Practical Instructions

Firebase Instructions (just in case)

1. In Xcode, in the menu bar, navigate to `File -> Add Package dependencies`.
2. In the dialog that opens, enter the following URL in the top-right search bar: https://github.com/firebase/firebase-ios-sdk
3. Click on the `Add Package` button, then there will be a little bit of loading.
4. In the `Choose Package Products for firebase-ios-sdk`, click on the `Add package` button.
5. This will add firebase-ios-sdk to your project.

Independent Quick Lookaround Guide

Prestep

1. In Android Studio, navigate to `File -> Open`. In the file dialog, navigate to your local repository and then to the subdirectory `/full-example`. The project should be compilable and runnable.

What to do:

1. Make sure you can run all three platforms (iOS will only work if you have a MacBook of course).
2. If you are curious how something works or is set up, check out the guide(s) below or do an unguided look around in whatever interests you. The idea is to get familiar with the code base before we dig into the practicals.

How the UI is organised and navigation is set up

1. Look around the `App.kt` file, in `composeApp/src/commonMain/kotlin/com.jetbrains.cameraapp`.
2. Notice navigation is set up in line 63-86. We are using the new(ish) JetBrains navigation library, and it supports type-safe navigation.
3. Curious about a particular screen? Check out the corresponding `Composable` function.

How the platform-specific cameras work

1. Look around the `CameraScreen.kt` file in `composeApp/src/commonMain/kotlin/com.jetbrains.cameraapp.camera`, and notice we are defining an `expect` function that's also a `Composable`. The platforms will each implement their `actual fun CameraScreen` in a platform-specific way.
2. If you're curious about how Android is implemented, head over to the `androidMain` source set - notice we are using the Camera2(x) library.
3. If you're curious about how iOS is implemented, head over to the `iosMain` source set. Notice here we are checking whether there is a camera or not, and then either showing the camera preview or just a `Select file` button (that is used with the FileKit library).
4. If you're curious about how Desktop is implemented, head over to the `desktopMain` source set. Notice here we're implementing the Desktop to be a simple `Select file` button (that is used with the FileKit library).

How the platform-specific photo filters work

1. Look at the `Filter.kt` file in `composeApp/src/commonMain/kotlin/com.jetbrains.cameraapp.filter`, and notice we're creating an `interface` for all filters to implement.
2. Look at the `BlackAndWhiteFilter.kt` file in the same directory. Notice we expect each platform to provide a B&W filter function that returns something of type `BlackAndWhiteFilter`. The same goes for the Gaussian blur filter.
3. Can you find the two platform-specific filters for each platform? It would be in `androidMain / iosMain / desktopMain`, and we need to implement the `actual fun` providing the two filters.

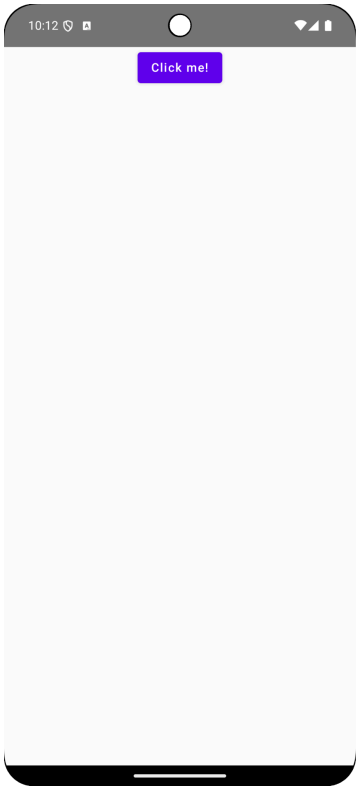
Practical 1: Splash Screens

General guideline: do something you're **unfamiliar** with / **animate** a static splash screen.

1.1 Add a static splash screen to the Android app

Prestep

1. In Android Studio, navigate to File -> Open. In the file dialog, navigate to your local repository and then to the subdirectory /practical1/starter. The project should be compilable and runnable and will look like this.



Add the splash screen compatibility library

1. Navigate to the `composeApp/src/build.gradle.kts file`.
2. Add the splash screen library dependency to the androidMain source set. It should look something like this:

```
sourceSets {
    androidMain.dependencies {
        …
        implementation("androidx.core:core-splashscreen:1.0.1")
    }
    commonMain.dependencies {
        …
    }
}
```

3. Remember to sync Gradle.

Import the .svg file as a vector asset

1. In Android Studio's menu bar, navigate to `File > New > Vector Asset`.

2. In the `Configure Vector Asset` dialog, select the `Local File` radio button.
3. In the `Path file` selector control, navigate to the `camera.svg` file (in your local repository) under `assets/practical1/android` and click on the Open button.
4. Set the size of the vector asset to be 100 dp x100 dp.
5. Click on the Next button.
6. In the `Confirm Icon Path` dialog, ensure that the source set (`main` or `androidMain`) and output file (`res/drawable/camera.xml`) are set correctly, and click on the `Finish` button.
7. In the `Project` navigator window, ensure the corresponding XML file appears in the output directory as above.

Make the vector drawable fit

The Android splash screen API has some dimension requirements for the splash screen icon:

- Icon with an icon background: must be 240×240 dp and fit within a circle 160 dp in diameter.
- Icon without an icon background: must be 288×288 dp and fit within a circle 192 dp in diameter.

This is because the splash screen API applies a circle mask to the screen. Conforming to the above requirements ensures that the entire logo fits into the circle mask's visible area. You can read more about this here.

An easy way to do this is as follows:

1. Open the vector drawable XML file, and note the viewport width (`android:viewportWidth`) and height (`android:viewportHeight`)
2. Wrap all <`path`> elements into a top-level <`group`> element with the following attributes:
    a. `android:pivotX="x"` where x is the value is half `android:viewportWidth` determined in step 1 (without dp)
    b. `android:pivotY="y"` where y is the value is half `android:viewportHeight` determined in step 2 (without dp)
    c. `android:scaleX="0.4"` (this value is based on the width of your vector asset, mine was 100dp)
    d. `android:scaleY="0.4"` (this value is based on the height of your vector asset, mine was 100dp)

Create a splash screen theme

It's now necessary to create a customized splash screen theme:

1. Right-click on `composeApp/src/androidMain/res/values`

2. Select `New > Values resource file` in the menu
3. Call the new values resource file something like "`splash_screen_theme.xml`" and click on the OK button
4. Edit the resulting file to look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Theme.App.Splash" parent="Theme.SplashScreen">
        <item name="windowSplashScreenBackground">
            #E24462
        </item>
        <item name="windowSplashScreenAnimatedIcon">
            @drawable/camera
        </item>
        <item name="postSplashScreenTheme">
            @android:style/Theme.Material.Light.NoActionBar
        </item>
    </style>
</resources>
```

Let's analyze our new splash screen theme. The splash screen theme, `Theme.App.Splash`, inherits from the `Theme.SplashScreen` theme defined by the Android splash screen library. We are overriding three of its attributes:

`windowSplashScreenAnimatedIcon` to define the vector drawable acting as our logo

`windowSplashScreenBackground` to define the background color of the rest of the screen

`postSplashScreenTheme` to define the next theme - this will most probably be the theme of the rest of the application

Apply the splash screen theme

1. Navigate to the `composeApp/src/androidMain/res/AndroidManifest.xml` file
2. Change the theme of both the application and the first `Activity` to `Theme.App.Splash`.

This could look as follows. `MainActivity` is the only `Activity` in the app.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application
        …
        android:theme="@style/Theme.App.Splash">
        <activity
            …
            android:name=".MainActivity"
            android:theme="@style/Theme.App.Splash">
            …
        </activity>
    </application>
</manifest>
```

Install the splash screen

1. Open up the first activity in the application. In my example app, this is `MainActivity`.
2. Add a call to `installSplashScreen()` before calling `setContent{...}`
3. Import the function as the IDE suggests.

Run the application

1. Select `composeApp` in the `Run Configurations` menu, then click on the `Run` button.
2. The splash screen with the logo will show briefly, then disappear to show the first activity content.

Troubleshooting

It might be necessary to change the scaling in the section Making the vector drawable fit to ensure no part of the icon is cut off.

1.2 Add a static splash screen to the iOS app

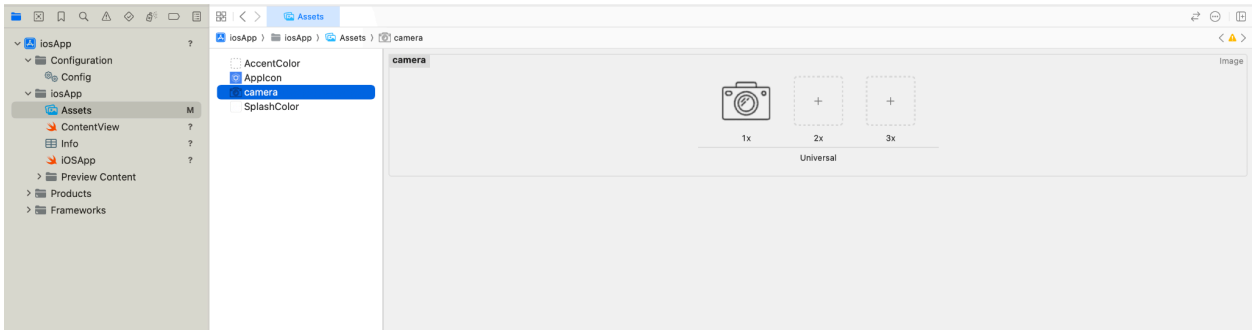
Hop over to Xcode

1. In Android Studio, navigate to `iosApp` and right-click on `iosApp.xcodeproj`
2. Select `Open In > Xcode`

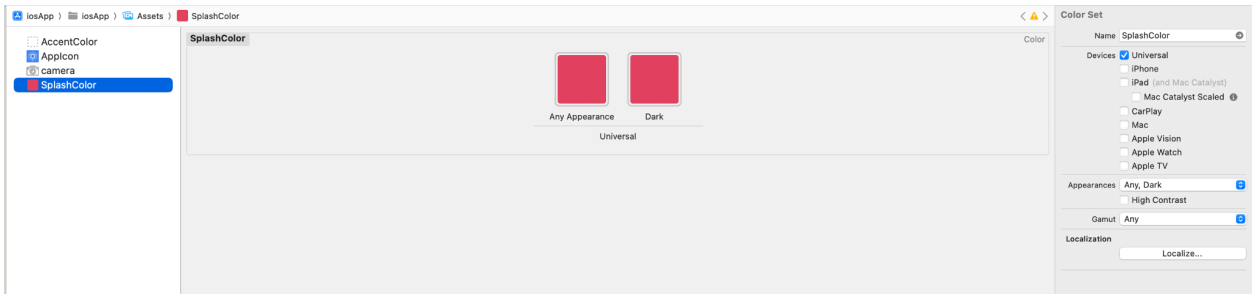
The project will now be correctly opened in Xcode.

Add the image and color to your Xcode project

1. In Xcode, navigate to `iosApp > iosApp` and open up `Assets`
2. Drag and drop your logo from `/assets/practical1/ios` into the white Assets view.

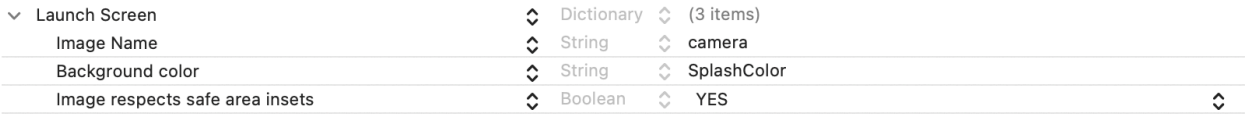


3. Right-click on the `Assets` view and select `New Color Set.`
4. Open the `Attributes inspector`. You may have to open the `Inspectors` panel by opening the button on the top right button  , then  .
5. Click on `Any Appearance` in the `Color` panel.
6. Fill in `#E24462` in the right-side panel's `Hex` value and set the `Name` to `SplashColor`
7. Repeat for the `Dark` appearance.



Create the Launch Screen

1. In the left side panel, navigate and click on the top-level `iosApp` (and make sure to be on the `iosApp` target)
2. In the middle panel, select `Info`
3. In the `Customized iOS Target Properties` table edit the property `Launch Screen`



4. Add three properties underneath `Launch Screen` (there is autocomplete) using right-click-> Add row:
    a. `Image Name`, which is the image you added in the previous step, without the extension.
    b. `Image respects safe area insets`, which is whether to respect safe areas.
    c. `Background color`, which is `SplashColor`.

Run the iOS app to see the launch screen

If you have run the app before, there may have been some caching issues and the launch screen won't show. To fix this, a combination of the following might be necessary:

- Navigate to the menu `Product > Clean Build Folder`
- Hard-close the app already running. You need to swipe up from the bottom, and then close the specific app by swiping it up.
- Delete the app from the device.

You can then run the app as usual and enjoy the very brief showing of the launch screen.

1.3 Add a static splash screen to the Desktop app

1. Add the `png` image from `assets/practical1/desktop` to `assets/common` in `desktopMain` (you will have to create the directories).
2. Update the `build.gradle.kts` file for `compose.desktop`:

```
compose.desktop {
   application {
       …
       nativeDistributions {
           targetFormats(TargetFormat.Dmg, TargetFormat.Msi,
TargetFormat.Deb)
           …
           appResourcesRootDir =
           layout.projectDirectory.dir("src/desktopMain/assets")
                       jvmArgs +=
           "-splash:${'$'}APPDIR/resources/camera.png"
       }
   }
}
```

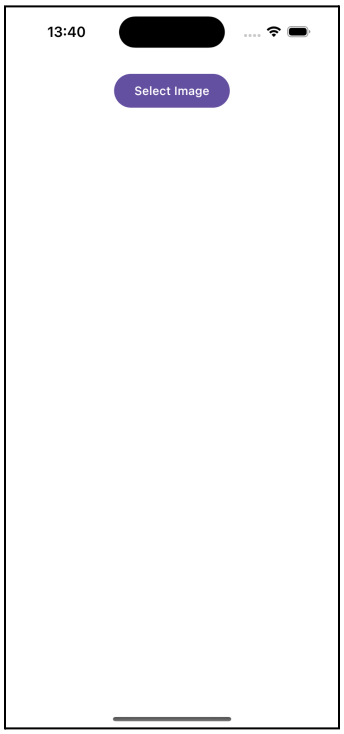
3. Remember to sync Gradle.
4. In the terminal: `./gradlew composeApp:runDistributable`

1.4 Animated splash screens for Android & iOS

See this article.

Practical 2: Debugging KMP shared code from Swift

Prestep

1. In Android Studio, navigate to File -> Open. In the file dialog, navigate to your local repository and then to the subdirectory /practical2/starter. The project should be compilable and runnable and will look like this (if you run it on iOS) (doesn't require an iPhone).
2. 

Running the application

1. Run the iOS application on the simulator.
2. Select an image from the gallery.
3. Click on the Blur button.
4. Notice that the image is not being blurred. Let's find out why!

In the next few steps, we will connect to a simulator process to debug it with lldb.

Getting the app process port number

1. Switch to the command line.

2. One way to get the port number is to use the following command:
   `xcrun simctl launch booted com.jetbrains.cameraapp.KotlinConfCameraApp`

   The following output will then be output, with the number at the end being the port number.
   `com.jetbrains.cameraapp.KotlinConfCameraApp: 23535`

Troubleshooting: If a new simulator is created and booted, don't panic! Just install the application on **that** simulator, by dragging the application from this path onto the simulator screen.
The path will be:
`deep-dive-into-kmp/practical2/starter/build/ios/Debug-iphonesimulator/KotlinConfCameraApp.app`
Run the `xcrun` command again to get the port number.

Launching lldb

1. Still in the command line, run the command: `lldb`
2. The `(lldb)` prompt will be displayed.

Attaching to the process

1. In the lldb prompt, use: `process attach -p port_number` (for me it was 23535).
2. It will take a little time, then say something like:
   `Target 0: (KotlinConfCameraApp) stopped.`

Importing to konan_lldb.py pretty printer

1. Let's start by finding this file's path on your machine. Mine is:
   `/Users/pamelahill/.konan/kotlin-native-prebuilt-macos-aarch64-2.1.20/tools/konan_lldb.py`
   Where `pamelahill` is my username, `macos-aarch64` is my architecture, and `2.1.20` is my project's version of Kotlin. Try to find yours by browsing your filesystem.
2. Now run in the lldb prompt:
   `command script import your_path_to_konan_lldb.py`
   I didn't get any feedback when I run it, so don't worry if that's the case.

   If you see memory address when you try to print out variables later in lldb, this command wasn't run correctly, maybe check the path / ask for help.

Setting a breakpoint

1. In the lldb prompt:
   `b -f GaussianBlurFilter.ios.kt -l 11`

   This sets a breakpoint on the file `GaussianBlurFilter.ios.kt`, line number 11.

   It will say something like this:
   ```
   Breakpoint 1: where =
   KotlinConfCameraApp.debug.dylib`kfun:com.jetbrains.cameraapp.filt
   er.IOSGaussianBlurFilter#filter(kotlin.String;kotlin.Float){} +
   728 at GaussianBlurFilter.ios.kt:11:5, address =
   0x000000010493ab14
   ```
2. Type: `continue`

   This continues all threads until the next breakpoint is hit. It will say something like this, based on your port number:
   ```
   Process 23535 resuming
   ```

Running to the breakpoint

1. In the app on the simulator, select an image.
2. Click the blur button, and you will see something like this:
   ```
   Process 23535 stopped
   * thread #13, queue = 'com.apple.root.default-qos', stop reason =
   breakpoint 1.1
       frame #0: 0x000000010493ab14
   KotlinConfCameraApp.debug.dylib`kfun:com.jetbrains.cameraapp.filt
   er.IOSGaussianBlurFilter#filter(_this=[],
   imagePath=/Users/pamelahill/Library/Developer/CoreSimulator/Devic
   es/E2CDBC51-474D-49C9-9EDD-C101335B4A44/data/Containers/Data/Appl
   ication/F728F6AD-D9BC-4836-ABC9-7A335A1D559E/tmp/CPH-2023-slides-
   0F48FFAA-7CB3-4782-9EC5-6CB00DFB84E3.jpeg, radius=25){} at
   GaussianBlurFilter.ios.kt:11:5
      8
      9       @OptIn(ExperimentalForeignApi::class)
      10      class IOSGaussianBlurFilter : GaussianBlurFilter {
   -> 11          override fun filter(imagePath: String, radius:
   Float) {
      12              // Validate file exists and is an image
      13              validateImageFile(imagePath)
   ```

```
    14
Target 0: (KotlinConfCameraApp) stopped.
```

3. The arrow is pointing to where in the code the debugger has stopped.
4. Next, type: `step`

   The arrow will step into the function and advance.

Debugging the blur filter

1. Using the following commands, navigate to line 19. Can you see the error? Can you fix it?

   `n` to step over
   `po` to print out a variable or expression

   You're welcome to use any lldb commands you want, of course. You can either use the `help` command or the slides for a quick guide. The solution is in the footnotes of this page.[1]

Practical 3: Modularisation

Presteps:

1. In Android Studio, navigate to `File -> Open`. In the file dialog, navigate to your local repository and then to the subdirectory `/practical3/starter.` The project should be compilable and runnable.

Creating the camera module

1. In Android Studio, click on `File -> New -> New Module`
2. In the `Create new module` dialog, select `Kotlin Multiplatform Shared Module` as the template type
3. In the right-hand panel, set `Module name` to `camera`, and `Package name` to `com.jetbrains.cameraapp.camera`
4. Click `Finish` button

Removing the Platform files

The Android Studio template creates unnecessary files.

---

[1] You need to use `imagePath` to set up the URL not `imagepath`, a variable declared in `App.mobile.ios.kt`.

1. Open the camera module, and delete:
    a. `commonMain/kotlin/com.jetbrains.cameraapp.camera/Platform.kt`
    b. Do the same for the `Platform` files in `androidMain` and `iosMain`.

Adding the desktop target

The Android Studio template doesn't have a desktop target added to the created module by default. Let's fix that.

1. Open the camera module in the `Project` view
2. Under the `src` folder, create a `desktopMain` directory and `kotlin` directory below that.
3. Right-click on the `kotlin` directory and select `Mark directory as -> Sources root`
4. Under the `kotlin` directory, create the package `com.jetbrains.cameraapp.camera`

Updating the camera module's build.gradle.kts file

1. Open the `camera` module's `build.gradle.kts`
2. Check that the following plugins are added to the plugin part of the DSL. The first two should already be there. We will be using Compose Multiplatform in this module, necessitating the 3rd and 4th plugins. Navigation requires serialization, so we need to add the 5th plugin to the section.

```
plugins {
    alias(libs.plugins.kotlinMultiplatform)
    alias(libs.plugins.androidKotlinMultiplatformLibrary)
    alias(libs.plugins.composeMultiplatform)
    alias(libs.plugins.composeCompiler)
    alias(libs.plugins.kotlinxSerialization)
}
```

3. The Android Studio template needs us to apply the default hierarchy template, and we also need to add the new JVM target for desktop. Add the following code right after the ios binary framework setup code.

```
kotlin {
    …

    applyDefaultHierarchyTemplate()

    jvm("desktop")

    sourceSets {
       …
```

```
}
```

4. Setup the desktop dependencies in the sourceSets section. Remember we'll be using `FileKit` to select the image from the file system for desktop.

```
getByName("desktopMain") {
    dependencies {
        implementation(libs.filekit.dialogs.core)
        implementation(libs.filekit.dialogs.compose)
    }
}
```

5. Sync Gradle

Moving the source file to the camera module

1. Move the file `CameraScreen.desktop.kt` from the composeApp module, `desktopMain` source set TO the `camera` module, `desktopMain` source set (specifically `camera/src/desktopMain/kotlin/com.jetbrains.cameraapp.camera`).
2. There may be some import issues, leave them for now.

Moving the common code to the camera module

1. Move the file `CameraScreen.kt` from the `composeApp` module, `commonMain` source set TO the `camera` module, `commonMain` source set (specifically `camera/src/commonMain/kotlin/com.jetbrains.cameraapp.camera`).
2. Update the `camera` module's `build.gradle.kts` file to include Compose and JetBrains' navigation library.

```
sourceSets {
        …
        commonMain {
            dependencies {
                implementation(libs.kotlin.stdlib)
                implementation(compose.runtime)
                implementation(compose.foundation)
                implementation(compose.material3)
                implementation(compose.ui)
                implementation(compose.components.resources)
                implementation(compose.components.uiToolingPreview)
                implementation(libs.navigation.compose)
            }
```

```
        }
    …
}
```

Moving the Android code to the camera module

1. Move the file `CameraScreen.android.kt` from the `composeApp` module, `androidMain` source set TO the `camera` module, `androidMain` source set (specifically `camera/src/androidMain/com.jetbrains.cameraapp.camera`).
2. Update the `camera` module's `build.gradle.kts` file to include the Camera 2 dependencies.

```
sourceSets {
        …
        androidMain {
            dependencies {
                implementation(libs.androidx.camera.core)
                implementation(libs.androidx.camera.camera2)
                implementation(libs.androidx.camera.lifecycle)
                implementation(libs.androidx.camera.view)
            }
    }
    …
}
```

Moving the iOS code to the camera module

1. Move the file `CameraScreen.ios.kt` from the `composeApp` module, `iosMain` source set TO the `camera` module, `iosMain` source set (specifically `camera/src/iosMain/com.jetbrains.cameraapp.camera`).
2. Update the `camera` module's `build.gradle.kts` file to include the `FileKit` dependencies.

```
sourceSets {
        …
        iosMain {
            dependencies {
                implementation(libs.filekit.core)
                implementation(libs.filekit.dialogs.core)
                implementation(libs.filekit.dialogs.compose)
            }
    }
```

```
        …
}
```

Sync and fix import issues

1. Sync Gradle
2. Fix any import issues in the camera module. There should only be one left, related to `CameraAppScreen`. We will fix that in the `navigation` module creation step.
3. Add the `camera` module to the `composeApp's commonMain` dependencies like so:

```
api(project(":camera"))
```

Creating the navigation module

1. Similarly, create a new module `navigation` to the project, with the package `com.jetbrains.cameraapp.navigation`. It must also support Android/iOS/Desktop.
2. Remember to remove the unnecessary `Platform` files.
3. Copy the file `CameraAppScreen.kt` from the `composeApp` module, `commonMain` source set TO the `navigation` module, `commonMain` source set.
4. Can you figure out how to update the `build.gradle.kts` file? You will need to:
   - Update the plugins. Hint: You need to add one related to serialization
   - Apply the default hierarchy template
   - Add the JVM desktop target
   - Setup the `desktopMain` sourceSet and add the `dependencies` block (nothing needed specifically here, it's just a placeholder)
   - Add the right dependencies to the `commonMain` block. Hint: They are related to serialization
5. Add the `navigation` module to the `composeApp's commonMain` dependencies like so:

```
api(project(":navigation"))
```

6. Do the same for the `camera` module's `commonMain` as it will also use this class.
7. Sync gradle / import / run Android/iOS/Desktop.

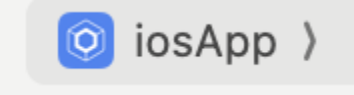
Practical 4: Fixing a Memory Leak

Presteps:

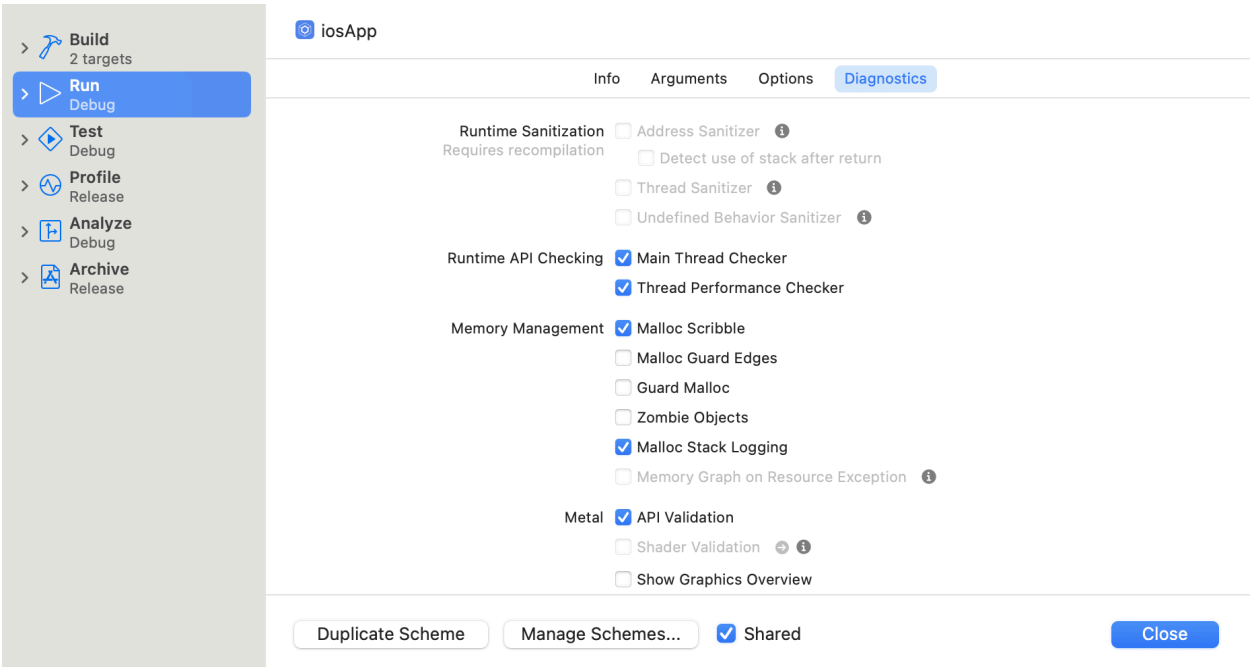
1. In Android Studio, navigate to `File > Open`. In the file dialog, navigate to your local repository and then to the subdirectory `/practical4/starter`. The project should be compilable and runnable.

Enable malloc debugging info on the scheme

1. In Android Studio, open the project in Xcode, by right clicking on the `iosApp/iosApp.xcodeproj` file, then `Open In -> Xcode`
2. The project will open in Xcode.
3. In Xcode, click on `iosApp` to open the scheme menu. It's at the top of the screen, next to the selected device.

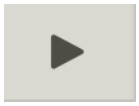


4. Select `Edit Scheme`
5. In the `Memory Management` section of the popup dialog, enable `Malloc Scribble` and `Malloc Stack Logging`, and click the `Close` button.

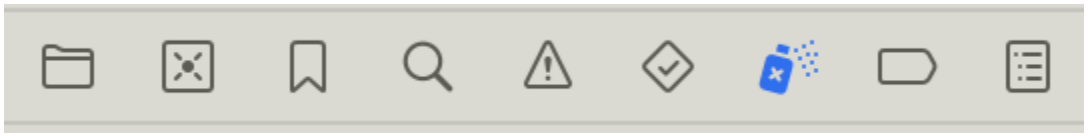


Run and create the leak

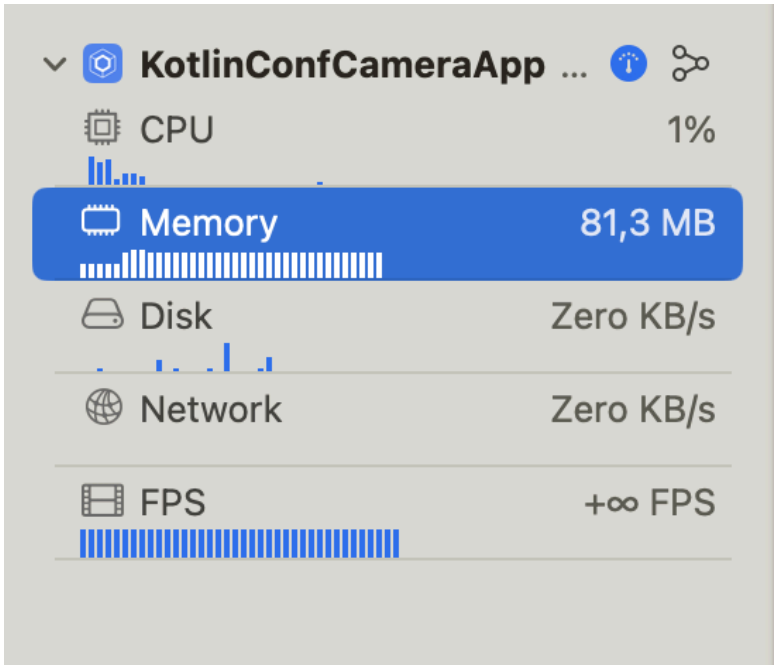
1. Click on the `Run` button at the top left of the screen



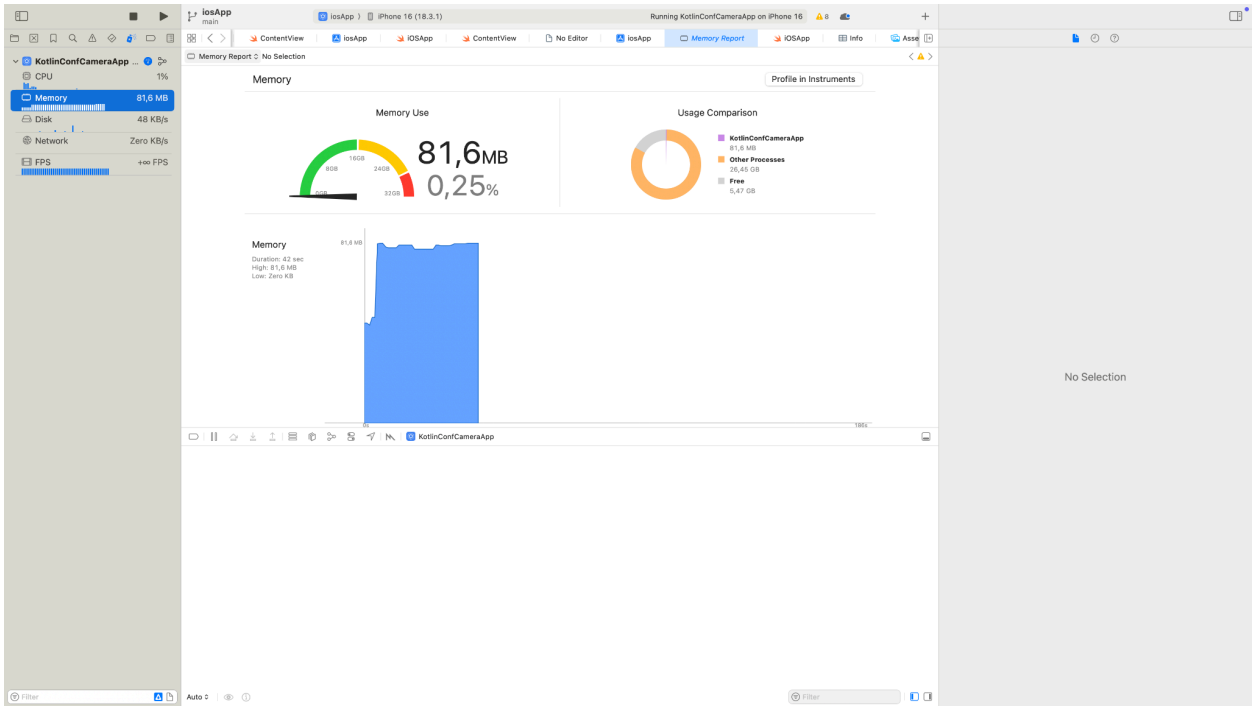
2. The application will then open on the simulator.
3. On the simulator, select an image, the preview screen will display with the image and three buttons.
4. In Xcode, open the `Debug navigator` at the top left icon menu of the screen. To me it looks like a spraypaint can.



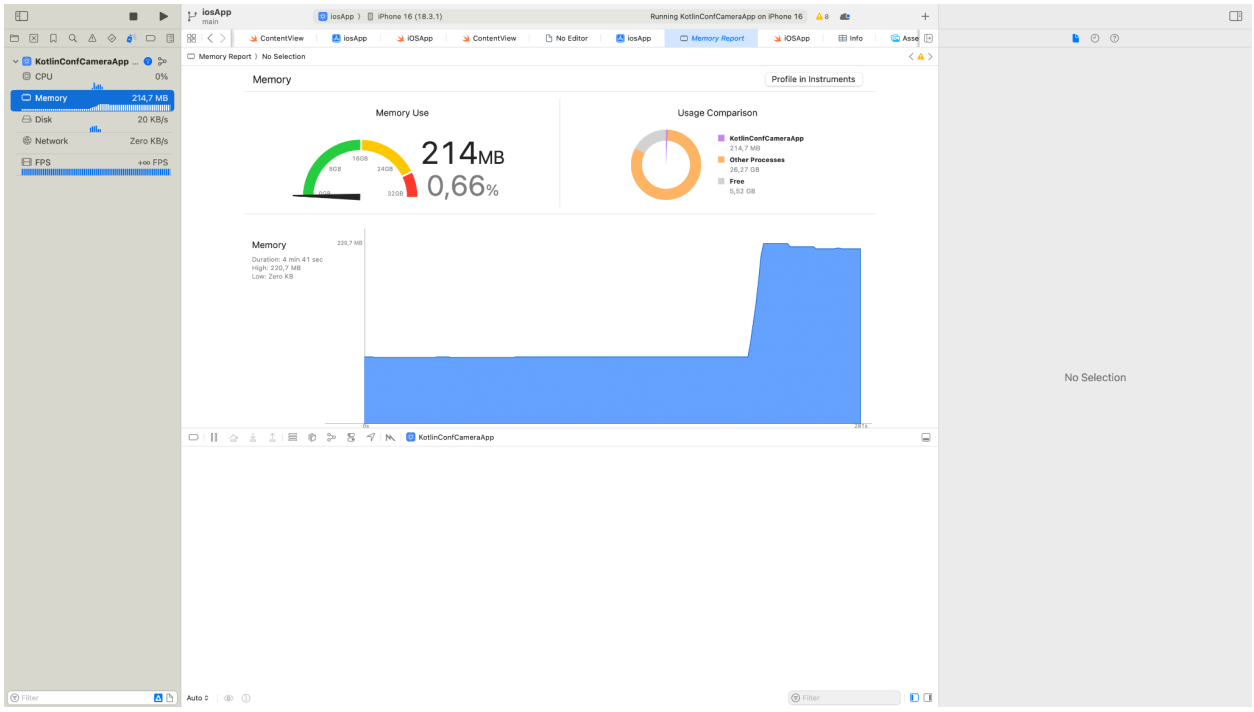
5. Click on the `Memory` section in the `debug navigator`.



6. The screen will then look like this.



7. Take note what the memory usage looks like. Then, on the simulator, press the B&W button a number of times - let's say 10.
8. Back in Xcode, look at the memory usage again. It might look like this, notice the increased memory usage.



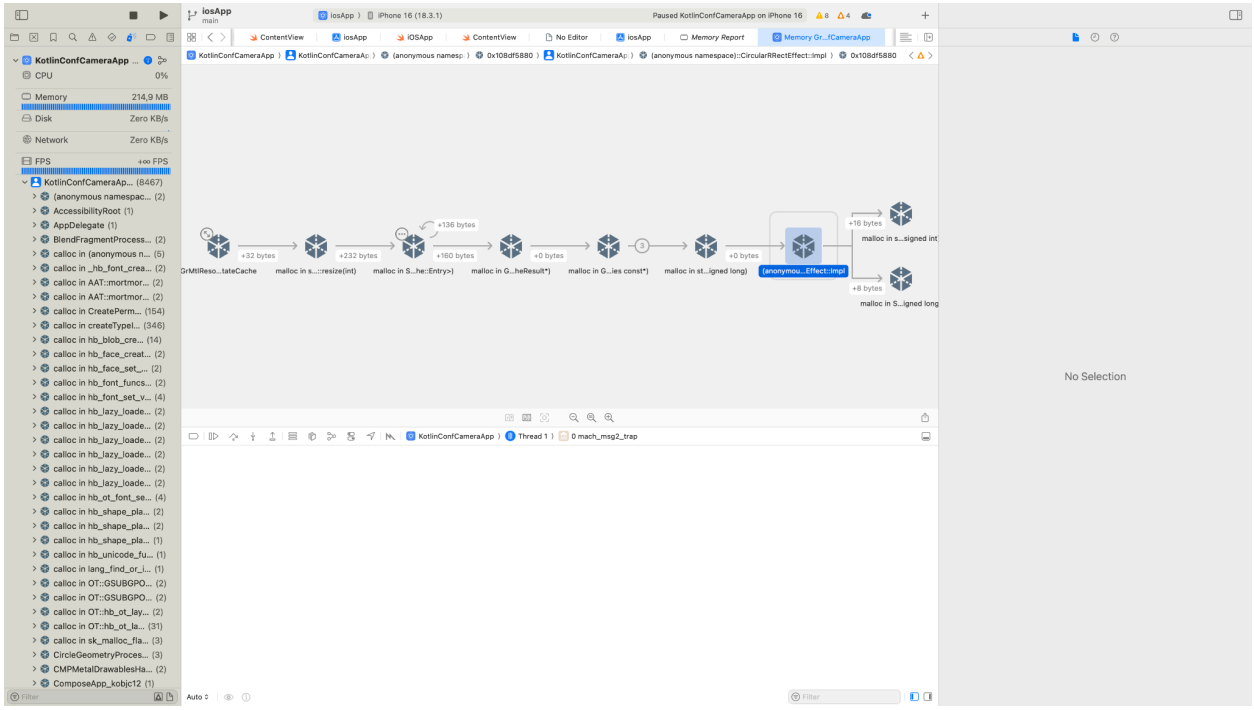
9. We have a memory leak!

Find the cause of the leak

1. Click on the `Debug Memory Graph` button. It's roughly in the middle of the screen, in another icon menu that looks like this.

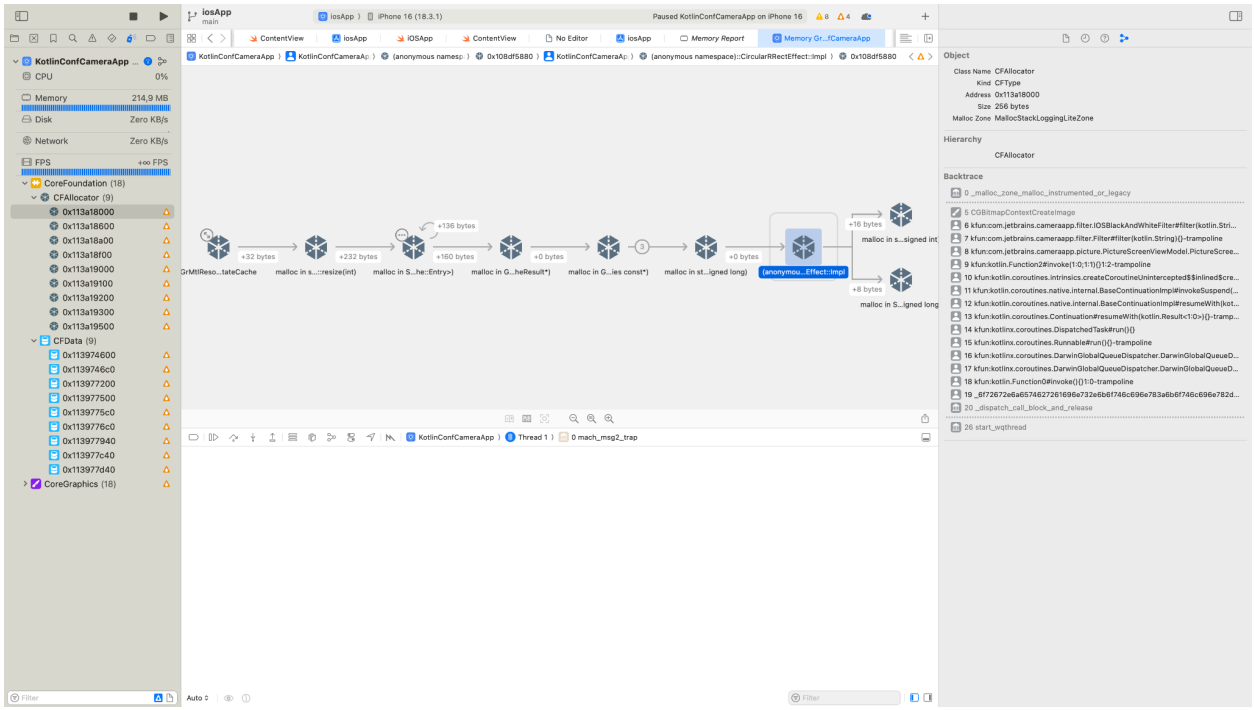
The icon looks like this.

2. The resulting screen looks like this, with a big graph in the middle.

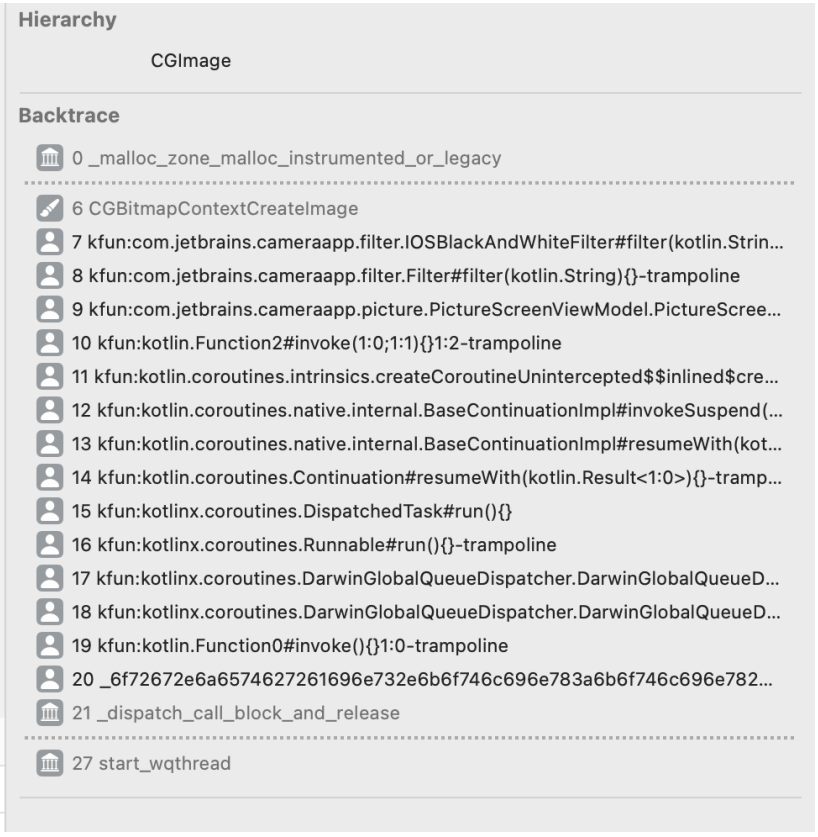


3. At the bottom left of the screen, filter for only leaked allocations by selecting the button at the bottom left of the screen. It looks like this.

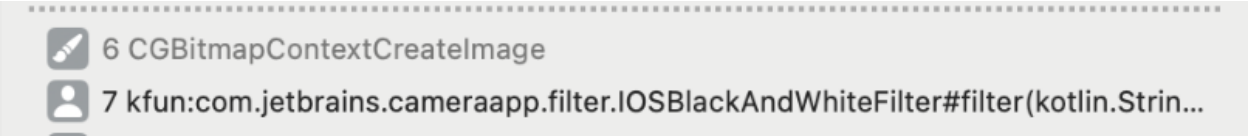
The resulting screen only shows the leaked allocations in the panel on the left.



4. Open up the `CoreGraphics` section, and then `CGImage`. We can see we are leaking `CGImages` somehow. Let's find out where.
5. Click on one of the allocations, then examine the panel on the right.



We can see that we are allocating memory for `CGImage` when we call `CGBitmapContextCreateImage`, in the `IOSBlackAndWhiteFilter` class, `filter` function.



Fix the leak and review

1. In Android Studio, open `composeApp/src/iosMain/kotlin/com.jetbrains.cameraapp.filter/BlackAndWhiteFilter.ios.kt` in the editor window.
2. Look for the usage of `CGBitmapContextCreateImage`. It should be on line 40, where we create the `grayscaleImage` variable.
3. Oops! We never release the memory! Let's fix that by going to the end of the null-check block and releasing the memory manually.

```
if (grayscaleImage != null) {
    …
    CGImageRelease(grayscaleImage)
}
```

4. Now, re-run the app and check the memory graph. Should be all fixed with no detected leaks.