React Native

A Native Code Integration Perspective

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor thesis

Degree Programme in Business Information Technology

Visamäki – Hämeenlinna University Centre – Finland

Fall, 2023.

Balogun Nathan

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology Abstract

Author Balogun Nathan Year 2023

Subject React Native: A native code integration perspective.

Supervisors Tommi Lahti

ABSTRACT

React Native is a popular cross-platform framework for building mobile apps that can run on both iOS and Android platforms using a single codebase. This thesis explores the capabilities of React Native for Android app development and the potential of using native code to enhance its functionalities.

The paper provides an overview of the React Native framework and its advantages and disadvantages. It briefly discusses the performance and compatibility of React Native with native Android code, and how native modules can be integrated into React Native projects to improve app functionality.

To demonstrate the potential of combining native code with React Native, the paper presents a few examples of mobile apps developed using React Native and how native code was used to optimize their performance and add missing features.

Overall, this thesis highlights the potential of using React Native for Android app development and demonstrates how native code can be used to obtain some functionalities. By briefly analysing the performance and compatibility of React Native with native code, and presenting a few examples of real-world apps, this thesis aims to give amateur programmers a glimpse into the possibilities of using React Native for Android app development.

Keywords React Native, Java, Native

Pages 42 pages and appendices 71 pages

Glossary

AOT	Ahead of time
API(s)	Application Programming Interface(s)
App	Application
ART	Android Runtime
CPU	Central Processing Unit
iOS	iPhone Operating System
JIT	Just in Time
JMF	Java Media Framework
JNI	Java Native Interface
NDK	Native Development Kit
NPM	Node Package Manager
SDK	Software Development Kit
UI	User Interface
VDOM	Virtual Document Object Module

Contents

1 Introduction 1

2 React Native 2

2.1 Overview of React Native 2

2.2 Advantages and Disadvantages of React Native 2

2.3 React Native Architecture 5

2.4 Development tools for React Native 6

2.5 Limitations of React Native 7

2.6 Comparison with Traditional Native Development Approaches 8

3 Native API 11

3.1 Native APIs 11

3.2 Importance of Native APIs in Mobile App Development 11

3.3 Native APIs in React Native 12

4 Java 15

4.1 What is Java? 15

4.2 Using Java to Enhance React Native Functionality 15

4.3 Additional Important Notes 16

5 Java vs React Native: A Comparison of Features 18

5.1 Incorporating Java into a React Native Project 23

6 Methodology 25

6.1 Data Collection 25

6.2 Results 25

6.3 Limitation 26

7 Native Code Injection (Practical Examples) 27

7.1 Important Subjects to Consider 27

7.2 Example 1 (Java Module with Strings) 31

7.3 Example 2 (Camera Module with External Libraries) 34

8 Reflection 38

9 Conclusion 39

9.1 Summary of Study 39

9.2 Implications for Developers and Organizations 39

9.3 Recommendations for Future Research 39

References 41

Appendices

Table 1

Appendix 1:	Native Module Class	1
Appendix 2:	Native Module Package	2
Appendix 3:	Register the package.	4
Appendix 4:	Rebuild the application.	5
Appendix 5:	Implement the module.	6
Appendix 6:	Manifest Permissions	8
Appendix 7:	Build.gradle dependency	9
Appendix 8:	Build.gradle defaultConfig	10
Appendix 9:	RNOpenCvLibraryModule.java	11
Appendix 10:	RNOpenCvLibraryPackage.java	14
Appendix 11:	OpenCV.js	15
Appendix 12:	Screens/CameraScreen.js	16
Appendix 13:	Styles/Screens/CameraScreen.js	20
Appendix 14:	CircleWithinCircle.js	22

Figures

Figure 1 Interpreted approach of React Native architecture. 6

Figure 2 Maven dependency in pom.xml 28

Figure 3 Android dependency in build.gradle 28

Figure 4 Importing the module. 35

Figure 5 Project build.gradle.....36

Figure 6 SDK build.gradle 36

Figure 7 Folder tree.....37

Tables

Table 1 Advantages of React Native 3

Table 2 React Native Pros and Cons.....3

Table 3 Pros and Cons of React Native Development.....4

Table 4 Limitations of React Native 7

Table 5 Developer Use Case 9

Table 6 Importance of Native APIs in mobile development.....12

Table 7 Commonly Used APIs in React Native.....13

Table 8 Best practices to consider when working with react native using Native APIs. .14

Table 9 Important methods and annotations to be considered when making a module. 29

Table 10 methods to be considered when rewriting the lifecycle methods. 29

1 Introduction

The proliferation of mobile devices has created an increasing demand for cross-platform mobile applications that can run on both iOS and Android platforms. The need for fast development cycles, code reuse, and cost-effective solutions has led to the rise of React Native as a popular framework for building cross-platform mobile applications.

React Native is an open-source framework that allows developers to write code in JavaScript and React and produce applications for both iOS and Android platforms. The framework offers several benefits for building cross-platform mobile applications, including a fast development cycle, a large and supportive community, and the ability to reuse code across platforms.(*What Is React Native?*, n.d.)

However, React Native also has limitations that must be considered when deciding on a development approach. This thesis provides an analysis of the capabilities and limitations of React Native in building cross-platform mobile applications and how they could be adressed with Java. The research includes a basic examination of the architecture and design of React Native, as well as a comparison with traditional native development approaches. The study also investigates the integration of Java projects into React Native and its ability to access native APIs and integrate with existing native code.

The goal of this thesis is to provide valuable insights for amateur developers who are new to react native and looking to build android mobile applications using native language in React Native. By understanding the strengths and weaknesses of React Native, developers can make informed decisions about the best approach for their specific needs.

This thesis research would cover the following questions:

- What are the key features and benefits of React Native for building android mobile applications?
- What are the limitations of React Native in building android mobile applications and how can they be addressed using java?
- How could native language be integrated into a react native project?

2 React Native

This chapter provides a brief overview of React Native. This chapter will also discuss some of the limitations of React Native, its pros and cons, architecture and development tools needed. Finally, the chapter will briefly compare react native to the traditional native development approach.

2.1 Overview of React Native

React Native is an open-source framework for building cross-platform mobile applications using JavaScript and React. (*(3) Exploring Key IT Technologies | LinkedIn*, n.d.)The framework allows developers to build native-like applications for both iOS and Android platforms using a single codebase. React Native was developed by Facebook and was first released in 2015. Since its release, it has gained popularity among developers and organizations due to its ability to build high-quality, performant mobile applications using a single codebase. (Dayanand, 2023; Paterska, n.d.)

React Native differs from other mobile app development frameworks in several keyways. One of the most significant differences is that React Native uses a single codebase for both iOS and Android app development. This means that developers can write code once and deploy it across both platforms, which can save time and reduce development costs. However, to achieve this cross-platform functionality, React Native has a unique architecture that differs from traditional native app development. In traditional native app development, each platform has its own set of components and code, which can result in duplication and increased development time. In contrast, React Native uses a bridge to communicate between the JavaScript code and the native components on each platform. This architecture allows developers to write a single codebase that can be easily translated into native components on both iOS and Android.

2.2 Advantages and Disadvantages of React Native

There are several advantages and disadvantages to using React Native for cross-platform mobile application development, including:

Table 2 Advantages of React Native

Cross-platform compatibility	React Native allows developers to build applications for both iOS and Android platforms using a single codebase, reducing development time and costs. ("Convert Figma to Flutter Seamlessly: Boost Your App Development")
High performance	React Native applications have native-like performance, providing a smooth user experience for end-users.
Easy to learn	React Native is built using JavaScript and React, making it easy for developers who are already familiar with these technologies to start building applications.
Large community	React Native has a large and active community of developers, providing a wealth of resources and support for building applications.

(*Benefits of Using React Native for Mobile App Development*, n.d.)

React Native is a popular cross-platform mobile app development framework that makes use of JavaScript to create native-looking apps. The benefit of code reuse is provided to developers, enabling them to write once and deploy across several platforms. There are some things to take in mind, though, as with any technology. To give readers a thorough grasp of React Native's capabilities in mobile app development, the following tables will examine React Native's advantages and disadvantages, as well as its advantages and limitations.

Table 3 React Native Pros and Cons

Pros	Cons
Cross-Platform out of the box: Target both Android and iOS from a single development team and codebase. Lower cost and time to market compared to native.	**Performance**: Not the best choice for high-performance apps, specifically those with graphically intensive or data-heavy workloads.
Native Modules: Capability to leverage the best of both worlds by supplementing React Native app with native modules.	**Developer Awareness**: Knowledge of the Bridge versus underlying platform is needed to avoid performance pitfalls and develop native modules.

Hot Reloading: Productivity boost during development by quickly seeing and testing changes without manually reloading the app.	**Reliance on Maintainers**: Developers must wait on maintainers of React Native to address issues and provide support for new features.

(*React Native vs Native*, n.d.)

Table 4 Pros and Cons of React Native Development

Pros	Cons
Cost efficient and cross platform solution	With the current React native version not being 1.0 (latest 0.62.2), we are yet to see the best from React Native
No difference between iOS and android	Common UI for both platforms (limited customisation capabilities).
Native Rendering and native UI component	React Native sometimes faces the lack of the third-party libraries and APIs. However, it can be solved by creating modules in Native language.
Open source and supported by Facebook.	More time spent on fixing small problems: None native code can make it complicated to address some device related issues.
React Native community is actively growing and developing	Complicated UI elements and animations are not the strong sides of React Native. Some components of the mobile apps cannot be written in JavaScript so developers should have some experiences in iOS/Android native app development too.

Web developers can learn React Native quicky and easily	In a single threaded JavaScript environment, performance problems might arise, and you need to be extra careful while building your UI.
Hot Reloading = faster debugging process.	Several little glitches in chrome debugging

(Bansal, 2021)

2.3 React Native Architecture

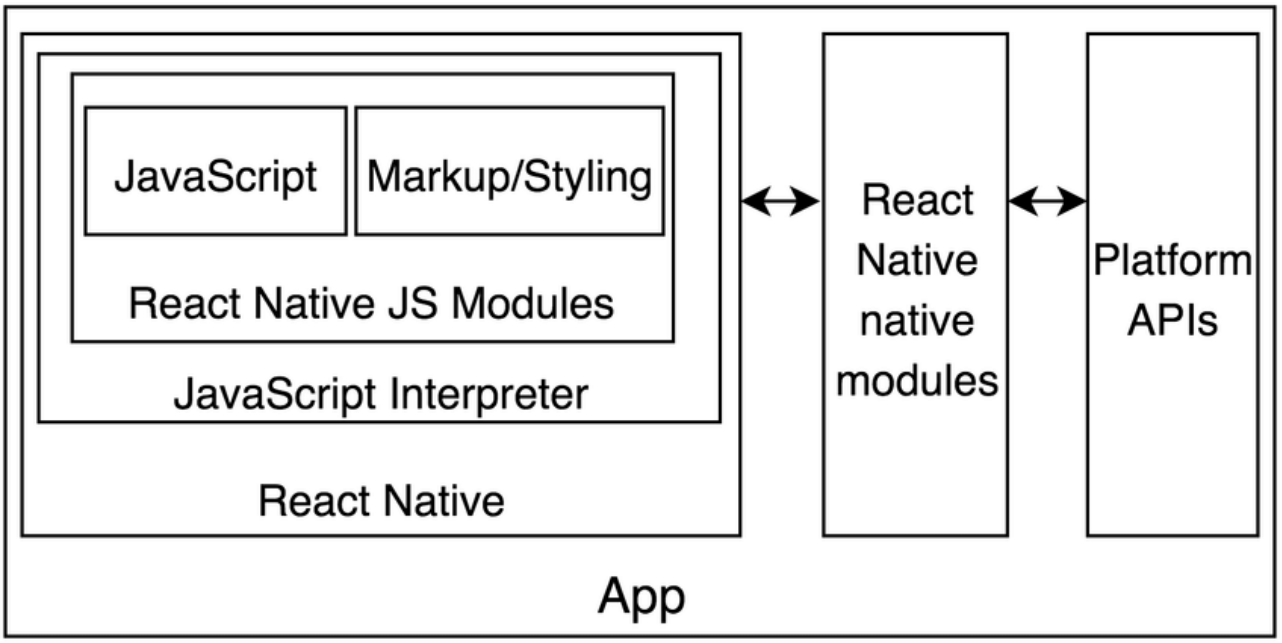
React Native architecture is unique compared to traditional native app development. As mentioned earlier, React Native uses a bridge to communicate between the JavaScript code and the native components on each platform. The bridge is responsible for sending and receiving messages between the JavaScript runtime and the native environment. This architecture enables React Native to provide a smooth and responsive user experience, as the bridge only sends the necessary data and updates to the native components when required.

React Native also utilizes a Virtual DOM (Document Object Model) which is a lightweight copy of the actual DOM tree. This allows React Native to manage UI updates more efficiently by minimizing the amount of data that needs to be sent between the JavaScript code and the native components. When a change occurs in the UI, React Native updates the Virtual DOM and then calculates the minimum number of changes needed to update the actual DOM.

In addition to the bridge and Virtual DOM, React Native also provides a set of core components and APIs that enable developers to build native mobile apps. These components include View, Text, Image, and TextInput, among others. React Native also provides third-party libraries and plugins that can be used to extend the functionality of the core components.

Understanding the architecture of React Native is crucial for developers who want to build cross-platform mobile apps using this framework. In the next section, the next discussion will be to explore the development tools available for building React Native apps.

Figure 1 Interpreted approach of React Native architecture.



(Biørn-Hansen & Ghinea, 2018)

2.4 Development tools for React Native

React Native provides several development tools that make it easier to build and test mobile apps. These tools include:

Expo: Expo is a toolchain that enables developers to build and test React Native apps without requiring them to configure native build tools or install any native dependencies. Expo provides a set of APIs for accessing native device functionality, as well as a set of pre-built UI components. Expo also includes a mobile app that developers can use to preview their apps as they build them.(Vatne, 2019)

React Native CLI: React Native CLI is a command-line interface that developers can use to create, build, and run React Native apps. React Native CLI requires developers to configure native build tools for each platform they want to target but provides more control over the build process.(Verma, 2023)

Visual Studio Code: Visual Studio Code is a popular code editor that supports React Native development. It provides a number of extensions that enable developers to write, debug, and test React Native apps directly from the editor.(*React JavaScript Tutorial in Visual Studio Code*, n.d.)

Android Studio and Xcode: Android Studio and Xcode are the official IDEs for developing Android and iOS apps, respectively. React Native integrates with these IDEs, enabling developers to build and run React Native apps directly from within the IDE.(RNDM, 2019)

Understanding the different development tools available for React Native is crucial for developers who want to build high-quality mobile apps. Each tool has its own strengths and weaknesses, so developers should choose the tool that best fits their needs and preferences.

2.5 Limitations of React Native

While there are many advantages to using React Native, there are also some limitations that should be considered when deciding whether to use the framework for cross-platform mobile application development, including: (Kundariya, 2020)

Table 5 Limitations of React Native

Limited support for native APIs	React Native may not have full support for all native APIs, which could limit the functionality of the applications built using the framework.
Dependence on third-party libraries	React Native depends on third-party libraries for some functionalities, which can increase development time and make it difficult to maintain the codebase over time.
Performance issues	React Native applications may experience performance issues, particularly when compared to traditional native applications. E.g., Memory Leakage, Launch Time, Navigator Issues, Large app size. (A, 2022; tagline, 2023)
Debugging	An advantage of React Native Framework is that it contains a built-in inspector for debugging your applications. But this built-in functionality of the framework also comes with some limitations. For instance, you might need a chrome debugger along with the built-in inspector to debug your code.

	This creates a dependency of the system on the chrome debugger to edit every element and its properties, which seems quite impossible.

2.6 Comparison with Traditional Native Development Approaches

When it comes to mobile app development, there are two main approaches: traditional native development and cross-platform development. Traditional native development involves writing separate codebases for each platform (Android and iOS) using the platform-specific programming languages and tools (Java/Kotlin for Android and Swift/Objective-C for iOS). Cross-platform development, on the other hand, involves using a single codebase to build apps that can run on multiple platforms.

React Native is a cross-platform development framework that offers a number of advantages over traditional native development approaches. Firstly, React Native allows developers to write code once and deploy it to multiple platforms, which can save a significant amount of time and effort. This is achieved by using a common language (JavaScript) and a single codebase for both Android and iOS.

Secondly, React Native offers a faster development cycle compared to traditional native development. Changes to the codebase can be quickly tested and deployed using hot reloading, which can significantly reduce development time.

Finally, React Native offers a more streamlined development experience, with a simpler and more consistent API compared to the platform-specific APIs used in traditional native development. This can make it easier for developers to learn and use the framework and can result in faster development times and fewer errors.

However, there are some trade-offs to using React Native instead of traditional native development approaches. While React Native can offer faster development times, it may not be as performant as traditional native development for certain types of complex applications or animations. Additionally, React Native may not provide access to all of the platform-specific features and APIs available in traditional native development.(hoffnmazor, 2023)

Table 6 Developer Use Case

Requirement	Native	React Native
Do you need to make an iOS-only or Android-only app?	x	
Do you need to make an overly complex app which utilises a large portion of platform-specific code?	x	
Do you plan to maintain the app over an extended period, without fear of Facebook quitting react native?	x	
Does your app need to support new mobile OS features as soon as they are released?	x	
Do you have a small team with limited time and resources, and need to make an app for both platforms?		x
Do you want to take advantage of fast build time and features such as hot reloading and love reloading?		x
Do your developers have strong React/Web development background?		x

Is your app going to look and behave the same on both platforms?		x

(Bansal, 2021)

In conclusion, React Native offers several advantages over traditional native development approaches, including faster development times and a more streamlined development experience. However, there are trade-offs to using React Native, and developers should carefully consider their project requirements and choose the approach that best meets their needs.

3 Native API

Native APIs are crucial for mobile app development because they allow developers to create high-performance applications that take full advantage of the capabilities of the underlying hardware.

This chapter would cover what Native APIs are and how crucial they in building a mobile application. By the end of this chapter, the reader should be able to know commonly used APIs in mobile applications.

3.1 Native APIs

Native APIs are software interfaces that allow developers to access the underlying hardware of a mobile device. They provide a way for apps to interact directly with the device's camera, microphone, GPS, and other features, enabling developers to create feature-rich, high-performance mobile applications. Native APIs are typically written in a low-level language such as C or C++, which allows them to interact more closely with the device hardware. By using Native APIs, developers can create apps that are faster, more responsive, and more reliable than those that rely solely on web-based technologies.

3.2 Importance of Native APIs in Mobile App Development

Native APIs are crucial for mobile app development for several reasons. Primarily, they provide a way for developers to access device-specific features that would otherwise be unavailable. This allows developers to create apps that are tailored to the capabilities of a specific device, delivering a better user experience.

In addition, Native APIs offer several other benefits that make them an essential component of mobile app development. These benefits include:

Table 7 Importance of Native APIs in mobile development

Better performance	Native APIs allow apps to interact more closely with the device hardware, resulting in faster and more responsive apps.
Improved reliability	By using Native APIs, developers can create apps that are more stable and less prone to crashes or other errors.
More seamless user experience	Native APIs allow apps to integrate more closely with the device hardware, resulting in a more seamless and immersive user experience.

(agency, n.d.)

3.3 Native APIs in React Native

React Native is a popular cross-platform mobile app development framework that allows developers to create apps for both iOS and Android platforms using a single codebase. (agency, n.d.)While React Native provides a range of tools and libraries for building mobile apps, Native APIs play a critical role in enabling developers to access device-specific features and create high-performance apps.

How native APIs are used in React Native

React Native uses Native APIs to provide a bridge between the app code and the underlying device hardware. This allows developers to access device-specific features such as the camera, GPS, accelerometer, and more, using the same JavaScript codebase for both iOS and Android platforms. (*Core Components and Native Components · React Native*, n.d.)

One of the key benefits of using Native APIs in React Native is that they allow developers to create high-performance apps that are faster and more responsive than those built using web-based technologies. Native APIs also provide a more seamless user experience by allowing apps to integrate more closely with the device hardware.

Commonly used Native APIs in React Native

The importance of frequently used APIs in the creation of native applications must be acknowledged before moving on to the comparison table. The benefit of native applications is that they have direct access to the frameworks and APIs made available by operating systems like iOS and Android. These APIs provide a wide range of capabilities, including using cutting-edge technologies like augmented reality or machine learning as well as handling network connectivity, accessing device hardware, and interacting with system services. By utilizing these APIs, developers can build incredibly specialized and feature-rich experiences that precisely match the capabilities of each platform. However, to access the full range of native APIs, cross-platform frameworks like React Native frequently need extra steps.

Table 8 Commonly Used APIs in React Native

CameraRoll API	Allows developers to access the device's camera roll and select photos to use within the app. (*@react-Native-Camera-Roll/Camera-Roll*, 2019/2023)
Geolocation API	Provides access to the device's GPS sensor, allowing developers to track the user's location. (🚧 *Geolocation · React Native*, 2022)
PushNotificationIOS API	Allows developers to send push notifications to iOS devices. (*@react-Native-Community/Push-Notification-Ios*, 2019/2023)

PermissionsAndroid API	Provides a way for developers to request permission to use certain device features, such as the camera or microphone. (*PermissionsAndroid · React Native*, 2023)
AsyncStorage API	Provides a simple key-value storage system that allows apps to store and retrieve data locally. (*AsyncStorage · React Native*, 2023)

Best Practices for Working with Native APIs in React Native

Working with Native APIs in React Native can be challenging, especially for developers who are new to the framework. To ensure that an app performs well and delivers a great user experience, it is important to follow some best practices when working with Native APIs. Some best practices to keep in mind include:

Table 9 Best practices to consider when working with react native using Native APIs.

Use the appropriate Native API for each feature	React Native provides a range of Native APIs for accessing device-specific features, so it is important to choose the one that best suits your needs.
Optimize for performance	When using Native APIs, it is important to optimize your app's performance to ensure that it runs smoothly on all devices. (Wagner & AppsFlyer, 2022)

Test on multiple devices	Because Native APIs can vary from device to device, it is important to test your app on a range of devices to ensure that it works as expected.

4 Java

This chapter will cover the basics of Java and its role in mobile app development, as well as how it can be used to enhance React Native applications. It will start by providing an overview of Java and its key features, such as High-Performance Computing, Java libraries and advances native functionality.

4.1 What is Java?

"Java is a general-purpose, class-based, object-oriented programming language designed for having lesser implementation dependencies. It is a computing platform for application development. Java is fast, secure, and reliable, therefore. It is widely used for developing Java applications in laptops, data centres, game consoles, scientific supercomputers, cell phones, etc." (Hartman, 2020)

4.2 Using Java to Enhance React Native Functionality

Java is a popular programming language that can be used to develop native Android apps, which can offer more advanced functionality and performance than cross-platform solutions like React Native. Here are some insights on how one could use Java to work on the features mentioned:

High-performance computing: Java is a strongly typed language that compiles to machine code, which means that it can offer high performance for computationally intensive tasks. Android's SDK includes several powerful tools for animation and graphics, which can be used to create complex animations and visual effects. Additionally, Java has a rich set of libraries and frameworks that can be used for data processing, such as Apache Spark, which is designed for big data processing. ('Performance Optimizations for React Native Applications,' 2020)

Advanced native functionality: Java can be used to access lower-level operating system functionality that may not be available through a cross-platform framework like React Native. For example, Java's Native Development Kit (NDK) can be used to write C or C++ code that can be compiled into a native library and called from Java code. This can be used to interface with hardware devices, or access lower-level operating system functionality that may not be available through the Java SDK. (Śpiewak, 2022)

User experience: Java offers more control over the user interface than React Native, which can be important for creating a highly customized user experience. Java's Android SDK includes a rich set of UI components, as well as the ability to create custom views and layouts. Additionally, Java offers greater control over animations and transitions, which can be used to create a more polished and engaging user experience.(Śpiewak, 2022)

Java Libraries: Java libraries provide several advantages in building Android apps compared to React Native libraries.

Firstly, Java has a much larger and more mature library ecosystem than React Native, with a wide range of libraries available to solve specific problems or provide specific functionality. This gives developers more flexibility and options when building Android apps.

Secondly, because Java has been used in Android development for much longer than React Native, many of the libraries available for Java have been extensively tested and refined over time, making them more reliable and stable than their React Native counterparts.

Thirdly, Java libraries can take advantage of Android's native APIs, which provides better performance and access to device-specific features. This is particularly important for apps that require access to hardware components or native functionality.

Overall, the extensive library ecosystem, reliability and stability, and access to native APIs make Java libraries advantageous in building Android apps compared to React Native libraries.

4.3 Additional Important Notes

JNI (Java Native Interface)

JNI is the Java Native Interface. It defines a way for the bytecode that Android compiles from managed code (written in the Java or Kotlin programming languages) to interact with native code (written in C/C++). JNI is vendor-neutral, has support for loading code from dynamic shared libraries, and while cumbersome at times is reasonably efficient. (*JNI Tips | Android NDK*, n.d.)

For example, a React Native application may require access to the device's camera, but the existing React Native camera API does not provide the necessary functionality. In this case, developers can use JNI to write custom Java code that interacts with the device's camera and then call this code from their React Native application. This allows them to take advantage of the full range of features provided by the device's camera, rather than being limited by the React Native camera API.

In summary, JNI allows developers to integrate native functionality into their React Native projects and can provide a solution when certain features are not available through React Native's built-in APIs.

5 Java vs React Native: A Comparison of Features

This chapter will delve deeper to compare the performance, usability, and efficiency of Java and React Native in developing specific features. The following features will be analysed:

Custom Camera Views

When it comes to developing mobile applications, both Java and React Native offer unique features and benefits. Java is a popular programming language used for Android development, while React Native is a framework that allows developers to write cross-platform applications using JavaScript.

One feature that developers commonly implement in mobile applications is the ability to access the device's camera. In Java, this can be done using the Camera API, which provides a wide range of features and customization options. Developers can create custom views and controls for the camera interface, adjust the camera settings, and process the captured images and videos.

In React Native, the camera functionality is provided through the React Native Camera component. While this component offers basic camera features such as taking pictures and recording videos, it may not provide the same level of customization options that are available in Java. However, React Native offers other benefits such as the ability to write cross-platform code and a faster development cycle.

In summary, Java offers extensive customization options for the camera functionality in mobile applications, while React Native provides cross-platform compatibility and faster development time. It depends on the specific needs and requirements of the project when deciding between Java and React Native for camera functionality and other features.

Audio and Video Processing

Audio processing is an essential aspect of many mobile applications, particularly those focused on music and audio recording. While React Native provides some basic audio processing capabilities, Java offers more advanced features that can enhance the overall audio experience of a mobile application.

Java provides access to various audio libraries and frameworks, such as the Java Sound API and the Java Media Framework, which can be used to perform real-time audio processing. Developers can use these libraries to implement features such as audio filtering, noise reduction, and audio effects.

One way to use Java for audio processing in React Native is through the Java Native Interface (JNI). The JNI is a programming framework that enables Java code to interact with native code, such as C or C++. By utilizing JNI, developers can access native audio processing libraries and integrate them into their React Native application.

Furthermore, the use of Java for audio processing in React Native can improve performance, as Java is a compiled language that runs natively on the device. This provides faster processing times than interpreted languages like JavaScript.

3D Graphics

3D processing is an important aspect of many mobile applications, particularly those in gaming and virtual reality. React Native provides some basic 3D functionality out of the box, but it may not be sufficient for more advanced use cases.

One way to enhance 3D processing in React Native is to leverage Java libraries. Java has a rich ecosystem of 3D libraries, including jMonkeyEngine, LibGDX, and Java3D, among others. These libraries provide powerful features for working with 3D models, rendering, and physics simulation. By integrating these libraries into React Native projects, developers can unlock a wide range of possibilities for 3D applications.

One area where Java can be useful for 3D processing in React Native is in the development of custom views. React Native provides a flexible framework for building custom UI components, and Java libraries can be used to create 3D views that can be embedded into React Native applications. This approach allows developers to create highly immersive and interactive interfaces that leverage the power of 3D graphics.

Overall, by incorporating Java libraries for 3D processing, React Native developers can enhance the capabilities of their applications and create more engaging user experiences.

In conclusion, Java provides more advanced features and capabilities for developing specific features such as custom camera views, audio and video processing, and 3D graphics. While React Native is a popular cross-platform framework for mobile app development, it may not be the best option for certain features that require more advanced capabilities.

Performance:

Java is known for its high-performance capabilities, making it an asset when it comes to optimizing React Native applications. React Native, being a cross-platform framework, is often faced with performance issues due to the need to maintain the same codebase across multiple platforms. These issues can range from slow animations to choppy user interfaces, resulting in a suboptimal user experience. By using Java in conjunction with React Native, developers can mitigate these performance issues.

One way in which Java can help improve React Native performance is using native modules. Native modules are written in the platform's native language and are called from JavaScript using a bridge. By utilizing native modules written in Java, developers can take advantage of the platform's native performance capabilities and reduce the overhead introduced by the JavaScript bridge. This approach is particularly useful when dealing with computationally intensive tasks such as image processing or complex animations.

Another way in which Java can help improve React Native performance is using third-party libraries. Java has a vast ecosystem of libraries available for various tasks, including image processing, networking, and database access. By using these libraries in React Native applications, developers can leverage the high-performance capabilities of Java without having to write custom native modules.

In addition, Java's just-in-time (JIT) compilation can also help improve React Native performance. JIT compilation allows Java code to be compiled at runtime, which can result in faster execution times compared to ahead-of-time (AOT) compilation used in some other mobile app development frameworks. This can be especially beneficial in React Native applications that heavily rely on dynamic, JavaScript-based UI components.

Libraries:

While React Native has a significant number of open-source libraries, Java boasts a vast collection of libraries that cater to various industries and domains. Java libraries are extensively used by developers to enhance the functionality of applications, especially in terms of performance and user experience. The availability of powerful and reliable libraries in Java has made it a go-to choose for developers looking to improve their application's functionality without having to write code from scratch.

On the other hand, React Native libraries have a smaller user base compared to Java libraries. This is partly due to React Native being a newer technology in the app development space. Additionally, the focus of React Native is primarily on providing a cross-platform framework, and while there are libraries available for specific functionalities, they are not as extensive as the ones available in Java.

Moreover, Java libraries are also more mature, and have been tested and used extensively, making them more reliable compared to React Native libraries. The development of Java libraries has been going on for several decades, and many of these libraries have stood the test of time and proven their reliability over the years. In contrast, React Native libraries are still in their early stages and are continually being developed, making them more prone to errors and bugs.

Overall, while React Native does have a significant number of libraries, Java libraries offer better functionality and reliability due to their extensive development history and maturity. Therefore, using Java libraries in conjunction with React Native can help developers achieve more efficient and robust application development.

Memory Management:

Memory management is a critical aspect of any application development process. Java's built-in memory management system, which uses garbage collection, is one of the language's most significant advantages over other programming languages. Garbage collection automatically frees up memory that is no longer in use, reducing the risk of memory leaks and other performance issues.(You & Hu, 2021)

In contrast, React Native relies on the JavaScript engine's memory management system, which does not have the same level of control over memory allocation. As a result, React Native apps are more prone to memory leaks and other performance issues caused by memory management problems.

Java's memory management system offers several benefits that can enhance React Native application performance. Java's garbage collection system ensures that memory is efficiently allocated and deallocated, reducing the risk of memory leaks. Additionally, Java's memory management system can be configured to optimize performance for specific application requirements, ensuring that the app runs smoothly even under high loads.

In summary, Java's memory management system offers significant advantages over React Native's memory management system, making it a valuable tool for enhancing React Native app performance. By leveraging Java's memory management capabilities, developers can ensure that their React Native apps run smoothly and efficiently.

5.1 Incorporating Java into a React Native Project

Java files could be incorporated into react native projects using the following methods:

Native Code Injection: Native code injection is a way to bridge Java code with React Native JavaScript code. A create a Java class that extends the “ReactContextBaseJavaModule” class and implement methods that can be called from JavaScript is the popular method of making native code injections. These methods can be used in React Native components.

External Libraries: External Java libraries can be used in a React Native project. By adding the library to the project's "build.gradle" file and use it in the Java code and then creating a Native Module or Native Component to expose the library's functionality to the React Native components.

Third-party Modules: Third-party modules that provide Java functionality could be imported. There are many modules available on npm that provide Java code. These modules can be added to the project and use them in the React Native code.

Cross-platform Tools: Cross-platform tools like Flutter or Xamarin can be used to create an app in Java and then integrate it with a React Native app using Native Modules or Native Components.

Expo: If using Expo to develop a React Native app, the expo eject command can be used to eject an app and add Java code. This will create a native Android project that can be modified as needed.

6 Methodology

This chapter outlines the research design and methods used to analyse the capabilities and limitations of react native in building android mobile applications.

This study used a mixed-methods approach to investigate React Native's drawbacks as a framework for developing mobile apps and consider how using Java might help to get over such drawbacks. The following research questions served as a guide for the research design:

1. What are some of the limitations of React Native as a mobile app development framework?
2. How can Java be used to overcome these limitations?

6.1 Data Collection

To address the research questions, data was gathered through two primary methods: a comprehensive literature review and a case study on functionalities.

The literature review involved a thorough search of academic and industry sources to identify key limitations of React Native and potential applications of Java. This included peer-reviewed journals, conference proceedings, books, and online resources.

The case study conducted was to test out probable literature reviews on online resources and learning materials. The case study was primarily focused on how native language could be implemented in a react native project and examples to test out the commonly used method.

6.2 Results

The results of the study revealed several key limitations of React Native, including performance issues, limited customization options, and difficulty integrating with native code. The analysis also revealed several potential benefits of using Java in combination with React Native, including improved performance and increased customization options.

The experiences and challenges of experienced React Native developers were also analysed, revealing common themes such as difficulties with debugging, lack of access to native modules, and limited community support.

6.3 Limitation

This research project has some limitations that may have affected the validity and reliability of the results. One of the limitations was the scope of the research, which focused only on the limitations of React Native without considering other mobile development platforms or frameworks. Another limitation was the use of publicly available documentation and online resources, which may not provide a comprehensive overview of the limitations of React Native.

In addition, the research did not involve data collection from actual users, and there was no survey conducted to gather more in-depth insights. The sample size was limited, which may affect the generalizability of the findings. Finally, the research may have been influenced by personal biases or assumptions of the researcher.

Despite these limitations, this research provides valuable insights into the limitations of React Native and how Java can be used to mitigate these limitations. Future research can address these limitations by examining other mobile development platforms and frameworks, involving larger and more diverse samples, and gathering data from actual users.

7 Native Code Injection (Practical Examples)

As mentioned, native code injection are common ways to bridge java files to react native JavaScript code. This chapter will discuss the step-by-step processes to take when using a String Java module in React Native applications. It is important to note the versions of various software used in the making of these projects are:

1. Gradle: 7.5.1
2. Android studio
3. React Native: 0.71.1
4. React Native Camera: ^4.2.1

7.1 Important Subjects to Consider

Understanding the ReactContextBaseJavaModule Class:

The ReactContextBaseJavaModule class is the base class for all native modules in React Native that are written in Java. It provides a way to access the React Native JavaScript runtime from the native code and exposes methods that can be called from the JavaScript code.

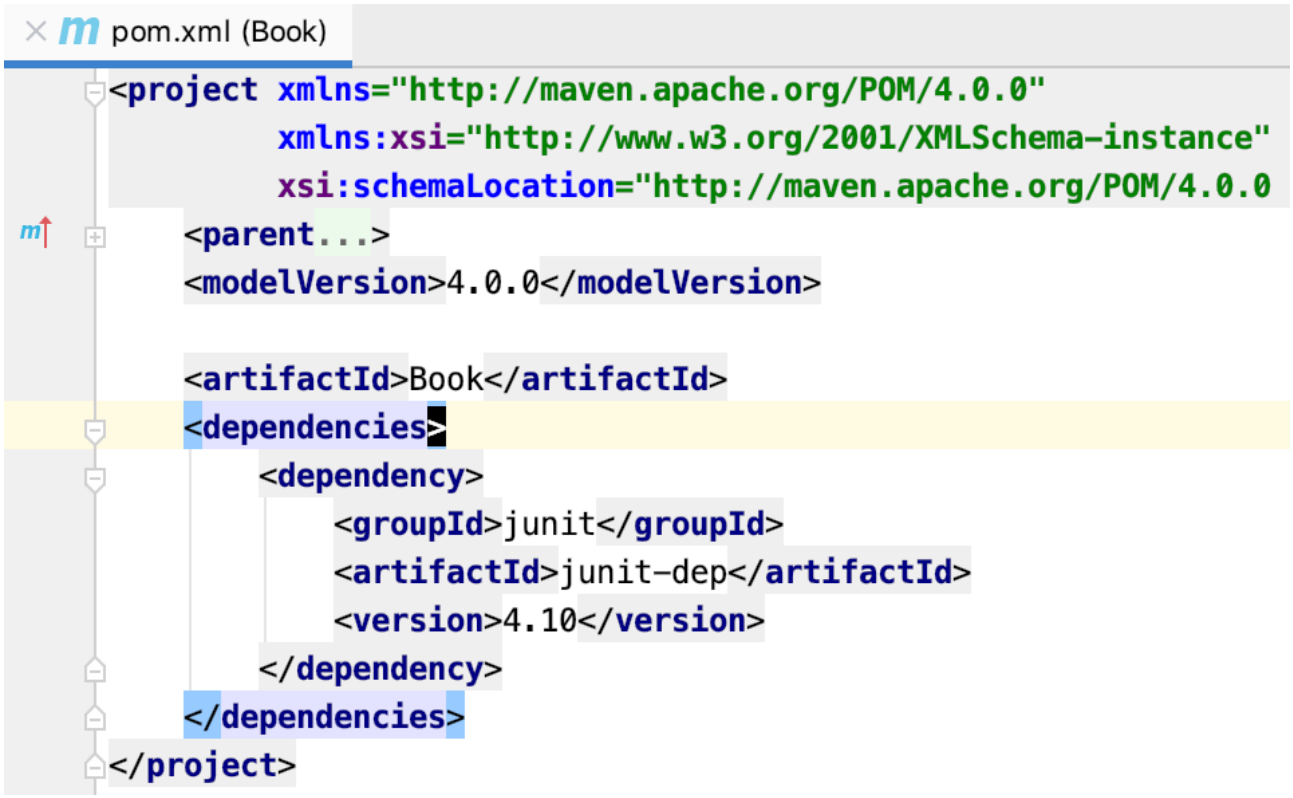
Maven Libraries.

When writing java applications, it is common to install dependencies using maven in a "pom.xml" file. To use such dependencies in a react native application with the research on this thesis, it has been discovered the dependencies could be acquired by writing in the "dependencies" block using the "build.gradle" file in the project using the format:

```
Implementation "<groupId>:<artifactId>:<version>"
```

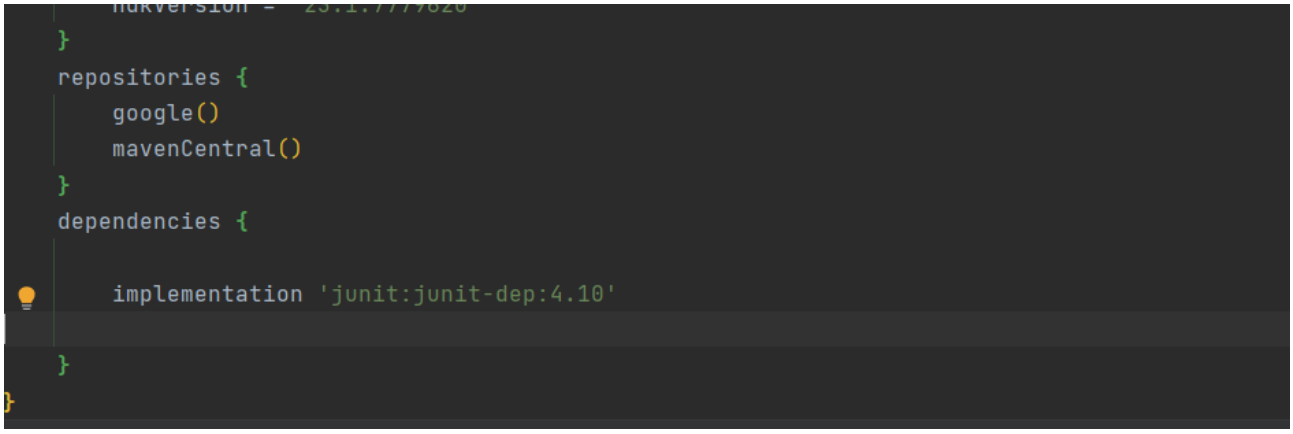
An example would be:

Figure 2 Maven dependency in pom.xml



(*Maven Dependencies | IntelliJ IDEA*, n.d.)

Figure 3 Android dependency in build.gradle



Methods and Annotations:

The ReactContextBaseJavaModule class provides several methods that can be overridden to implement the functionality of the native module. These methods include getName (), getConstants (), and @ReactMethod annotations.

Table 10 Important methods and annotations to be considered when making a module.

getName ()	This method returns the name of the native module as a string. The name is used by the JavaScript code to access the module.
getConstants ()	This method can be overridden to provide constants that can be used by the JavaScript code.
@ReactMethod	This annotation is used to mark a method as a JavaScript accessible method. Methods marked with this annotation can be called from the JavaScript code.

Lifecycle Methods:

The ReactContextBaseJavaModule class provides several lifecycle methods that can be overridden to handle the lifecycle of the native module. These methods include initialize (), onCatalystInstanceDestroy (), and onCatalystInstanceReady ().

Table 11 methods to be considered when rewriting the lifecycle methods.

initialize ()	This method is called when the native module is initialized. This method is used to perform any initialization tasks.
onCatalystInstanceDestroy ()	This method is called when the Catalyst instance is destroyed. This method is used to perform any clean-up tasks.
onCatalystInstanceReady ():	This method is called when the Catalyst instance is ready. This method is used to perform any tasks that require the Catalyst instance to be ready.

Understanding the ReactPackage Interface:

The ReactPackage interface is used to define a package that can register native modules and views with the React Native framework. The interface provides two methods, createNativeModules () and createViewManagers (), that must be implemented to register the native modules and views, respectively.

Implementing the ReactPackage Interface:

To use a String Java module in a React Native application, a package must be created that implements the ReactPackage interface. In the package, register the native module using the getNativeModules () method and any view managers using the getViewManagers () method.

7.2 Example 1 (Java Module with Strings)

Using a java module, this project would send strings from java to react native console. (Full Stack Niraj, 2022)

Create a Native Module:

To use a String Java module in React Native, a native module needs to be created. A native module is a bridge between the React Native JavaScript code and the native code written in Java or Objective-C. To create a native module in Java, follow these steps:

1. Create a new Java class that extends the ReactContextBaseJavaModule class.
2. Implement the getName () method to return the name of the module. This name will be used in the JavaScript code to access the module.
3. Implement the methods that needs to be exposed to the JavaScript code. For a String Java module, it would be advisable to implement a method that takes a string as input and returns a string as output.
4. Annotate the methods with @ReactMethod to make them accessible from JavaScript.
5. Override the getConstants () method if there need be to provide some constants that can be used in the JavaScript code.

Reference Appendix 1 for better understanding.

Create a Package:

Once the native module has been created, a package needs to be created to register the module with React Native. A package is a Java class that extends the ReactPackage class and is responsible for registering the native modules with React Native. To create a package, follow these steps:

1. Create a new Java class that extends the ReactPackage class.
2. Implement the createNativeModules () method to return a list of native modules that would need to be registered.
3. Implement the createViewManagers () method if there need be to register any custom view managers.
4. Override the getName () method to return the name of the package.

Note: Reference Appendix 2.

Register the Package:

To use the native module in a React Native application, the package needs to be registered. To register the package, follow these steps:

1. Open the React Native application's MainApplication.java file.
2. Import the package that was created in the previous step.
3. Add the package to the list of packages in the getPackages () method.
4. Save the file and rebuild the application.

Run the following commands in the main terminal:

Reference Appendix 3 and 4.

Use the Module:

Once the package has been registered, use the native module into the React Native application. To use the module, follow these steps:

1. Import the native module into the JavaScript code using the NativeModules module.
2. Call the method that needs to be used from the module.
3. Use the result returned by the method.

Reference Appendix 5

7.3 Example 2 (Camera Module with External Libraries)

Using the android java package for face detection and checking the captured image for a blur using OpenCV. The purpose of this example is to give a better understanding towards image processing.

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Primary Installations

Before starting this project, it is important to get some npm installations to avoid further problems:

```
npm install --save react-native-camera.
```

```
npm install --save react-native-svg.
```

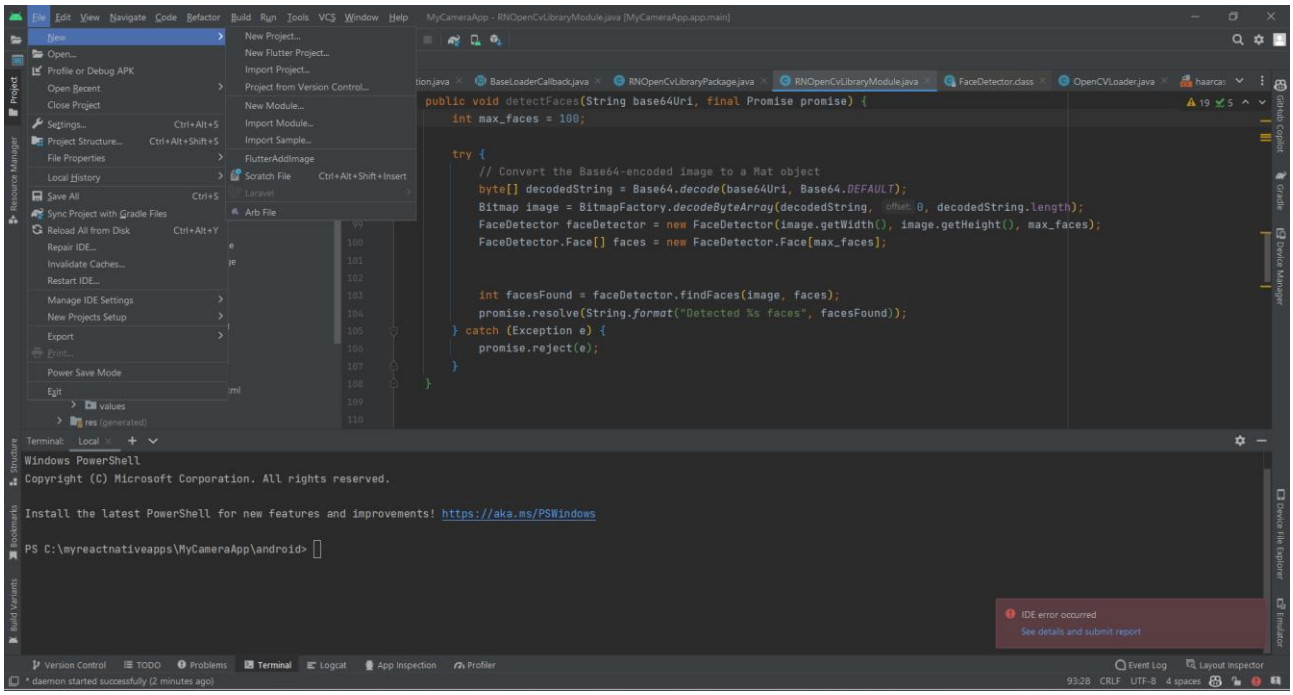
Download the Android SDK

The version of the sdk used in this project is version 4.7.0. The link is available here. (https://github.com/opencv/opencv/releases/download/4.7.0/opencv-4.7.0-android-sdk.zip)

Importing the Module

Accessing the project using android studio would be beneficial as it would highlight the untraceable errors. Ensure the android studio opens from the android folder of the react native project. Import the OpenCV module by clicking “File > New > Import Module” in the tab options.

Figure 4 Importing the module.



In the input tab, put in the absolute path to the sdk folder from the unzipped OpenCV sdk download.

Currently while programming android applications, it is impossible to use the camera without asking for permissions. Implementing the permissions in the "AndroidManifest.xml" would be a fix to the issue. (Reference Appendix 6)

Update the "build.gradle" file in the android/app folder to be able to use the module. Add it into the dependencies block. (Reference Appendix 7). Currently the camera in react native is facing issues with OpenCV and would require an extra addition to the "build.gradle" file. Add this into the "defaultConfig" block. (Reference Appendix 8)

Update the sdk "build.gradle" file to match the projects variable. Consider the following variables:

1. buildToolsVersion
2. minSdkVersion
3. compileSdkVersion
4. targetSdkVersion

For example:

Figure 5 Project build.gradle

```
buildscript {
    ext {
        buildToolsVersion = "33.0.0"
        minSdkVersion = 21
        compileSdkVersion = 33
        targetSdkVersion = 33

        // We use NDK 23 which has both M1 support and is the side-by-side NDK version from AGP.
        ndkVersion = "23.1.7779620"
    }
    repositories {
        google()
        mavenCentral()
    }
```

Figure 6 SDK build.gradle

```
defaultConfig {
    buildToolsVersion = "33.0.0"
    minSdkVersion  21
    compileSdkVersion = 33
    targetSdkVersion 33

    versionCode openCVersionCode
    versionName openCVersionName

    externalNativeBuild { ExternalNativeBuildOptions it ->
        cmake {
            arguments "-DANDROID_STL=c++_shared"
            targets "opencv_jni_shared"
```

Creating the Module and Package

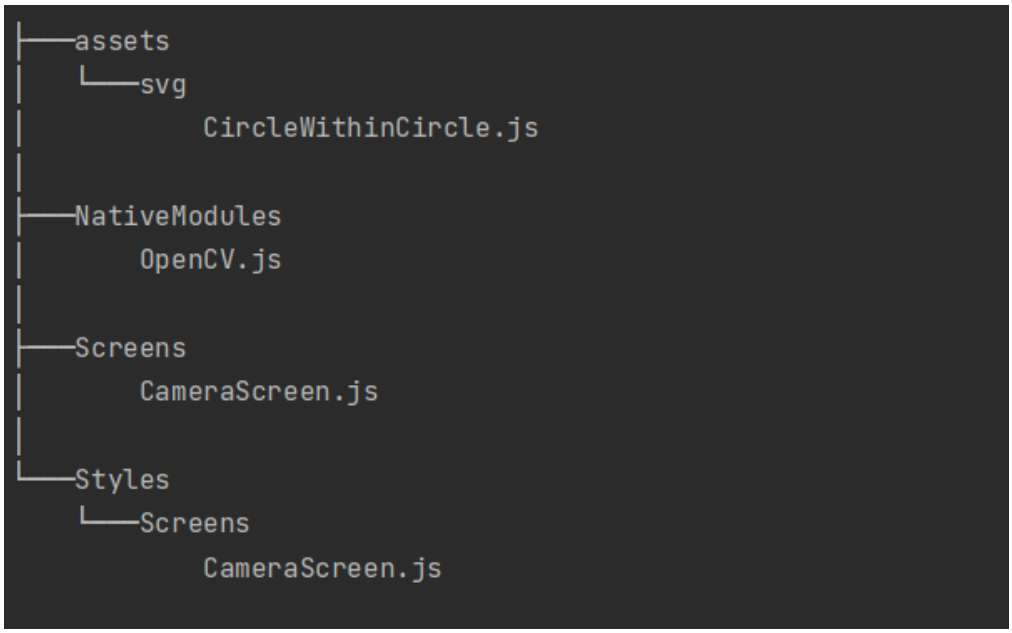
Create a new package called "`com.reactlibrary`" in the java directory. Create a java class called "RNOpenCvLibraryModule.java" (Reference Appendix 9). This class should create two methods. One method is meant to detect and return the amount of faces it has detected in an image using an inbuilt android package. The second method determines if the image is blurry to be considered as a good picture or not using OpenCV.

Create a java class called "RNOpenCvLibraryPackage.java." (Reference Appendix 10). Add the packages to the list of returned packages in the "MainApplication.java" file.

Implementing the Module

Create a folder called "src". In this folder, the directory should look like this:

Figure 7 Folder tree



Fill these Files as follows:

1. OpenCV.js (Reference Appendix 11).
2. Screens/CameraScreen.js (Reference Appendix 12).
3. Styles/Screens/CameraScreen.js (Reference Appendix 13).
4. CircleWithinCircle.js (Reference Appendix 14).

8 Reflection

The main objective of this project was to investigate the limitations of React Native as a mobile app development framework and explore how Java could be used to overcome these limitations. To achieve this objective, a mixed-methods approach was adopted, which involved conducting a comprehensive literature review to identify the key limitations of React Native and potential functionalities of Java. The results of this study contribute to the development of more efficient and effective mobile app development processes and highlight the potential benefits of using Java in combination with React Native.

During the research process, several challenges were met that affected the progress and quality of the study. One major challenge was the limited availability of practical APIs which were deprecated to work with or rather use. Many of which were advised against the research materials. an example of this would be the react native camera plugin. it caused a bit of struggles due to the difference in functionalities and versions.

Another challenge was the lack or unavailability of materials online to aid in this research. This caused a limitation on developments and recommendations that could have helped future developers and readers.

There are several topics for further investigation based on the limits and surprising results of this study. Future research can, for instance, examine how additional programming languages might be used in conjunction with React Native or investigate how the framework affects the user experience and app development workflows.

9 Conclusion

In conclusion, this study sheds light on the limitations of React Native and the potential for using Java to address these limitations. By addressing these limitations, developers can improve the performance, functionality, and user experience of mobile apps.

9.1 Summary of Study

In this research project, the limitations of React Native and the potential for using Java to mitigate these limitations have been explored. The research revealed several limitations of React Native, including performance issues, lack of third-party library support, and poor debugging capabilities. It was found that Java can be used to address some of these limitations by providing better memory management, advanced data structures, and more extensive debugging tools.

9.2 Implications for Developers and Organizations

The findings of this study have several practical implications for mobile app development. Firstly, developers who choose React Native as their mobile development platform should be aware of its limitations and consider using Java to address these limitations. Secondly, companies that use React Native for their mobile app development projects can benefit from incorporating Java expertise into their development teams.

9.3 Recommendations for Future Research

While this study sheds light on the limitations of React Native and the potential for using Java to mitigate these limitations, there is still much to explore in this area. As such, there are several recommendations for future research.

Firstly, future studies can investigate other programming languages and tools beyond Java that can be used to address the limitations of React Native. Additionally, research can examine the limitations of other mobile development platforms and frameworks to provide a more comprehensive understanding of the challenges faced by mobile app developers.

Secondly, future studies can involve data collection from actual users to obtain more in-depth insights into the limitations of React Native and the effectiveness of using Java to address these limitations. Such studies can also involve larger and more diverse samples to improve the generalizability of the findings.

Lastly, future studies can explore the effectiveness of incorporating Java expertise into mobile development teams and the potential benefits of doing so. Such studies can investigate how Java expertise can improve app performance, functionality, and user experience.

Overall, this research project highlights the importance of identifying and addressing the limitations of mobile development platforms. By doing so, developers can improve the performance, functionality, and user experience of mobile apps. Future research can expand upon these findings by exploring other mobile development platforms and frameworks and investigating the effectiveness of other programming languages and tools in addressing the limitations of these platforms.

References

AsyncStorage · React Native. (2023, January 12). https://reactnative.dev/docs/asyncstorage

Geolocation · React Native. (2022, December 15). https://reactnative.dev/docs/0.63/geolocation

A, V. (2022, December 9). *"React Memory Leaks: What, why, and how to clean them up!"* Medium. https://blog.devgenius.io/react-memory-leaks-what-why-and-how-to-clean-them-up-93606a07de84

agency, A. (n.d.). *10 Benefits of Native Mobile App Development*. Retrieved 26 March 2023, from https://arounda.agency/blog/10-benefits-of-native-mobile-app-development

Benefits of Using React Native for Mobile App Development. (n.d.). Retrieved 11 February 2023, from https://www.grazitti.com/blog/7-benefits-of-using-react-native-for-mobile-app-development/

Core Components and Native Components · React Native. (n.d.). Retrieved 26 March 2023, from https://reactnative.dev/docs/intro-react-native-components

Hartman, J. (2020, January 6). *What is Java? Definition, Meaning & Features of Java Platforms*. https://www.guru99.com/java-platform.html

JNI tips | Android NDK. (n.d.). Android Developers. Retrieved 20 March 2023, from https://developer.android.com/training/articles/perf-jni

Kundariya, H. (2020, December 2). *Analyzing The Various React Native Limitations*. ESparkBiz. https://www.esparkinfo.com/blog/react-native-limitations.html

Paterska, P. (n.d.). *What is React Native and When to Use It For Your App In 2023—Elpassion.com*. Retrieved 21 March 2023, from https://www.elpassion.com/blog/what-is-react-native-and-when-to-use-it

Performance Optimizations for React Native Applications. (2020, August 6). *Soshace*. https://soshace.com/performance-optimizations-for-react-native-applications/

PermissionsAndroid · React Native. (2023, March 17). https://reactnative.dev/docs/permissionsandroid

React Native vs Native: Which to choose for App Devs in 2023. (n.d.). Scalable Path. Retrieved 21 March 2023, from https://www.scalablepath.com/react-native/react-native-vs-native

@react-native-camera-roll/camera-roll. (2023). [Java]. react-native-cameraroll. https://github.com/react-native-cameraroll/react-native-cameraroll (Original work published 2019)

@react-native-community/push-notification-ios. (2023). [Objective-C]. react-native-push-notification. https://github.com/react-native-push-notification/ios (Original work published 2019)

Śpiewak, M. (2022, August 23). *Native vs Cross-Platform App Development: What is the right choice*. CrustLab. https://crustlab.com/blog/native-vs-cross-platform-app-development/

tagline. (2023, March 3). *8 Tips For Optimizing React Native Performance—Tagline Infotech*. https://taglineinfotech.com/react-native-performance/

Wagner, M., & AppsFlyer. (2022, February 8). Improving app performance (and why it's so important). *AppsFlyer*. https://www.appsflyer.com/blog/tips-strategy/app-performance/

Appendix 1: Native Module Class

```
1. //Module
2. package com.mycameraapp;
3.
4. import androidx.annotation.NonNull;
5. import androidx.annotation.Nullable;
6.
7. import com.facebook.react.bridge.Callback;
8. import com.facebook.react.bridge.ReactApplicationContext;
9. import com.facebook.react.bridge.ReactContextBaseJavaModule;
10.import com.facebook.react.bridge.ReactMethod;
11.
12.public class HelloWorldModule extends ReactContextBaseJavaModule {
13.    public HelloWorldModule(@Nullable ReactApplicationContext reactContext)
   {
14.        super(reactContext);
15.    }
16.
17.    @NonNull
18.    @Override
19.    public String getName() {
20.        return "Module";  //Name of the module to be used in Javascript file
21.
22.    }
23.
24.    @ReactMethod
25.    public void sayName(String name, Callback callback) {
26.        try {
27.            String message = "This is your app, " + name;
28.            callback.invoke(null, message);
29.        } catch (Exception e) {
30.            callback.invoke(e, null);
31.        }
32.    }
33.}
```

(Full Stack Niraj, 2022)

Appendix 2: Native Module Package

```
//package

package com.mycameraapp;


import androidx.annotation.NonNull;


import com.facebook.react.ReactPackage;

import com.facebook.react.bridge.NativeModule;

import com.facebook.react.bridge.ReactApplicationContext;

import com.facebook.react.uimanager.ViewManager;


import java.util.ArrayList;

import java.util.Collections;

import java.util.List;


public class HelloWorldPackage implements ReactPackage {

    @NonNull

    @Override

    public List<ViewManager> createViewManagers(@NonNull ReactApplicationContext
reactContext) {

        return Collections.emptyList();

    }


    @NonNull

    @Override

    public List<NativeModule> createNativeModules(

            @NonNull ReactApplicationContext reactContext) {
```

```
        List<NativeModule> modules = new ArrayList<>();


        modules.add(new HelloWorldModule(reactContext));


        return modules;
    }
}
```

(Full Stack Niraj, 2022)

Appendix 3: Register the package.

```
1. //MainApplication.java@getPackages
2. packages.add(new HelloWorldPackage());
```

(Full Stack Niraj, 2022)

Appendix 4: Rebuild the application.

```
4. npm start
5. a
```

(Full Stack Niraj, 2022)

Appendix 5: Implement the module.

```
/**
 * Sample React Native App * https://github.com/facebook/react-native * * @format
*/
 import React, {useRef} from 'react';
 import type {PropsWithChildren} from 'react';
 import {
     SafeAreaView,
     ScrollView,
     StatusBar,
     StyleSheet,
     Text,
     Button,
     useColorScheme,
     NativeModules,
     TouchableOpacity,
     View, requireNativeComponent,
 } from 'react-native';


 const {StringModule} = NativeModules


 function App() {
     const handleOnPress = () => {
         StringModule.sayName("Nathan", (err: any, message: any) => {
             if (err) return console.log(err);
             console.log('message', message)
         })
     }
     return (
     <SafeAreaView style={styles.container}><Text onPress={handleOnPress}
                                                  style={styles.text}>Hello

         World</Text>
         </SafeAreaView>

     );

 }

 const styles = StyleSheet.create({
     container: {
         flex: 1,
         justifyContent: 'center',
         alignItems: 'center',
     },
```

```
    text: {
        fontWeight: "bold",
        fontSize: 18,
        textAlign: "center"

    },
    preview: {
        flex: 1,
        justifyContent: 'flex-end',
        alignItems: 'center',
    },
    capture: {
        flex: 0,
        backgroundColor: '#fff',
        borderRadius: 5,
        padding: 15,
        paddingHorizontal: 20,
        alignSelf: 'center',
        margin: 20,
    }, body: {
        flex: 1,
    },
})

export default App;
```

(Full Stack Niraj, 2022)

Appendix 6: Manifest Permissions

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 7: Build.gradle dependency

```
implementation project(path: ':sdk')
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 8: Build.gradle defaultConfig

```
missingDimensionStrategy 'react-native-camera', 'general'
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 9: RNOpenCvLibraryModule.java

```
package com.reactlibrary;


import com.facebook.react.bridge.Promise;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
import com.facebook.react.bridge.Callback;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;


import org.opencv.android.OpenCVLoader;
import org.opencv.core.CvType;
import org.opencv.core.Mat;

import org.opencv.android.Utils;


import org.opencv.imgproc.Imgproc;


import android.media.FaceDetector;
import android.util.Base64;


import androidx.annotation.NonNull;


public class RNOpenCvLibraryModule extends ReactContextBaseJavaModule {


    public RNOpenCvLibraryModule(ReactApplicationContext reactContext) {
        super(reactContext);

        OpenCVLoader.initDebug();

    }

    @NonNull
    @Override
    public String getName() {
        return "RNOpenCvLibrary";
    }

    @ReactMethod
    public void detectFaces(String base64Uri, final Promise promise) {
        int max_faces = 100;

        try {
            // Convert the Base64-encoded image to a Mat object
            byte[] decodedString = Base64.decode(base64Uri, Base64.DEFAULT);
            Bitmap image = BitmapFactory.decodeByteArray(decodedString, 0,
decodedString.length);
            FaceDetector faceDetector = new FaceDetector(image.getWidth(),
image.getHeight(), max_faces);
            FaceDetector.Face[] faces = new FaceDetector.Face[max_faces];
```

```
            int facesFound = faceDetector.findFaces(image, faces);
            promise.resolve(String.format("Detected %s faces", facesFound));
        } catch (Exception e) {
            promise.reject(e);
        }
    }


    @ReactMethod
    public void checkForBlurryImage(String imageAsBase64, Callback
errorCallback, Callback successCallback) {
        try {
            BitmapFactory.Options options = new BitmapFactory.Options();
            options.inDither = true;
            options.inPreferredConfig = Bitmap.Config.ARGB_8888;

            byte[] decodedString = Base64.decode(imageAsBase64, Base64.DEFAULT);
            Bitmap image = BitmapFactory.decodeByteArray(decodedString, 0,
decodedString.length);


//      Bitmap image = decodeSampledBitmapFromFile(imageurl, 2000, 2000);
            int l = CvType.CV_8UC1; //8-bit grey scale image
            Mat matImage = new Mat();
            Utils.bitmapToMat(image, matImage);
            Mat matImageGrey = new Mat();
            Imgproc.cvtColor(matImage, matImageGrey, Imgproc.COLOR_BGR2GRAY);

            Bitmap destImage;
            destImage = Bitmap.createBitmap(image);
            Mat dst2 = new Mat();
            Utils.bitmapToMat(destImage, dst2);
            Mat laplacianImage = new Mat();
            dst2.convertTo(laplacianImage, l);
            Imgproc.Laplacian(matImageGrey, laplacianImage, CvType.CV_8U);
            Mat laplacianImage8bit = new Mat();
            laplacianImage.convertTo(laplacianImage8bit, l);

            Bitmap bmp = Bitmap.createBitmap(laplacianImage8bit.cols(),
laplacianImage8bit.rows(), Bitmap.Config.ARGB_8888);
            Utils.matToBitmap(laplacianImage8bit, bmp);
            int[] pixels = new int[bmp.getHeight() * bmp.getWidth()];
            bmp.getPixels(pixels, 0, bmp.getWidth(), 0, 0, bmp.getWidth(),
bmp.getHeight());
            int maxLap = -16777216; // 16m
            for (int pixel : pixels) {
                if (pixel > maxLap)
                    maxLap = pixel;
            }

//            int soglia = -6118750;
            int soglia = -8118750;
            if (maxLap <= soglia) {
                System.out.println("is blur image");
            }

            successCallback.invoke(maxLap <= soglia);
        } catch (Exception e) {
            errorCallback.invoke(e.getMessage());
        }
```

```
    }
}
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 10: RNOpenCvLibraryPackage.java

```java
package com.reactlibrary;

import androidx.annotation.NonNull;


import java.util.Collections;
import java.util.List;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;
import com.facebook.react.bridge.JavaScriptModule;
public class RNOpenCvLibraryPackage implements ReactPackage {
    @NonNull
    @Override
    public List<NativeModule> createNativeModules(@NonNull
ReactApplicationContext reactContext) {
        return Collections.<NativeModule>singletonList(new
RNOpenCvLibraryModule(reactContext));
    }

    // Deprecated from RN 0.47
    public List<Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }

    @NonNull
    @Override
    public List<ViewManager> createViewManagers(@NonNull ReactApplicationContext
reactContext) {
        return Collections.emptyList();
    }
}
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 11: OpenCV.js

```
import {NativeModules} from 'react-native';

export default NativeModules.RNOpenCvLibrary;
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 12: Screens/CameraScreen.js

```
import React, {Component} from 'react';
import {
    AppRegistry,
    View,
    Text,
    Platform,
    Image,
    TouchableOpacity, ToastAndroid,
} from 'react-native';
import {RNCamera as Camera} from 'react-native-camera';
import Toast from 'react-native-toast-message';

import styles from '../Styles/Screens/CameraScreen';
import OpenCV from '../NativeModules/OpenCV';
import CircleWithinCircle from '../assets/svg/CircleWithinCircle';

export default class CameraScreen extends Component {
    constructor(props) {
        super(props);

        this.takePicture = this.takePicture.bind(this);
        this.checkForBlurryImage = this.checkForBlurryImage.bind(this);
        this.detectFaces = this.detectFaces.bind(this);
        this.proceedWithCheckingBlurryImage =
this.proceedWithCheckingBlurryImage.bind(this);
        this.repeatPhoto = this.repeatPhoto.bind(this);
        this.usePhoto = this.usePhoto.bind(this);
    }

    state = {
        photoAsBase64: {
            content: '',
            isPhotoPreview: false,
            photoPath: '',
        },
    };

    checkForBlurryImage(imageAsBase64) {
        return new Promise((resolve, reject) => {
            if (Platform.OS === 'android') {
                OpenCV.checkForBlurryImage(imageAsBase64, error => {
                    // error handling
                    console.log(error);
                }, msg => {
                    resolve(msg);
                });
            } else {
                OpenCV.checkForBlurryImage(imageAsBase64, (error, dataArray) =>
{
                    resolve(dataArray[0]);
                });
            }
        });
    }

    detectFaces(imageAsBase64) {
        return new Promise((resolve, reject) => {
            if (Platform.OS === 'android') {
```

```
                OpenCV.detectFaces(imageAsBase64).then(result => {
                    // console.log('result', result);
                    resolve(result);
                })
                    .catch(error => {
                        console.error('Error: ' + error);
                    });
            } else {
                OpenCV.detectFaces(imageAsBase64, (error, dataArray) => {
                    console.log('dataArray', dataArray);
                    console.log('error', error);
                    resolve(dataArray[0]);
                });
            }
        });
    }


    proceedWithCheckingBlurryImage() {
        const {content, photoPath} = this.state.photoAsBase64;

        this.checkForBlurryImage(content).then(blurryPhoto => {
            if (blurryPhoto) {
                ToastAndroid.show('Photo is blurred!', ToastAndroid.SHORT);
                return this.repeatPhoto();
            }
            ToastAndroid.show('Photo is clear!', ToastAndroid.SHORT);
            this.setState({photoAsBase64: {...this.state.photoAsBase64,
isPhotoPreview: true, photoPath}});
        }).catch(err => {
            console.log('err', err);
        });
    }

    proceedWithFaceDetection() {
        const {content, photoPath} = this.state.photoAsBase64;

        this.detectFaces(content).then(blurryPhoto => {

            console.log('blurryPhoto', blurryPhoto.toString());
            ToastAndroid.show(blurryPhoto.toString(), ToastAndroid.SHORT);
            this.setState({photoAsBase64: {...this.state.photoAsBase64,
isPhotoPreview: true, photoPath}});

        }).catch(err => {
            console.log('err', err);
        });
    }

    async takePicture() {
        if (this.camera) {
            const options = {quality: 0.5, base64: true};
            const data = await this.camera.takePictureAsync(options);
            this.setState({
                ...this.state,
                photoAsBase64: {content: data.base64, isPhotoPreview: false,
photoPath: data.uri},
            });
            // this.proceedWithCheckingBlurryImage(); //Uncomment this to check
for blurry image
            this.proceedWithFaceDetection();
        }
```

```
    }


    repeatPhoto() {
        this.setState({
            ...this.state,
            photoAsBase64: {
                content: '',
                isPhotoPreview: false,
                photoPath: '',
            },
        });
    }

    usePhoto() {
        // do something, e.g. navigate
        // use this to manipulate the photo however you want
    }


    render() {
        if (this.state.photoAsBase64.isPhotoPreview) {
            return (
                <View style={styles.container}>
                    <Image
                        source={{uri:
`${this.state.photoAsBase64.content}`}}
                        style={styles.imagePreview}
                    />
                    <View style={styles.repeatPhotoContainer}>
                        <TouchableOpacity onPress={this.repeatPhoto}>
                            <Text style={styles.photoPreviewRepeatPhotoText}>
                                Repeat photo
                            </Text>
                        </TouchableOpacity>
                    </View>
                    <View style={styles.usePhotoContainer}>
                        <TouchableOpacity onPress={this.usePhoto}>
                            <Text style={styles.photoPreviewUsePhotoText}>
                                Use photo
                            </Text>
                        </TouchableOpacity>
                    </View>
                </View>
            );
        }

        return (
            <View style={styles.container}>
                <Camera
                    ref={cam => {
                        this.camera = cam;
                    }}
                    style={styles.preview}
                    androidCameraPermissionOptions={{
                        title: 'Permission to use camera',
                        message: 'We need your permission to use your camera',
                        buttonPositive: 'Ok',
                        buttonNegative: 'Cancel',
                    }}
                    androidRecordAudioPermissionOptions={{
                        title: 'Permission to use audio recording',
```

```
                        message: 'We need your permission to use your audio',
                        buttonPositive: 'Ok',
                        buttonNegative: 'Cancel',
                    }}
                    captureAudio={true}
                >
                    <View style={styles.takePictureContainer}>
                        <TouchableOpacity onPress={this.takePicture}>
                            <View>
                                <CircleWithinCircle/>
                            </View>
                        </TouchableOpacity>
                    </View>
                </Camera>
            </View>
        );
    }
}


AppRegistry.registerComponent('CameraScreen', () => CameraScreen);
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 13: Styles/Screens/CameraScreen.js

```
import {
    StyleSheet,
} from 'react-native';

export default StyleSheet.create({
    imagePreview: {
        position: 'absolute',
        top: 0,
        right: 0,
        left: 0,
        bottom: 60,
    },
    container: {
        flex: 1,
        flexDirection: 'row',
    },
    repeatPhotoContainer: {
        position: 'absolute',
        bottom: 0,
        left: 0,
        width: '50%',
        height: 120,
        backgroundColor: '#000',
        alignItems: 'flex-start',
        justifyContent: 'center',
    },
    topButtonsContainer: {
        flexDirection: 'row',
        alignItems: 'center',
        position: 'absolute',
        top: 0,
        left: 0,
        width: '100%',
        padding: 10,
        justifyContent: 'space-between',
    },
    focusFrameContainer: {
        position: 'absolute',
        top: 0,
        left: 0,
        height: '100%',
        width: '100%',
    },
    focusFrame: {
        height: 90,
        width: 90,
        borderWidth: 1,
        borderColor: '#fff',
        borderStyle: 'dotted',
        borderRadius: 5,
    },
    photoPreviewRepeatPhotoText: {
        color: '#abcfff',
        fontSize: 15,
        marginLeft: 10,
    },
    usePhotoContainer: {
        position: 'absolute',
```

```
        bottom: 0,
        right: 0,
        width: '50%',
        height: 120,
        backgroundColor: '#000',
        alignItems: 'flex-end',
        justifyContent: 'center',
    },
    photoPreviewUsePhotoText: {
        color: '#abcfff',
        fontSize: 15,
        marginRight: 10,
    },
    preview: {
        position: 'relative',
        flex: 1,
        justifyContent: 'flex-end',
        alignItems: 'center',
    },
    takePictureContainer: {
        position: 'absolute',
        paddingVertical: 20,
        bottom: 0,
        left: 0,
        right: 0,
        justifyContent: 'center',
        alignItems: 'center',
    },
});
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)

Appendix 14: CircleWithinCircle.js

```
import React from 'react';
import {
    Svg,
    Circle,
} from 'react-native-svg';

const CircleWithinCircle = () => (
    <Svg height="68" width="68">
        <Circle cx="34" cy="34" fill="#FFF" r="28" />
        <Circle cx="34" cy="34" fill="transparent" r="32" stroke="#fff"
strokeWidth="2" />
    </Svg>
);

export default CircleWithinCircle;
```

(*How to Use React Native & OpenCV for Image Processing*, n.d.)