Mastering ProGuard in Android

When building Android applications, we might have used ProGuard in our project. In this blog, we are going to learn about all the features and how to use ProGuard efficiently in Android.

So, let break it down in the following points,

- What is ProGuard?
- How it is helpful for our application?

- How to use it in our project?
- Important things to Note.

What is ProGuard?

ProGuard is a free java tool in Android, which helps us to do the following,

- Shrink(Minify) the code: Remove unused code in the project.
- Obfuscate the code: Rename the names of class, fields, etc.
- Optimize the code: Do things like inlining the functions.

In short, ProGuard makes the following impact on our project,

- It reduces the size of the application.
- It removes the unused classes and methods that contribute to the 64K method counts limit of an Android application.
- It makes the application difficult to reverse engineer by obfuscating the code.

How it is useful for our application?

In Android, proguard is very useful for making a production-ready application. It helps us to reduce the code and make apps faster. Proguard comes out of the box by default in Android Studio and it helps in a lot of ways, few of them are mentioned below,

- It obfuscates the code, which means that it changes the names to some smaller names like for **MainViewModel** it might change the name to **A**. Reverse Engineering of your app becomes a tough task now after obfuscating the app.
- It shrinks the resources i.e. ignorers the resources that are not called by our Class files, not being used in our android app like images from drawables, etc. This will reduce the app size by a lot. You should always shrink your app to keep it light weighted and fast.

How to use it in our project?

To enable Proguard in your project, in the app's build.gradle add,

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-an

    }
}
```

Here, we have `minfyEnabled` as true, it activates the proguard which take from the file,

```
proguard-android.txt
```

It is under the release block, which means that it will only be applied to the release of the build we generate.

But it can be too much sometimes when the proguard removes too much code and it might break your code for the flow.

So, configuring the code we have to add some custom rules to make sure we remove the set of code from obfuscating. We can fix this by writing out custom rules in our proguard and it will respect while generating the build.

Now, let us see how we can write custom rules in proguard.

1. Keeping class files

Let say we have a data class, which is needed by some API to perform but which generating a build we obfuscate the class. For example, we have a User data class,

```
data class User(val id: String = "")
```

and we want not to obfuscate the class which generating build then to ignore it from obfuscating we use **`@Keep`** annotation and update the code like,

```
@Keep
data class User(val id: String = "")
```

This annotation helps the class to be ignored by using proguard while minified. This will preserve the class and its member functions even when they are not in use.

We can also use,

```
-keep
```

to preserve options of class while generating the build. Using **-keep** over **@Keep** we get more control over what to preserve and what not to.

But, we can also preserve the key of the **id** field in data model class by using @SerializedName (when using Gson library) like,

```
data class User(@SerializedName("id")
                val id: String = "")
```

If you notice here, we are not using @Keep.

2. Keeping the members for a class

Let's say we want to preserve only the class members and not the class while shrinking, then we use,

```
-keepclassmembers
```

in the proguard rule file. This will help us to ignore members of a specific class.

Consider the above User class, and we want to preserve all the public methods inside it. We write the rule like,

```
-keepclassmembers class com.mindorks.sample.User{
    public *;
}
```

Here, the class User keeps all the members all which have public modifiers.

3. Keeping names of the class and members

Let's say we want to keep all the same names of the class and members of a class if it's being used in the code i.e. if the class is not used it would be shrunk by proguard but not obfuscated because it has already been shrunk there is no need of obfuscation.

To do the above task we use,

```
-keepnames
```

Practical use of it looks like,

```
-keepnames class com.mindorks.sample.GlideModule
```

Here, if the GlideModule would keep all of its names of the class and the member function.

4. Using any Library in Android

When using any library we might want to write some custom rules for proguard. There might a case when the library throws a warning in the logcat or they might not even have their own proguard rules!!!

To fix that we need to add custom rules at our application side. For example, if we start getting warnings from any library then we add,

```
-dontwarn com.somelibrary.annotations.*
```

in our proguard rules and then we won't see any warning coming up in our logs.

And to write custom rules for the library you can write it like any other rule for your own class.

5. Only Obfuscate your project

Consider a very rare use case, where you just want to obfuscate the code and not shrink any resource. This is a very rare use case but might be useful for some small libraries, then we write the flags like,

```
-dontshrink
-dontoptimize
```

This will help us to not shrink and optimize the code and just obfuscate.

6. Maintaining annotations

While building the app, ProGuard removes all the annotation and it might still work fine for some set of codes in your project. But let's say if we need the annotations to not be removed then we have the option like,

```
-keepattributes *Annotation*
```

Here, it keeps the attributes for all annotations to be preserved in your app. It is by default present in our rules.

7. Optimization

After writing this amount of rules in ProGuard, we might need to provide an extra layer of optimization for our app. First, we update the **build.gradle** file like,

```
android {
  buildTypes {
    release {
      proguardFiles getDefaultProguardFile('proguard-andr
    }
  }
}
```

Now, generally, we don't use this option but the use case here is we have to perform an extra level of optimizations.

To increase the number of cycles in optimization for example like we want to check if the optimization is done properly or not and if it's not done it will optimize it again till a certain number of times we use,

```
-optimizationpasses 5
```

Here, it will run the optimization upto 5 times to make it more optimized.

Now, consider an example where we want to optimize our final classes more at a granular level compared to what it was before, we use,

```
-optimizations class/marking/final
```

Here, the final classes will be optimized upto 5 times max or it might even end early if the optimization is already done.

Now, if we want to optimize the private fields now we use,

```
-optimizations field/marking/private
```

The majority of times the optimization is done for the first time.

If we don't want to optimize at all we use,

```
-dontoptimize
```

This is how we can work in different ways to make our app more secure and lighter using proguard.

Important things to Note:

- Do not use something like *MainFragment.class.getSimpleName()* as a fragment TAG. Proguard may assign the same name (A.class) to two different fragments in different packages while obfuscating. In this case, two fragments will have the same TAG. It will lead to the bug in your application.
- Keep your mapping file of the Proguard to trace back to the original code. You may have to upload it at different places like PlayStore Console for seeing the original stack-trace of the crashes.

If you want to write better ProGuard rules, checkout for tips to write better rules.

Happy learning.

Team MindOrks :)