in this way. The automatic converter in future versions of Android Studio can create different results. The results below were done using Android Studio 4.0. Go to the user.java file and convert it to Kotlin: Menu Bar -> Code -> Convert Java File to Kotlin File. If your IDE requests correction after conversion, press yes. You should see the following Kotlin code: classUser(was first name: String?, was last name: string?) Please note that .java was renamed User.kt. Kotlin files have the extension .kt. Pro tip: If you paste Java code into a Kotlin file, the IDE automatically converts the pasted code to Kotlin. In our Java user class, we had two characteristics: first name and last name. Each had a getter and set method, making its value mutable. Kotlin's keywords for mutable variables are, so the converter user was for each of these properties. If our Java properties had only getters, they would be immutable and would have been declared as valvariables. val is similar to the final keyword in Java. One of the most important differences between Kotlin and Java is that Kotlin explicitly indicates whether a variable can accept a null value. It does this by adding a '?' to the type statement. Because we marked first and last names as nullable, the converter automatically marked the properties as nullable with String?. If you comment on your Java members as non-zero (using org.jetbrains.annotations.NotNull or androidx.annotation.NonNull), the converter will recognize this and make the fields non-null in Kotlin as well. Pro tip: In Kotlin, we recommend using immutable objects where possible (that is, using val instead of var) and avoiding zero-emission types. You should strive to make nullability meaningful and something you want to deal with specifically. The basic conversion has already been done. But we can write this in a more idiomatic way. Let's see how. Data class Our user class contains only data. Kotlin has a for classes with this role: data. By marking this class as a data class, the compiler will automatically create getters and setters for us. It will also deduce equivalent(),</String> </String> </User> </User> and toString() functions. Let's add the data search word to our user class: Data Class User(was first name: String, was last name: String) Kotlin, like Java, can have a primary constructor and one or more secondary constructors. The one in the example above is the primary constructor of the user class. If you convert a Java class that has multiple constructors, the converter will automatically create multiple constructors in Kotlin as well. They are defined by using the constructor keyword. Read more about constructors in the official documentation. If we want to create an instance of this class, we can do it as follows: val user1 = User(Jane, Doe) Equality Kotlin has two types of equality: Structural equality uses == operator and calls equal() to determine whether two instances are the same. Referential equality uses the === operator and checks whether two references point to the same object. The properties defined in the primary constructor of the computer class will be used for structural equality checks. val user1 = User(Jane, Doe) val user2 = User(Jane, Doe) val structurallyEqual = user1 == user2 // true val referentiallyEqual = user1 === user2 // false Read more about data classes in the official documentation. In Kotlin, we can assign default values to arguments in function calls. The default value is used when the argument is omitted. In Kotlin, constructors are also functions, so we can use default arguments to indicate that the default value for last name is zero. To do this, we only assign zero to last name. data class User(was first name: String?, was last name: String? = zero) // usage val jane = User(Jane) // same as User(Jane, zero) val joe = User(Joe, Doe) Kotlin allows you to mark your arguments when your functions are called: val john = User (first name = John, last name = Doe) As another usage style, let's say that your first name has zero as the default value and last name does not. In this case, because the default parameter will precede a parameter without default value, you must call the function with named arguments: data class User(was first name: String? = null, was last name: String?) // use val jane = User(last name = Doe) // same as User(zero, Doe) val john = User(John, Doe) Default values are an important and commonly used concept in the Kotlin code. In our codeplate, we always want to specify the first and last names in a user object declaration so that we don't need default values. If the feature has multiple parameters, consider using named arguments when making the code more readable. Before proceeding with the codeplate, make sure that the user class is a data class. Let's convert the repository to Kotlin. The automatic conversion result should look like this: import java.util.* class Repository private constructor() { private were users: MutableList<User?>? = zero fun getUsers (): List<User?>? { return users } val formattedUserNames: List<String?> get() { val username: =</String?> </String?> </User?> </User?> </User?> for (bruker i brukere) { var navn: Strengnavn = hvis (bruker!!. etternavn != null) { hvis (bruker!!. fornavn != null) { bruker!!. fornavn + + bruker!!. etternavn } ellers { bruker!!. etternavn } } ellers hvis (bruker!!. fornavn != null) { bruker!!. firstName } else { Unknown } userNames.add(name) } return userNames } companion object { private var INSTANCE: Repository? = null val instance: Repository? get() { if (INSTANCE == null) { synchronized(Repository::class.java) { if (INSTANCE == null) { INSTANCE = Repository() } } } return INSTANCE } } // keeping the constructor private to enforce the usage of getInstance init { val user1 = User(Jane,) val user2 = User(John, null) val user3 = User(Anne, Doe) users = ArrayList<Any?>() users.add(user1) users.add(user2) users.add(user3) } } Let's see what the automatic converter did: The list of users is nullable since the object wasn't instantiated at declaration time Functions in Kotlin like getUsers() are declared with the fun modifier The getFormattedUserNames() method is now a property called formattedUserNames The iteration over the list of users (that was initially part of getFormattedUserNames()) has a different syntax than the Java one The static field is now part of a companion object block An init block was added Note : The generated code is not compiled. Don't worry about it, we'll change it in the next steps. Before we move on, let's clean up the code a little bit. If we look in the constructor, we see that the converter made our users list a mutable list containing zeroable objects. While the list may actually be zero, let's assume it can't hold zero users. So let's do the following: Remove ? i User? in the User Type Declaration Remove ? i User? for the return type getUsers() so that it returns<User>List? Init block In Kotlin, the primary constructor cannot contain any code, so the initialization code is placed in init blocks. The functionality is the same. class Repository private constructor() { ... init { val user1 = User(Jane,) val user2 = User(John, null) val user3 = User(Anne, Doe) users = ArrayList<Any?>() users.add(user1) users.add(user2) users.add(user3) } } } Much of the init code handles initialization properties. This can also be done in the declaration of the property. For example, in the Kotlin version of our repository class, we see that the user property was initialized in the statement. private var users: MutableList<User>? = zero Learn more about initialization blocks from the official documentation. Kotlin's static properties and methods In Java, we use the static keyword for fields or functions to say that they belong to a class, but not to an instance of the class. This is why we created instance static fields in our Repository class. The Kotlin equivalent of this is the companion object block. Here you will also declare static fields and static functions. The converter created the paired and moved the INSTANCE field here. Handling singletons because</User> </Any?> </User> </Any?> </Any?> need only one instance of the repository class, we used singleton pattern in Java. With Kotlin, you can enforce this pattern at the compiler level by replacing the class search word with the object. Remove the private constructor and replace the class definition with the object repository. Also remove the companion object. object Repository { private were users: MutableList<User>? = zero fun getUsers():<User>List? { return users } val formattedUserNames: List<String> get() { val username: MutableList<String> = ArrayList (users!!. size) for (user in users) { was name: String name = if (user!!. last name != null) { if (user!!. first name != null) { user!!. first name + + user!!. last name } otherwise { user!!. next name } } otherwise if (user!!. first name != null) { user!!. first name } otherwise { Unknown } userNames.add(name) } return username } // keep the constructor private to enforce the use of getInstance init { val user1 = User(Jane,) val user2 = User(John, null) val user3 = User(Anne, Doe) users = ArrayList<Any?>() users.add(user1) users.add(user2) users.add(user3) } } } When using the object class, we only call functions and properties directly on the object, as follows: val formattedUserNames = Repository.formattedUserNames Note that if a property does not have a visibility modifier on it, it is public by default by default , as in the case of the formatted UserNames property in the repository object. Learn more about objects and companion objects from the official documentation. When you convert the repository class to Kotlin, the automatic converter made the list of users invalid because it was not initialized to an object when it was declared. As a result, for all use of the users object, non-zero claim operator!! must be used. (You will see users!! and user!! through the converted code.) It!! converts a variable to a non-zero type, so you can access properties or call functions on it. However, an exception will be thrown if the variable value is actually zero. By !!, you risk exceptions being thrown while driving. Instead, prefer to handle nullability by using one of these methods: Do a null control (if (users != zero) {...}) Using elvis operator ?: (covered later in the codeplate) Using some of kotlin standard features (covered later in the codeplate) Learn more about zero security from the official documentation. In our case, we know that the list of users does not need to be nullible, since it is initialized right after the object is constructed (in the init block). Thus, we can directly start the users object when we declare it. When you create instances of collection types, Kotlin offers additional help features to make the code more readable and flexible. Here we use a MutableList for users: private var users: MutableList ? = zero For your convenience<User>we can use the mutableListOf() function and provide the list item type. oppretter en tom liste som kan inneholde brukerobjekter.</User> </User> </Any?> </String> </String> </User> </User> </User> the data type of the variable can now be derived by the

compiler, removing the explicit type declaration for the Users property. private val users = mutableListOf<User>() We also changed was to val because users will contain an immutable reference to the list of users. Note that the reference is immutable, but the list itself is mutable (you can add or remove items). Since the user variable has already been initialized, remove this initialization from the init block: users = ArrayList<Any?>() Then the init block should look like this: init { val user1 = User(Jane,) val user2 = User(John, null) val user3 = User(Anne, Doe) users.add(user1) users.add(user2) users.add(user3) } With these changes, our user property is now not null and we can remove all unnecessary!! operator instances. Please note that you will still see compile errors in Android Studio, but continue with the next steps in the code discs to solve them. val username: MutableList<String?> = ArrayList (users.size) for (user in users) { was name: String name = if (user.lastName != null) { if (user.firstName != null) { user.firstName + + username } else { user.lastName } } otherwise if (user.firstName != null) { user.firstName } else { Unknown } userNames.add(name) } for userNames value, if you specify the type of ArrayList that holds strings, you can remove the explicit type in the declaration because it will be derived. val userNames = ArrayList<String>(users.size) Destructuring Kotlin allows the destruction of an object to a variety of variables by using a syntax called destruction declaration. We create several variables and can use them independently. For example, data classes support destruction so that we can destroy the user object in too loop to (first name, last name). This allows us to work directly with the first name and last name values. Update for loop as shown below. Replace all instances of user.firstName with first name and replace user.lastName with last name. for ((first name, last name) in users) { was name: String name = if (last name != null) { if (first name != zero) { first name + + last name } other { last name } } other if (first name != null) { first name } other { Unknown } userNames.add(name) } Read more about disposition of declarations in the official documentation. if expression The names in the list of usernames are not quite in the format we want yet. Since both last name and first name can be zero, we need to deal with nullability when building the list of formatted usernames. We want to show Unknown if one of the names is missing. Since the name variable does not change after it is set once, we can use val instead of var. Make this change first. val name: String Take a look at the code that specifies the name variable. It may look new to you to see a variable being set to equal, and if/other block of code. This is allowed because in Kotlin if, when, for and while they are expressions - they return a value. The last line </String?> </Any?> </User> </User> will be assigned to the name. The sole purpose of this block is to initialize the name value. Essentially, this logic presented here if the last name is zero is either set to first name or Unknown. name = if (last name != null) { if (first name != null) { first name + + last name } other { last name } } otherwise if (first name != null) { first name } other { Unknown } Read more about, when, for, and while in the official documentation. Elvis operator This code can be written more idiomatic using elvis operator ?:. The Elvis operator will return the expression on the left side if there is no zero, or the expression on the right side, if the left side is zero. So in the following code, first names are returned if there is no zero. If first name is zero, the expression returns the value on the right hand , Unknown: name = if (last name != null) { ... } else { first name ?: Unknown } Read more about the elvis operator in the official documentation. Kotlin makes it easy to work with strings with string templates. String templates allow you to refer to variables in string declarations by using the $ symbol before the variable. You can also place an expression in a string declaration by placing the expression in { } and using the $ symbol before that. Example: ${user.firstName}. The code currently uses string capture to combine the first name and last name in the user name. if (first name != null) { first name + + last name } Instead, replace the string catcher with: if (first name != null) { $firstName $lastName } Using string templates can simplify the code. Your IDE will show you warnings if there is a more idiomatic way to write your code. You will notice a squiggly underscore in the code, and when you hover over it, you will see a suggestion on how to refactor the code. Currently, you should see a warning that the name declaration can be linked to the task. Let's use this. Because the type of name variable can be deduced, we can remove the explicit string type declaration. Now our formattedUserNames look like this: val-formattedUserNames: List<String?> get() { val userNames = ArrayList<String>(users.size) for ((first name, last name) in users) { val name = if (last name != null) { if (first name != null) { $firstName $lastName } otherwise { last name } } other { first name ?: Unknown } userNames.add(name) } return username } We can make an additional tweak. Our ui logic shows Unknown in case the first and last names are missing, so we don't support zero objects. Thus, for the data type formattedUserNames replace list<String?> with list . val<String>formattedUserNames: List<String> Let's take a closer look at the formattedUserNames getter and see how we can make it more idiomatic. Right now, the code does the following: Creates a new list of strings Iterates through the list of users Constructing the formatted name for each user, based on the user's first and last name Returns</String> </String> </String?> </String> </String?>created list val formattedUserNames: List<String> get() { val userNames = ArrayList<String>(users.size) for ((first name, last name) in users) { val name = if (last name != null) { if (first name != null) { $firstName $lastName } otherwise { last name } } other { first name ?: Unknown } userNames.add(name) } return username } Kotlin provides a comprehensive list of collection transformations that make development faster and safer by expanding the properties of the Java Collections API. One of them is the map function. This function returns a new list that contains the results of applying the specified transform function to each item in the original list. So, instead of creating a new list and iterating through the list of users manually, we can use the map function and move the logic we had in for the loop inside the map body. By default, the name of the current list item used in the map is it, but for readability, you can replace it with your own variable name. In our case, let's call it user: val formattedUserNames: List<String> get() { return users.map { user -> val name = if (user.lastName != null) { if (user.firstName != null) { ${user.firstName} ${user.lastName} } otherwise { user.lastName ?: Unknown } } other { user.firstName ?: Unknown } name } } Note that we use the Elvis operator to return Unknown if user.lastname is null, since user.lastName is of type String? and a string is required for the name. ... else { user.lastname ?: Unknown } ... To simplify this even more, we can remove the name variable completely: val formattedUserNames: List<String> get() { return users.map { user -> if (user.lastName != null) { if (user.firstName != null) { ${user.firstName} ${user.lastName} } else { user.lastName ?: Unknown } else { user.firstName ?: Unknown } } } In addition to transformations, Kotlin Standard Library offers a wide variety of tools to manage your collections. From different collection types to a variety of operations that are specific to the general collection type or to different subtypes, such as List or Enter. In general, when working with collections, check to see if the functionality you plan to write in Kotlin is already implemented by the default library, and prefer to use it for your own implementation. We saw that the automatic converter replaced the getFormattedUserNames() function with a property called formattedUserNames that has a custom getter. Under the hood, Kotlin still generates a getFormattedUserNames() method that returns a list. In Java, we would reveal our class characteristics via getter and setter functions. Kotlin allows us to have a better differentiation between the characteristics of a class, expressed with fields and functions, actions that a class can do, expressed with functions. In our case, the repository class is very simple and does no actions, so it has only fields. The logic that triggered in the Java getFormattedUserNames() function is now triggered when you call</String> </String> </String> </String> of the formattedUserNames Kotlin property. Although we do not explicitly have a field that corresponds to the formattedUserNames property, Kotlin provides us with an automatic backing field named field that we can access if necessary from custom getters and setters. Sometimes, however, we will have some extra functionality that the automatic support field does not provide. Let's go through an example. Inside our Repository class, we have a mutable list of users who are exposed in the getUsers() feature that was generated from our Java code: funny getUsers (): List<User>? { Return users } The problem here is that by returning users, any consumer of the Repository class can change our list of users - not a good idea! Let's fix this using a support property. First, let's rename users _users. Select the variable name, right-click to > Rename the variable. Then, add a publicly immutable property that returns a list of users. Let's call it users: private val _users = mutableListOf<User>() val users: List<User> get() = _users At this point, you can delete the getUsers() method. With the change above, the private _users property becomes the support property of public users property. Outside the repository class, the _users list cannot be changed, as class consumers can only access the list through users. The Convention for Support Properties is to use a leading underscore. Learn more about properties from the official documentation. Full code: object Repository { private val _users = mutableListOf<User>() val users: List<User> get() = _users val formattedUserNames: List<String> get() { return _users.map { user -> if (user.lastName != null) { if (user.firstName != null) { ${user.firstName} ${user.lastName} } otherwise { user.lastName ?: Unknown } } otherwise { user.firstName ?: Unknown } } } init { val user1 = User(Jane,) val user2 = User(John, null) val user3 = User(Anne, Doe) _users.add(user1) _users.add(user2) _users.add(user3) } } } But if we want to reuse the same formatting logic in other classes, we must either copy and paste it or move it to the user class. Kotlin provides the ability to declare functions and properties outside of any class, object or interface. For example, the MutableListOf() function we used to create a new instance of a list is already defined in Collections.kt from kotlin standard library. In Java, when you need some tool functionality, you will most likely create an Util class and declare that functionality as a static feature. In Kotlin, you can declare top-level features, without having a class. However, Kotlin also provides the ability to create expansion features. These are features that expand a specific type but are declared outside of type. To expand the to a class, either because we do not own the class or because it is not open for inheritance, Kotlin created</String> </User> </User> </User> </User> </User> </User> declarations called extensions. Kotlin supports expansion features and expansion properties. The visibility of expansion features and properties can be limited by using visibility modifiers. These limit their use only to classes that need the extensions, and do not pollute the namespace. For the user class, we can either add an extension feature that calculates the formatted name, or we can keep the formatted name in an extension property. It can be added outside the repository class, in the same file: // extension function fun User.getFormattedName(): String { return if (last name != null) { if (first name != null) { $firstName $lastName } otherwise { last name ?: Unknown } } other { first name ?: Unknown } } // extension property val User.userFormattedName: String get() { return if (last Name != null) { if (first name != null) { $firstName $lastName } } otherwise { last name ?: Unknown } } otherwise { first name ?: Unknown } } // use: val user = User(...) val name = user.getFormattedName() val formattedName = userFormattedName We can then use extension features and properties as if they are part of the user class. Because the formatted name is a property of the user class and not a functionality in the Repository class, let's use the extension property. Our Repository file now looks like this: val User.formattedName: String get() { return if (last name != null) { if (first name != null) { $firstName $lastName } otherwise { last name ?: Unknown } } otherwise { first name ?: Unknown } object Repository { private val _users = mutableListOf<User>() val users: List<User> get() = _users val formattedUserNames: List<String> get() { return _users.map { user -> user.formattedName } init { val user1 = User(Jane,) val user2 = User(John, null) val user3 = User(Anne, Doe) _users.add(user1) _users.add(user2) _users.add(user3) } } Kotlin Standard Library uses extension features to extend the functionality of multiple Java APIs. many of the features of Iterable and Collection are implemented as extension features. For example, the map feature we used in an earlier step is an expansion feature on Iterable. In our repository class code, we add additional user objects to _users list. These conversations can be made more idiomatic using Kotlin scope features. If you only want to run code in connection with a specific object, without having to access the object based on its name, Kotlin offers 5 scope features: let, use, with, run, and also. These features make the code easier to read and more concise. All scope functions have a recipient (this), can have an argument (it) and can return a value. Here's a handy cheat sheet to help you remember when to use each feature: Download the scope features cheat sheets from here. Since we configure our _users object in our repository, we can make the code more idiomatic using the search function: init { val user1 = User(Jane,) val user2 = User(John, zero) val user3 </User> </User> </User> Doe) _users.apply { // this == _users add(user1) add(user2) add(user3) } } Learn more about scope features from the official documentation. In this codeplate, we covered the basics you need to start converting your code from Java to Kotlin. This conversion is independent of your development platform and helps ensure that the code you write is idiomatic Kotlin. Idiomatic Kotlin makes writing code short and cute. With all the features Kotlin provides, there are so many ways to make the code safer, more concise and more readable. For example, we can even optimize our repository class by _users the list of users directly in the statement, get rid of init block: private val users = mutableListOf(User(Jane,), User(John, zero), User(Anne, Doe)) We covered a wide variety of topics, from handling nullability, singletons, strings and collections to topics such as expansion features, top-level features, features and scope features. We went from two Java classes to two Kotlin classes that now look like this: User.kt computer class User (was first name: String?, was last name: String?) Repository.kt val User.formattedName: String get() { return if (last name != null) { if (first name != null) { $firstName $lastName } otherwise { last name ?: Unknown } another { first name ?: Unknown } } object Repository { private val _users = mutableListOf(User(Jane,), User (John, zero), User(Anne, Doe)) val users: List<User> get() = _users val formattedUserNames: List<String> get() = _users.map { user -> user.formattedName } } Here is a TL; DR of Java functionality and their mapping to Kotlin: Java Kotlin final object val object is equal to() == == ==== Class that contains only data class Initialization in constructor initialization in init block static fields and functions fields and functions declared in a companion object Singleton class object Note that one of the advantages of Kotlin is that it is 100% interoperable with Java programming language. Call Java-based code from Kotlin, or call Kotlin from Java-based code. You can have as little or as much of Kotlin in the project as you want. To learn more about Kotlin and how to use it on your platform, take a look at these resources: resources:</String> </User>