**Elvis Gamua**

# MULTI-LANGUAGE APP DEVELOPMENT - JAVA AND C++

**Thesis**

**CENTRIA UNIVERSITY OF APPLIED SCIENCES**

**Information Technology**

**August 2020**

**ABSTRACT**

| Centria University of Applied Sciences | Date<br>August 2020 | Author<br>Elvis Gamua |
|---|---|---|
| **Degree programme**<br>Information Technology | | |
| **Name of thesis**<br>MULTI-LANGUAGE APP DEVELOPMENT - JAVA AND C++ | | |
| **Instructor**<br>Dr Szewczyk Grzegorz | | **Pages**<br>76+3 |
| **Supervisor**<br>Dr Szewczyk Grzegorz | | |

Smartphones have become the most ubiquitous of consumer-electronic appliances. This is indicative of the importance these gadgets play in society. From calling, playing games, consulting credit information from the bank, these computationally powerful gadgets are used in many spheres in life. There exists an ever-increasing variety of apps to aid users perform a wide range of tasks. As usage increases for these gadgets, concerns arise for the best computational and security options. One way of solving this dual problem is to make hybrid apps using 2 or more programming languages. Such apps will optimize and exploit each programming language.

The entirety of this thesis investigates the computational aspect of smart phones. It investigates how mobile apps can be made using two programming languages. An investigation is made on passing and retrieving different types of data structures from Java to C++ and from C++ back to Java. Threads are finally given a brief look towards the end of this thesis.

**Key words:**

Dalvik and ART, JNI, JVM, NDK, references, Structures, Threads

## ABBREVIATIONS

| | |
|---|---|
| ABI | Application Binary Interface |
| APK | Android Package |
| API | Application Programming Interface |
| ART | Android Runtime |
| CPU | Central Processing Unit |
| ELF | Executable and Linkable Format |
| IOS | Iphone Operating System |
| JNI | Java Native Interface |
| JVM | Java virtual machine |
| LLDB | Low-Level Debugger |
| NDK | Native Development Kit |
| OAT | Of Ahead Time |
| SDK | Software Development Kit |
| UI | User Interface |

CONTENTS

**FIGURES**

**TABLES**

**CODE SNIPPETS**

# 1 INTRODUCTION

The advent of smart phones introduced a new dawn to programming with the introduction of the Android operating system. This operating system has been managed for years now by Java even though JetBrains in 2011 unveiled the Kotlin programming language. New Java APIs have been gradually added to help manage the capabilities of this now ubiquitous system. Java and its APIs are great, however, Google developers have utilized  the benefits of C/C++, working in tandem with Java and Kotlin, to procure greater processing power in the creation of games and other services in an effort to render the Android Operating System much more rewarding, productive and most of all, provide an interactive machine-human environment with an exciting user-experience to their clients.

This project is divided into six core parts. In chapters 2 and 3, a brief introduction will be made of the native development kid (NDK) and the Java native interface (JNI). In these two chapters, a look is taken at the basic configuration procedures for the development platform. Brief introductions will be made of CMake which is the build system generator that Android uses for the native platform, the Java native interface (JNI) which is a protocol on how Java code should call and be called from natively written code, the Application Binary Interface (ABI) which is information for the CPU instruction set to be used and Dalvik and ART. A look will also be taken at the structure of native programs.

In chapters 4 and 5 an investigation will be made on how to pass/retrieve primitive types, array types, object types and array of object types to/from a native class. In chapter 6 an app will be made that launches a background thread to calculate prime numbers up to a certain value and the result of this operation received at the Java end. Chapter 7 will conclude the investigative chapters with an attempt at passing a C++ struct object to a Java object. The discussed code samples in this project are found in the writers Github account. A link to the pages containing the code samples can be found in the list of references of this project. On the landing page of the link, Project1 corresponds to the PrimitiveTypesApp app, Project2 to the arrayTypes app, Project3 to the ObjectTypes app, Project4 to the arrayOfObjects app and Project6 to the CopyingStructInfo app.

# 2 THE NATIVE DEVELOPMENT KIT (NDK)

It is the intent of this chapter to explore the essence of NDK, which entails the configuration of Android Studio to accommodate the NDK with the written Java program. This chapter illuminates the justification into the deployment of NDK, highlights how libraries are etched into the Android Studio system and ultimately accomplish the end product/system of choice. Moreover, an investigation will be launched into revealing the inner workings of the Java Native Interface (JNI), which is the protocol used in Java to establish a reversible/dynamic communication between C/C++ and Java. JNI spells out a specific procedure on how methods can be initialized from Java and detail how objects, methods, classes can be referenced and accessed from the native class, methods or libraries.

## 2.1 Difference between C++ and Java and meaning of "native"

One of the major differences apart from the language features of Java and C++ is that C++ compiles directly into the machine code thus the meaning of the word native. Native means that the source code has been compiled to the machine code which is native to the computer in question. The machine code is the format that permits the Central Processing Unit to execute instructions. Java on the other hand is both an interpreted and a semi compiled language. Java is semi compiled because it does not compile directly to machine code but instead compiles to an intermediary code called byte code. The Java source code is compiled to a Java bytecode. The bytecode then needs an interpreter to be able to convert it to machine readable code. (Liang 2015.)

Apart from the differences in language features between Java and C++, there are some notable differences in representation of primitive types, arrays, objects and management of memory that make it difficult for code in these two languages to be mixed. Java has a single way to represent and use any systems memory but C++ has different ways. Using C++ therefore, one type can have different sizes on different platforms. The representation of arrays in memory for Java is also different. Java arrays are objects unlike C++ arrays that are primitive types. This means access of these types are inherently different. There is incompatible memory management in the two languages. Memory management in Java is the work of the garbage collector which performs an automatic task, but memory management in C++ is a job that has to be handled by the programmer. (University Of Princeton 2019.)

## 2.2 Definition of the NDK

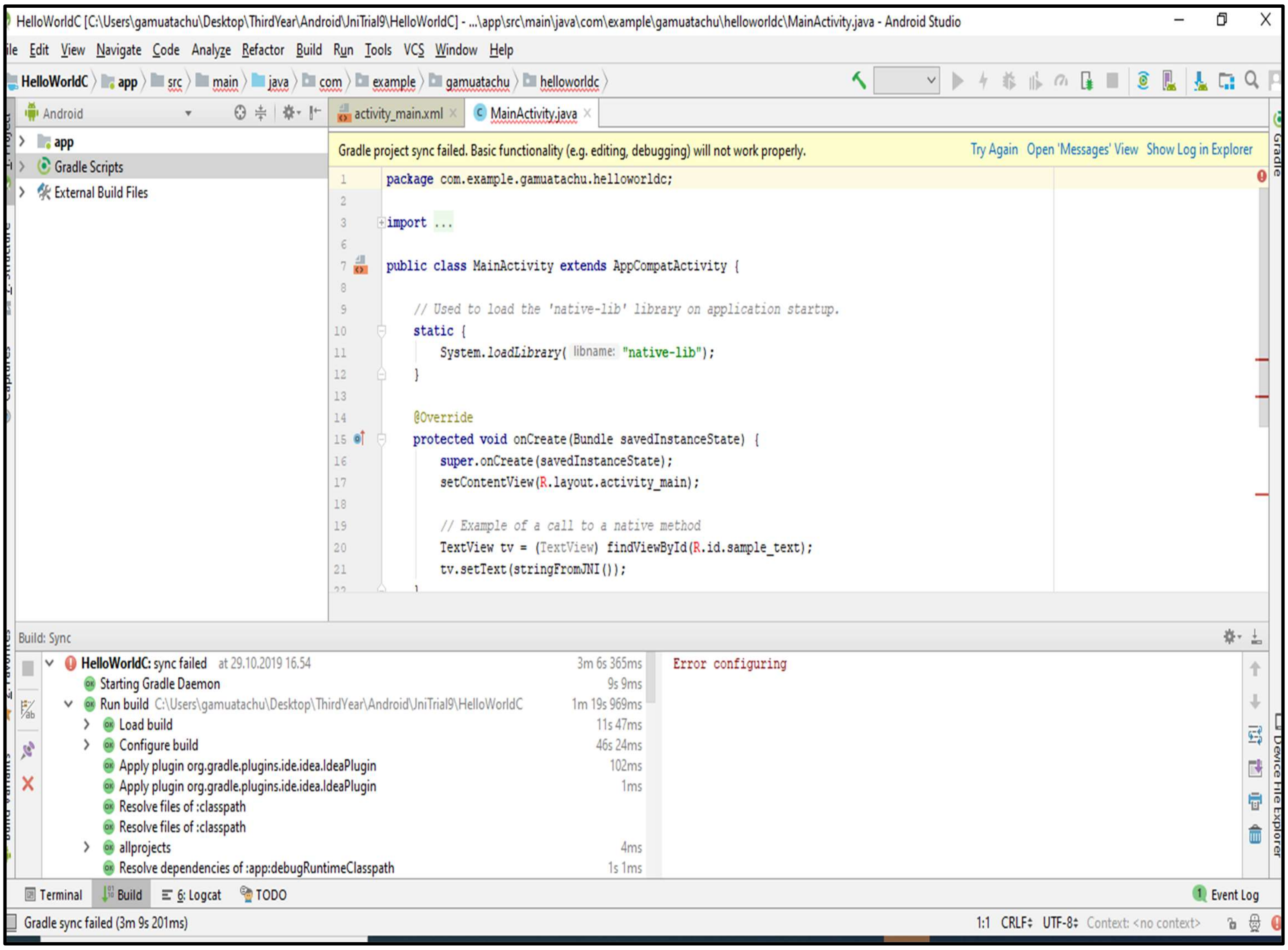
The native development kit (NDK) is a toolset that allows the use of C and C++ code with Android. It allows for the incorporation of this code into applications through the Java native interface (JNI). Many of the Android applications in the market use NDK and JNI. NDK can be useful in cases where there is need to achieve low latency or run computationally intensive applications, such as games or physics simulations. C++/C library reuse is another reason NDK could be useful. Starting from Android studio 2.2, NDK can be used to compile C/C++ code into a native library and then embedded into the APK file with Gradle. The application can then make runs to the native library when it must use the Java Native Interface (JNI). Now, to compile C/C++ code Android uses CMake. It also uses NDK-build. The functionality of CMake will be exploited in the course of this project. ( Android Developers Documentation 2019.)

## 2.3 Reasons for using NDK

There are many reasons why NDK could be used. Firstly, as mentioned in section 2.2, NDK could be used to run computationally intensive applications. Secondly, re-using existing C/C++ code in a new Android application could be yet another reason for using the NDK. Another reason again could be the need to develop an application that will run on other platforms like IOS and Windows. And finally, when need arises to use some processor features that are otherwise absent in the SDK or optimize critical code at assembly level. ( Android Developers Documentation 2019; Liang 1999.)

## 2.4 How to configure Android Studio

To be able to run native code, download of additional tools must be done. CMake, NDK and LLDB are tools that must be additionally downloaded. The SDK manager interface presents links for the download of these tools. Syncing problems may arise while configuring Android studio. The classpath dependency under build Gradle, is a likely scenario in this light. The classpath is an Android configuration property that indicates to Gradle where to find the dependency files for a project in order for the classes in a project to make use of them. While the initial program was under investigation it was discovered that Gradle always failed to sync the project as shown in Figure 1. It was discovered that a higher classpath was needed. The classpath was thus changed from classpath 'com.android.tools.build:gradle:3.1.3' to classpath 'com.android.tools.build:gradle:3.2.1' as shown below in figure 2. ( Android Developers Documentation 2019.)


HelloWorldC [C:\Users\gamuatachu\Desktop\ThirdYear\Android\UniTrial9\HelloWorldC] - ...\app\src\main\java\com\example\gamuatachu\helloworldc\MainActivity.java - Android Studio
ile Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
HelloWorldC app src main java com example gamuatachu helloworldc
Android
app
Gradle Scripts
External Build Files
activity_main.xml
MainActivity.java
Try Again
Open 'Messages' View
Show Log in Explorer
Build: Sync
Terminal
Build
6: Logcat
TODO
Event Log
1:1 CRLF UTF-8 Context: <no context>

Gradle project sync failed. Basic functionality (e.g. editing, debugging) will not work properly.

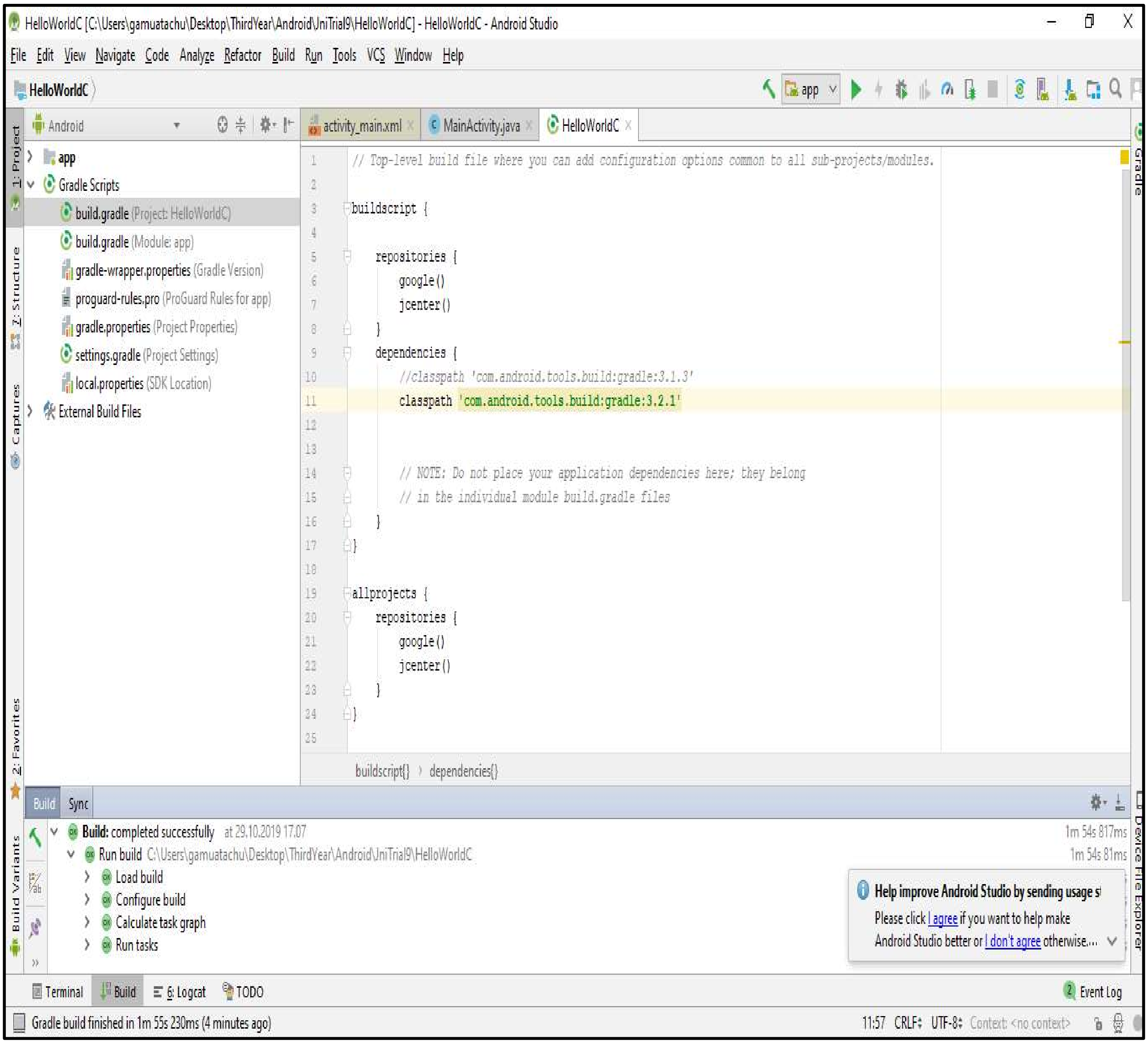
```
package com.example.gamuatachu.helloworldc;

import ...

public class MainActivity extends AppCompatActivity {

    // Used to load the 'native-lib' library on application startup.
    static {
        System.loadLibrary( libname: "native-lib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Example of a call to a native method
        TextView tv = (TextView) findViewById(R.id.sample_text);
        tv.setText(stringFromJNI());
```

```
HelloWorldC: sync failed  at 29.10.2019 16.54                          3m 6s 365ms
  Starting Gradle Daemon                                                   9s 9ms
  Run build C:\Users\gamuatachu\Desktop\ThirdYear\Android\UniTrial9\HelloWorldC   1m 19s 969ms
    Load build                                                            11s 47ms
    Configure build                                                       46s 24ms
    Apply plugin org.gradle.plugins.ide.idea.IdeaPlugin                      102ms
    Apply plugin org.gradle.plugins.ide.idea.IdeaPlugin                        1ms
    Resolve files of :classpath
    Resolve files of :classpath
    allprojects                                                                4ms
    Resolve dependencies of :app:debugRuntimeClasspath                      1s 1ms
```

```
Error configuring
```

Gradle sync failed (3m 9s 201ms)

FIGURE 1. Project fails to build with classpath lower than or equal to 3.1.3


HelloWorldC [C:\Users\gamuatachu\Desktop\ThirdYear\Android\UniTrial9\HelloWorldC] - HelloWorldC - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
HelloWorldC
Android
app
Gradle Scripts
build.gradle (Project: HelloWorldC)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
External Build Files
activity_main.xml
MainActivity.java
HelloWorldC
buildscript{} › dependencies{}
Build
Sync
Terminal
Build
6: Logcat
TODO
Event Log

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {

    repositories {
        google()
        jcenter()
    }
    dependencies {
        //classpath 'com.android.tools.build:gradle:3.1.3'
        classpath 'com.android.tools.build:gradle:3.2.1'


        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

```
Build: completed successfully at 29.10.2019 17.07                1m 54s 817ms
  Run build C:\Users\gamuatachu\Desktop\ThirdYear\Android\UniTrial9\HelloWorldC   1m 54s 81ms
    Load build
    Configure build
    Calculate task graph
    Run tasks
```

Help improve Android Studio by sending usage s
Please click I agree if you want to help make Android Studio better or I don't agree otherwise....

Gradle build finished in 1m 55s 230ms (4 minutes ago)    11:57 CRLF: UTF-8: Context: <no context>

FIGURE 2. Project builds when classpath is changed to a classpath higher than 3.1.3

## 2.5 CMake

CMake is the build system generator that Android uses. CMake compiles CMake scripts (from CMakeList.txt) and feeds the output to Ninja which is the compiler that the NDK uses. Files are fed into CMake and CMake generates new files that build systems like Ninja, MakeFiles and Xcode can use. Instead of integrating files directly into all these native build tools, files can be integrated once into CMake and CMake will generate output files that all these systems can understand. This is useful when cross platform development is needed or when a developer needs to change compilers in a project. The developer will simply make use of CMake and describe a path for his files instead of adding all these files into the various build tools he is using. In case there is need to use a different native compiler, CMake will generate code for the new compiler in question. (CMake 2019.)

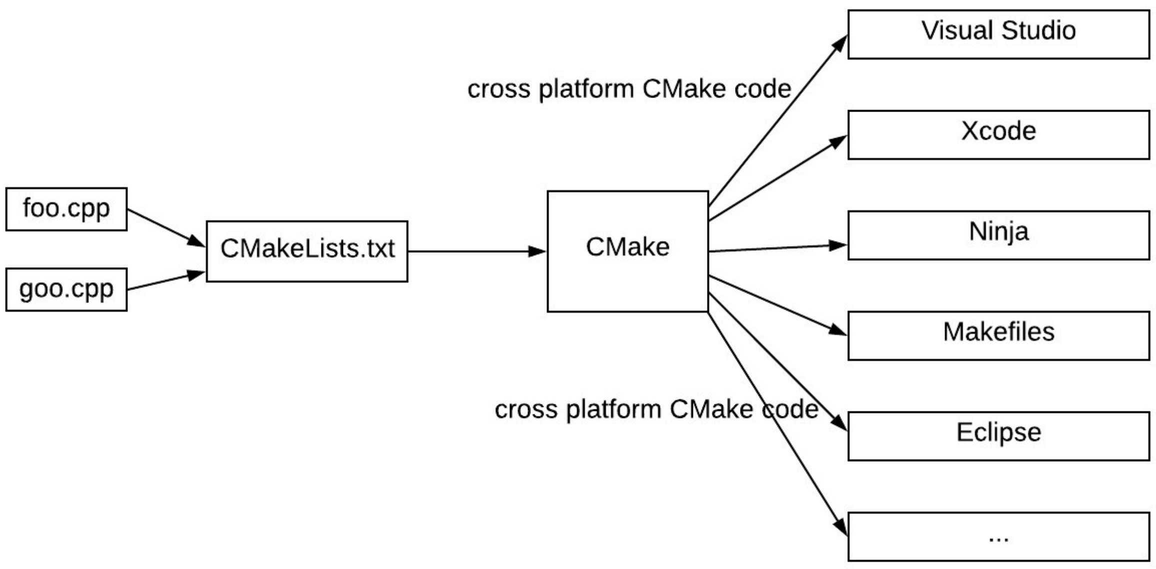
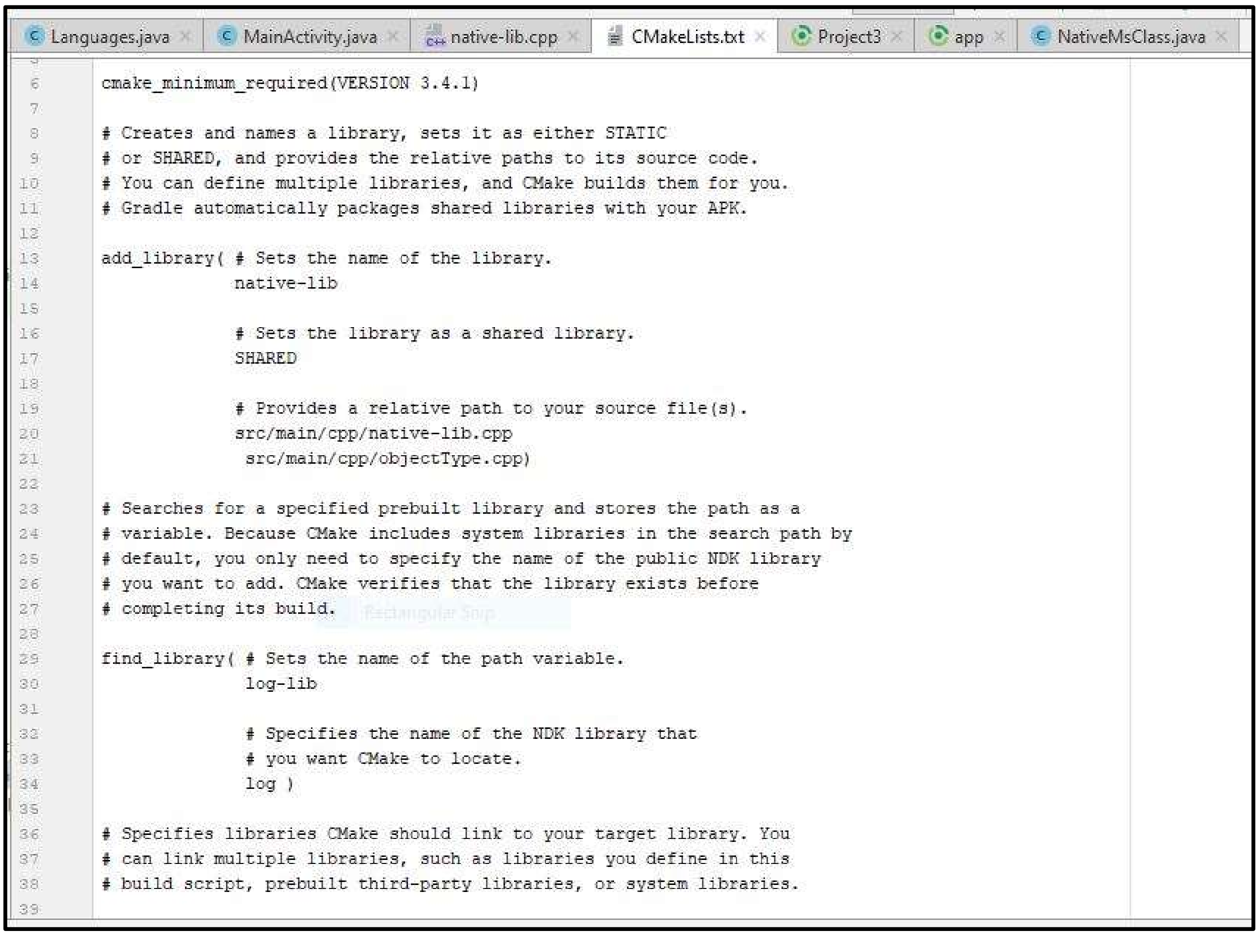


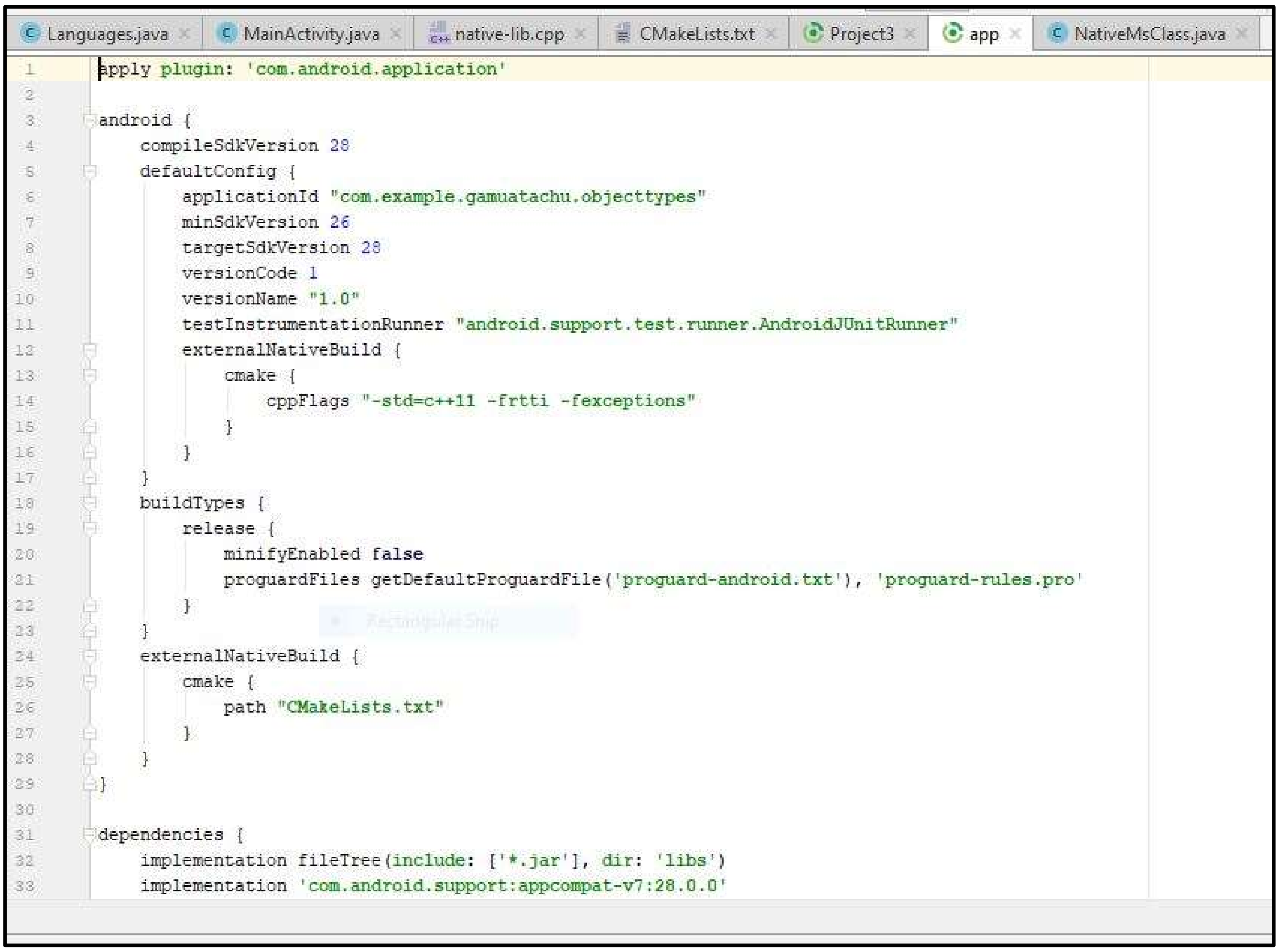
FIGURE 3. How CMake Works pictorially

CMake allows the description of a project in its CMakeList.txt as shown in figure 4. The CMakeList.txt file is found in the External Build Files in the Android project directory. There, the minimum version of CMake can be set, what libraries to add, what libraries to target and what libraires to find. The add_library() provides a relative path to the source files that are being used. The target_link_library() specifies a target library that should be linked with the log library of the NDK and finally the find_library() finds an NDK library and stores its location as a variable. The variable can be used later to refer to the library in the build scripts. After configuration of a new CMake build script, Gradle needs to be configured to include the CMake project as a build dependency, so that Gradle builds and packages the native library with the app's APK. The path to allow Gradle find the CMakeList.txt has to be specified in the build.gradle (Module: app) as shown in figure 5. ( Android Developers Documentation 2019; CMake 2019.)



```
cmake_minimum_required(VERSION 3.4.1)

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
             native-lib

             # Sets the library as a shared library.
             SHARED

             # Provides a relative path to your source file(s).
             src/main/cpp/native-lib.cpp
              src/main/cpp/objectType.cpp)

# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.

find_library( # Sets the name of the path variable.
              log-lib

              # Specifies the name of the NDK library that
              # you want CMake to locate.
              log )

# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.
```

FIGURE 4. Section of CMakeList.txt



```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.example.gamuatachu.objecttypes"
        minSdkVersion 26
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
                cppFlags "-std=c++11 -frtti -fexceptions"
            }
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    externalNativeBuild {
        cmake {
            path "CMakeLists.txt"
        }
    }
}

dependencies {
    implementation fileTree(include: ['*.jar'], dir: 'libs')
    implementation 'com.android.support:appcompat-v7:28.0.0'
```

FIGURE 5. Section of build.gradle (Module: app)

## 2.6 JNI

The Java Native Interface (JNI) is the interface or protocol that Java uses to communicate to C/C++ and C/C++ uses to communicate to Java. The portability of code across many platforms is ensured when JNI is used. JNI is a protocol that defines how methods can be called from Java and how from the native side class members, objects, methods, classes can be referenced and accessed. ART and Dalvik both have JNI in their systems. JNI defines the standards and procedures for calling code in C/C++ and sending the result back to Java. The JNI framework is programmatically very versatile. Information about Java classes is derived from the native side. These classes can be loaded in the native side. Strings, arrays and primitive types can be created and then JNI can use these to do its tasks. Objects and primitive types can be created, updated and passed to be accessed between native side and Java side. The native side can also call a method that is only defined as native in the Java side and it can also call a pure Java method. Parameters can be passed from Java to native method and back to Java. Exceptions are not left out as they can be thrown from the native side and handled from the Java side. (MIT Education 2019; Krajci & Cummings 2013.)

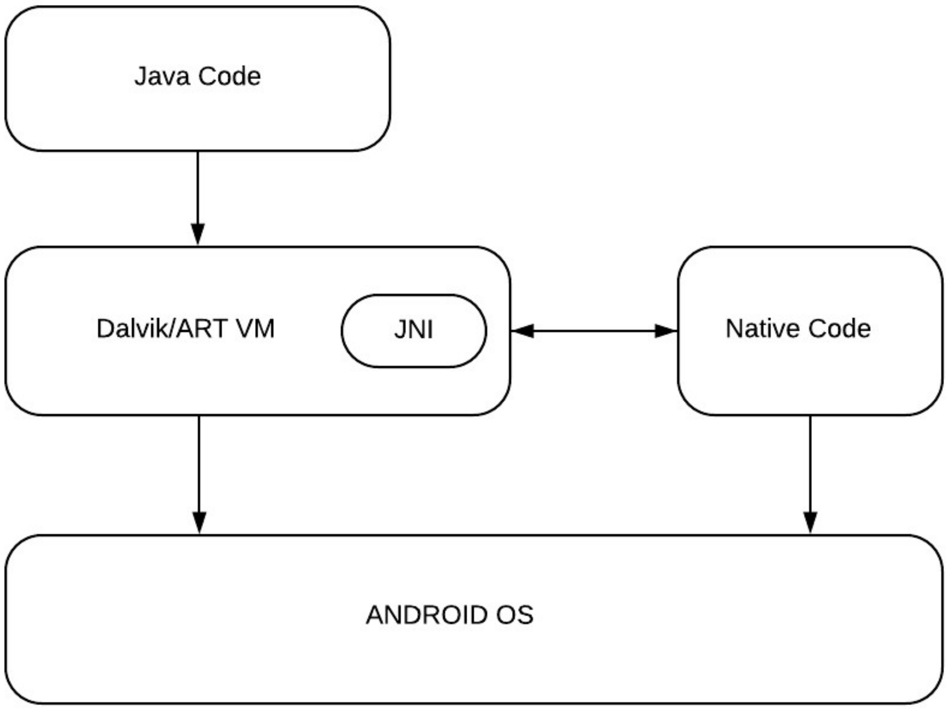


FIGURE 6. JNI work flow 1

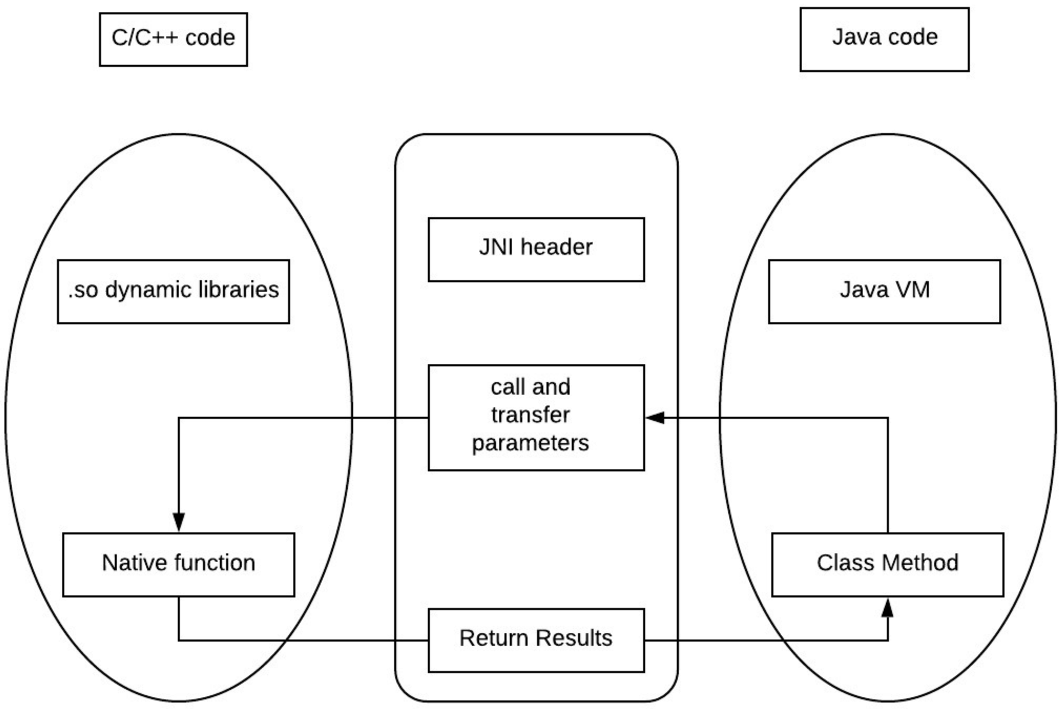
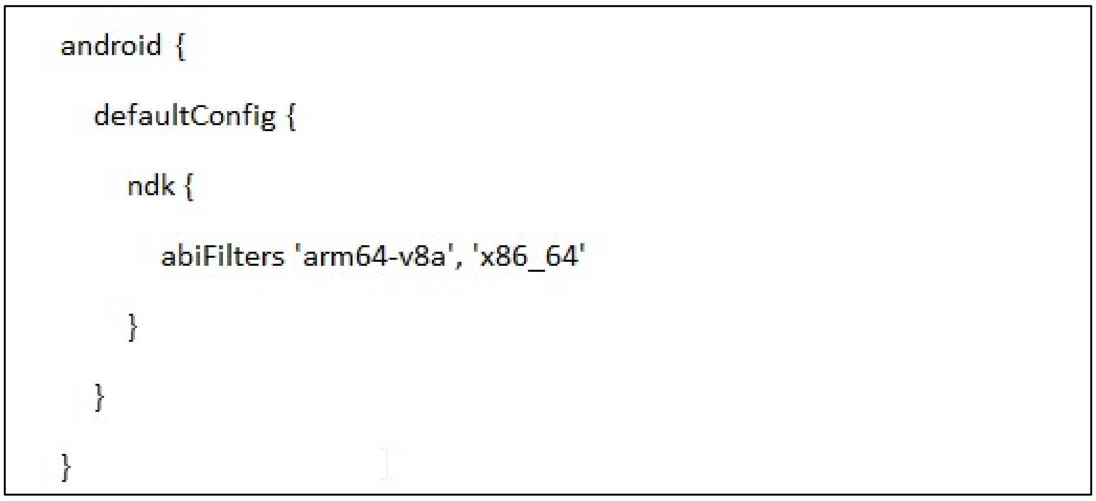


FIGURE 7. JNI Work Flow 2

## 2.7 ABIs

There are many different Android devices in the market and each of these devices have different CPUs and instructions sets. An Application Binary Interface (ABI) is information for the CPU instruction set to be used, the endianness of memory stores, convention for passing data between applications and system, format for executable binaries and name mangling conventions for C++. There are 4 supported ABIs in Android vis Armeabi-v7a, arm64-v8a, x86, x86_64. Generating code for a specific ABI can be done in Gradle as shown in code snippet 1. The code in code snippet 1 builds for all 64-bit ABIs. By default, Gradle builds for all non-deprecated ABIs.  The default build is to include the binaries for each ABI in a single APK. (Android Developers Documentation 2019.)



```
android {
   defaultConfig {
      ndk {
         abiFilters 'arm64-v8a', 'x86_64'
      }
   }
}
```

CODE SNIPPET 1. Generating Code for specific ABI

## 2.8 Dalvik and ART

When an application is made, the Java source codes are compiled to class files by the Java compiler. These class files are then further converted to a .dex file by a tool in the SDK called dx. The Android asset packaging tool then converts the resources, images, native codes and the .dex files to an .apk (Android package) file. This is the file that is distributed. From Android 5.0, ART was the sole and exclusive run time used, but to maintain backward compatibility, ART used the original .dex files as input. This was intended to render devices that continue to use the Dalvik Virtual machine workable. The dex2aot tool in ART converts the .dex files to oat files and saves them in an ELF file. OAT is ahead of time, which means all the files are converted and saved in the device and not compiled each time the application is to be run like the Dalvik machine does. ART uses a combination of ahead of time compilation and Just in Time compilation. (Android Developers Documentation 2019; Vogel & Scholz 2012.)

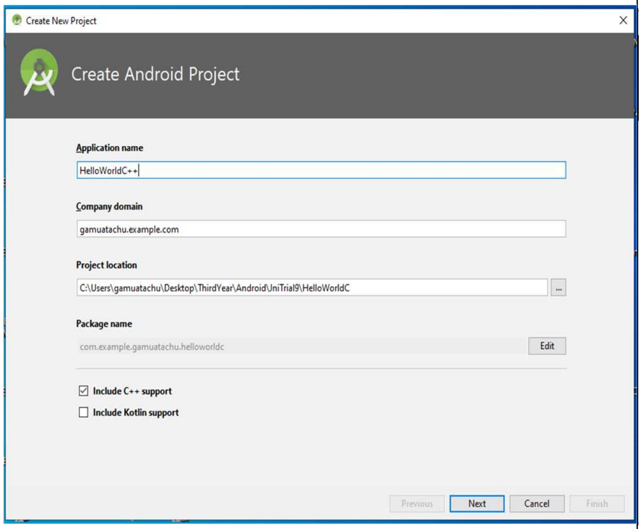
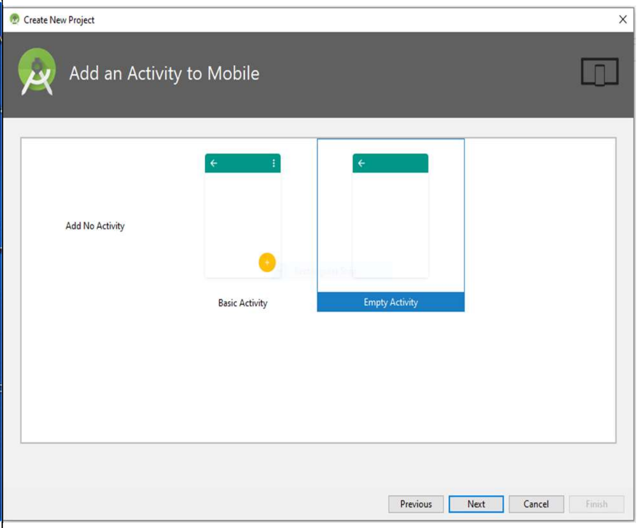
# 3 THE STRUCTURE OF NATIVE PROGRAMS

In this chapter a basic native application would be created. This application would be like the "Hello world" using NDK. After knowing how to create a hello world, it will be worth reviewing briefly the JNI ecosystem. A look will be taken at the signature of a native method in a Java class and how the corresponding method is declared and implemented in the native library. The conclusion of the chapter will look at the mapping of Java types to native language (C/C++) types.

## 3.1 Creating a basic C++ supported project

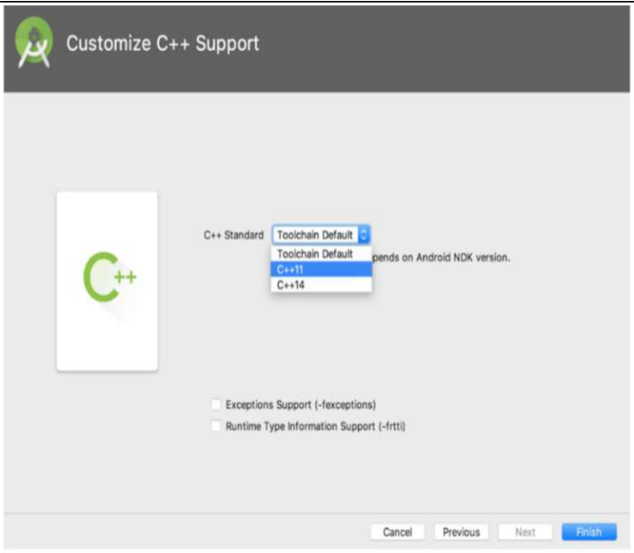
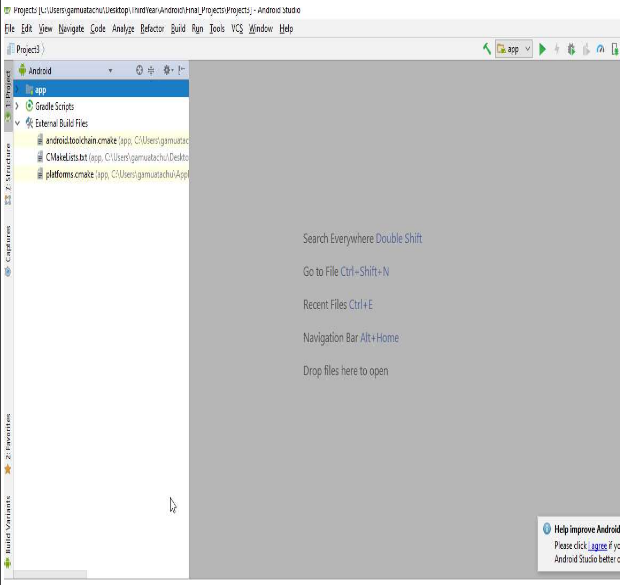
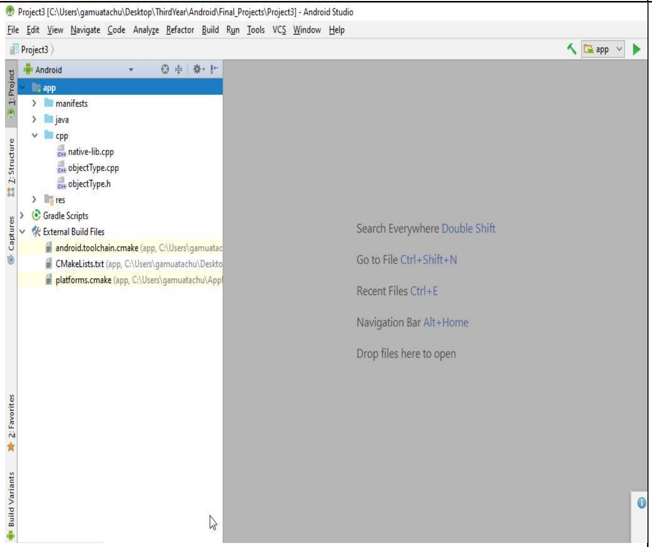
The steps are simple. As shown below (Table 1), support for C++ must initially be included. An "empty activity" should be selected. The definition of the C++ standard used should also be selected. This lets CMake set the relevant compiler and linker flags to use in building the project. The programmer should tick "exceptions support" if the Android project should have compatibility with versions of NDK earlier than NDKr5 and finally the programmer should tick "run time type information support" if there is need for run time support. On clicking "Finish", the External build files under the Android Module will be displayed. This contains the CMakeList.txt file discussed earlier in section 2.5, as well as the native-lib, which exists in the cpp folder under the app module.

TABLE 1. Steps to create a C++ based project

| | |
|---|---|
|  Create New Project<br>Create Android Project<br>Application name: HelloWorldC++<br>Company domain: gamuatachu.example.com<br>Project location: C:\Users\gamuatachu\Desktop\ThirdYear\Android\UniTrial9\HelloWorldC<br>Package name: com.example.gamuatachu.helloworldc<br>[✓] Include C++ support<br>[ ] Include Kotlin support<br>Previous Next Cancel Finish<br>a) C++ support included |  Create New Project<br>Add an Activity to Mobile<br>Add No Activity<br>Basic Activity<br>[✓ Empty Activity]<br>Previous Next Cancel Finish<br>b) select empty activity |

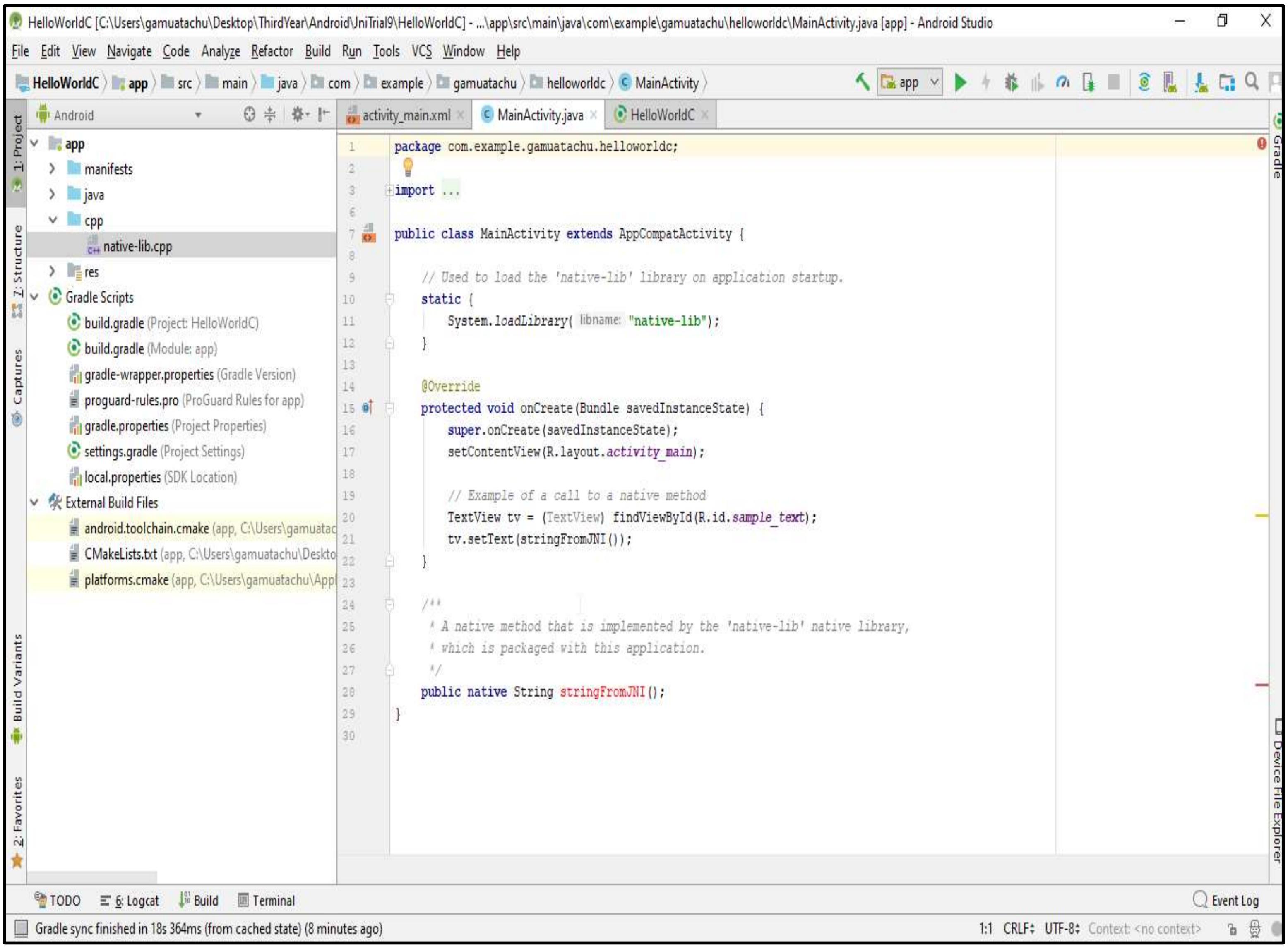
(continues)

TABLE 1. (Continued)

| | |
|---|---|
|   Customize C++ Support<br>C++ Standard: Toolchain Default<br>Toolchain Default<br>C++11<br>C++14<br>...pends on Android NDK version.<br>Exceptions Support (-fexceptions)<br>Runtime Type Information Support (-frtti)<br>Cancel<br>Previous<br>Next<br>Finish c) Selecting C++ standard, exceptions and run time support |   Android<br>app<br>Gradle Scripts<br>External Build Files<br>android.toolchain.cmake<br>CMakeLists.txt<br>platforms.cmake<br>Search Everywhere Double Shift<br>Go to File Ctrl+Shift+N<br>Recent Files Ctrl+E<br>Navigation Bar Alt+Home<br>Drop files here to open d) External Build Files |
|   Project3 [C:\Users\gamuatachu\Desktop\ThirdYear\Android\Final_Projects\Project3] - Android Studio<br>Android<br>app<br>manifests<br>java<br>cpp<br>native-lib.cpp<br>objectType.cpp<br>objectType.h<br>res<br>Gradle Scripts<br>External Build Files<br>android.toolchain.cmake<br>CMakeLists.txt<br>platforms.cmake<br>Search Everywhere Double Shift<br>Go to File Ctrl+Shift+N<br>Recent Files Ctrl+E<br>Navigation Bar Alt+Home<br>Drop files here to open e) native-lib in cpp folder under the app module. | |

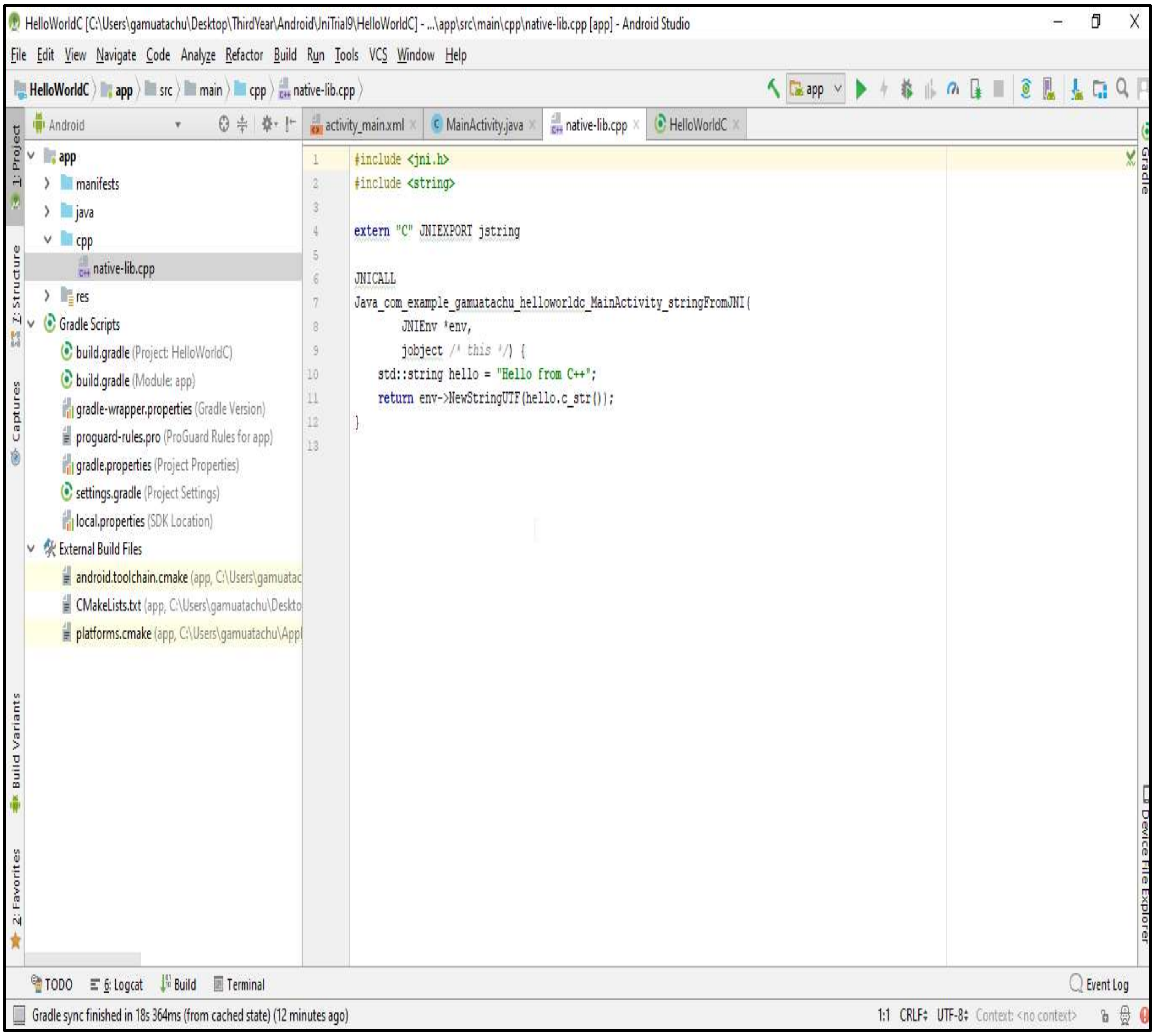
### 3.1.1 Loading of native library

The native library can be loaded anywhere there is need for usage of a native method as shown in figure 8. Following object-oriented paradigm however, it is advisable to have a dedicated class where all the native methods are declared and the native library is loaded. As shown below (FIGURE 8), MainActivity class extends the AppCompatActivity class. The activity will work normally without the MainActivity class extending the AppCompatActivity class. This extends was intended to run the activity on earlier versions of Android. Loading a native library in Java class allows native methods to be called in that class. Additionally the JNI_OnLoad() method is called when the native library is loaded. (Android Developers Documentation 2019; MIT Education 2019; Oracle Documentation 2017.)


HelloWorldC [C:\Users\gamuatachu\Desktop\ThirdYear\Android\JniTrial9\HelloWorldC] - ...\app\src\main\java\com\example\gamuatachu\helloworldc\MainActivity.java [app] - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
HelloWorldC app src main java com example gamuatachu helloworldc MainActivity
Android
app
manifests
java
cpp
native-lib.cpp
res
Gradle Scripts
build.gradle (Project: HelloWorldC)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
External Build Files
android.toolchain.cmake (app, C:\Users\gamuatac
CMakeLists.txt (app, C:\Users\gamuatachu\Deskto
platforms.cmake (app, C:\Users\gamuatachu\Appl
activity_main.xml
MainActivity.java
HelloWorldC
TODO
6: Logcat
Build
Terminal
Event Log
Gradle sync finished in 18s 364ms (from cached state) (8 minutes ago)
1:1 CRLF UTF-8 Context: &lt;no context&gt;

```java
package com.example.gamuatachu.helloworldc;

import ...

public class MainActivity extends AppCompatActivity {

    // Used to load the 'native-lib' library on application startup.
    static {
        System.loadLibrary( libname: "native-lib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Example of a call to a native method
        TextView tv = (TextView) findViewById(R.id.sample_text);
        tv.setText(stringFromJNI());
    }

    /**
     * A native method that is implemented by the 'native-lib' native library,
     * which is packaged with this application.
     */
    public native String stringFromJNI();
}
```

FIGURE 8. native-lib loaded in MainActivity class and declaration of native method


HelloWorldC [C:\Users\gamuatachu\Desktop\ThirdYear\Android\UniTrial9\HelloWorldC] - ...\app\src\main\cpp\native-lib.cpp [app] - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
HelloWorldC app src main cpp native-lib.cpp
app
Android
activity_main.xml
MainActivity.java
native-lib.cpp
HelloWorldC
app
manifests
java
cpp
native-lib.cpp
res
Gradle Scripts
build.gradle (Project: HelloWorldC)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
External Build Files
android.toolchain.cmake (app, C:\Users\gamuatac
CMakeLists.txt (app, C:\Users\gamuatachu\Deskto
platforms.cmake (app, C:\Users\gamuatachu\App
1: Project
7: Structure
Captures
Build Variants
2: Favorites
Gradle
Device File Explorer
TODO
6: Logcat
Build
Terminal
Event Log
1:1 CRLF: UTF-8: Context: <no context>

```cpp
#include <jni.h>
#include <string>

extern "C" JNIEXPORT jstring

JNICALL
Java_com_example_gamuatachu_helloworldc_MainActivity_stringFromJNI(
        JNIEnv *env,
        jobject /* this */) {
    std::string hello = "Hello from C++";
    return env->NewStringUTF(hello.c_str());
}
```

Gradle sync finished in 18s 364ms (from cached state) (12 minutes ago)

FIGURE 9. Corresponding native method of Java class definition in native library

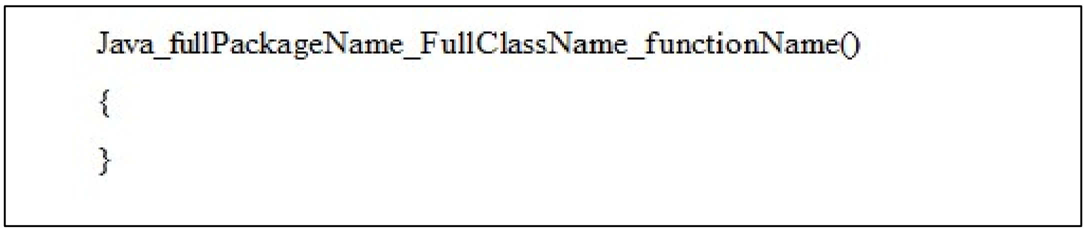
### 3.1.2 The native method, native-lib and other libraries

After loading a native library in a class, a method with the native keyword can then be declared in that class. If the native methods are declared in a dedicated class, an object of this class can be created in the current class to be used to access the native methods. The latter approach is used in this project. The native-lib can then be edited. If there is need for .cpp files and .h files same may be created and added. A path to the .cpp file should be included in the add_library() of the CMakeList.txt file. The native method has a declaration in the Java file which ends with a semicolon. In the native files, however, a native method has a definition. Its header is indicative of the package, class and return type of the method. ( Android Developers Documentation 2019; MIT Education 2019.)

A native method can be declared when the native library is loaded into a Java class. The method must contain the native keyword as can be seen in the "HelloWorldC++" of figure 8. The native keyword specifies that this method is not a Java method but one which will be written or defined in another language. This declaration must be terminated with a semi-colon since the implementation is not done on the Java side. The code in code snippet 2 is a definition of stringFromJNI() method declared as shown in figure 8 and also defined in the native-lib as shown in figure 9. ( Android Developers Documentation 2019; MIT Education 2019.)

```
extern "C" JNIEXPORT jstring

JNICALL

Java_com_example_gamuatachu_helloworldc_MainActivity_stringFromJNI(

JNIEnv *env,

jobject /*this */)

{

   std::string hello ="hello from C++";

   return env->NewStringUTF(hello.c_str() );

}
```

CODE SNIPPET 2. Definition of StringFromJNI Method in native-lib,cpp extern "C" prevents name mangling. This is necessary in case there is need to link a C code to the native method using a C/C++ compatible header file. The compiler will not mangle the name and would therefore know that the C++ method defined in the C++ file and declared in the corresponding C header file are both the same. JNIEXPORT and JNICALL are necessary to register the method and be able to call it from the dynamic table of the built binary (.so file). The native method name is concatenated as shown in code snippet 3. If fullPackageName is for example com.example.gamuatachu.helloworldc, then all the dots (.) have to be replaced by an underscore _ .(MIT Education 2019; Oracle Documentation 2017.)

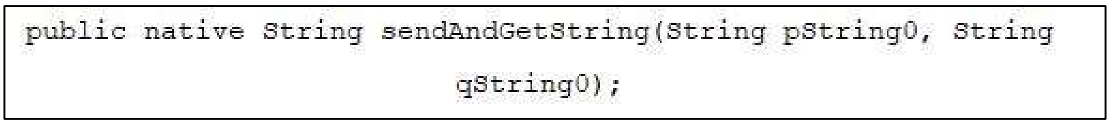


```
Java_fullPackageName_FullClassName_functionName()
{
}
```

CODE SNIPPET 3. Function name in native-lib

## 3.2 Mapping Java and native types

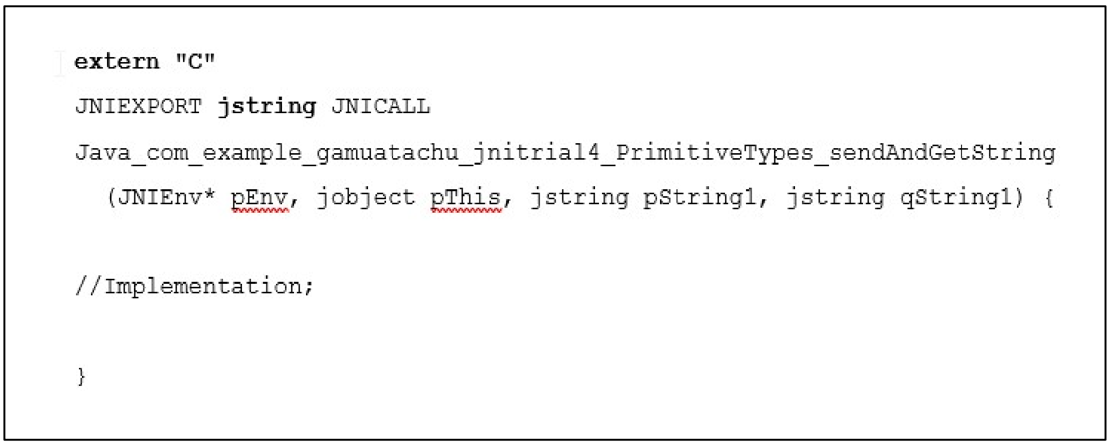
The JNI provides a means to map Java types to native types. Mapping can be done for both primitive types and reference types. The native types provide a way to work with Java types. A primitive Java type is received on the native side as either a jboolean, a jbyte, a jchar, a jshort, a jint, a jlong, a jfloat, a jdouble or void. When sending a primitive variable types back to a Java class it should also be sent as one of jboolean, jbyte, jchar, jshort, jint, jlong, jfloat, jdouble or void. Any of these could be used on the native side just as a boolean, byte, char, short, int, long, float, double or void without explicitly type casting. This is, however, not possible with arrays, strings and other objects sent from the Java side. Consider a method that has been declared as native in a Java class as below (CODE SNIPPET 4). Its corresponding native implementation will look like the code in code snippet 5.



```
public native String sendAndGetString(String pString0, String
                              qString0);
```

CODE SNIPPET 4. Method declared with native keyword in a Java class

The type jstring is the native implementation of the Java type String. The jstrings pString1 and qString1 represents a JNI reference to the Java string objects pString0 and qString0. The jobject pThis is a reference to the object that calls the method sendAndGetString(String pString0, String qString0); in Java. Table 2 and figure 10 summarise the primitive types and reference type mapping. (MIT Education 2019; Oracle Documentation 2017.)



```
extern "C"
JNIEXPORT jstring JNICALL
Java_com_example_gamuatachu_jnitrial4_PrimitiveTypes_sendAndGetString
  (JNIEnv* pEnv, jobject pThis, jstring pString1, jstring qString1) {

//Implementation;


}
```

CODE SNIPPET 5. Native implementation of native Java method sendAndGetString in native-lib

The JNI types also have sizes in bits (TABLE 2). These sizes permit usage of fixed width integer types which were defined since C++11. Definition and declaration of variables on the native side in the various android projects in this thesis is carried out using these fixed width integer types. These types are int8_t, int16_t, int32_t, int64_t, uint8_t and uint16_t. (Gamua 2020.)

TABLE 2. Primitive type mapping from Java to native

| Java type | Native Type | Size in bits |
|---|---|---|
| boolean | jboolean | 8, unsigned |
| byte | jbyte | 8 |
| Char | jchar | 16, unsigned |
| short | jshort | 16 |
| int | jint | 32 |
| long | jlong | 64 |
| float | jfloat | 32 |
| double | jdouble | 64 |
| void | jvoid | - |

Arrays, strings and other objects types are sent as reference types. These reference types must explicitly be converted by calling on the methods of the JNIEnv class through its pointer variable. All reference types are subclassed from the JNI jobject type. Arrays have other JNI sub types. These are the jintArray, jlongArray, jfloatArray, jdoubleArray, jbooleanArray, jbyteArray, jcharArray, jshortArray and jobjectArray. Strings are sent and retrieved from the native side using the JNI jstring type. Class references are retrieved from the native side as a jclass which is an internal typdef of a jobject. It is worth mentioning that the jsize is a typedef of a jint. (Oracle Documentation 2017.)

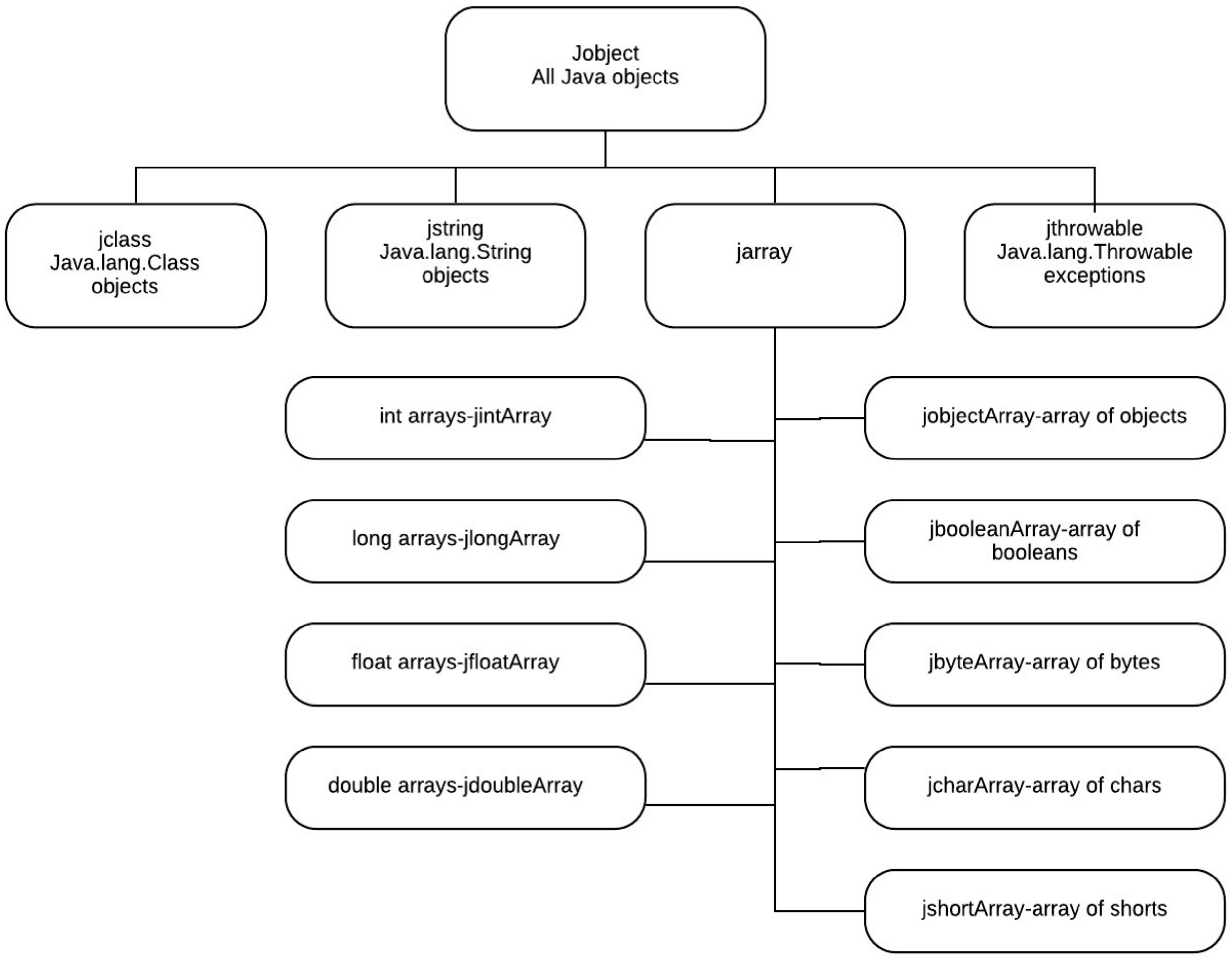


FIGURE 10. Reference type mapping from java to native

## 3.3 The JNI interface pointer

The interface pointer is a pointer which is created per thread JNI data structure. It points to a thread local data which correspondingly points to the JNI function table shared by all threads. The interface pointer permits the manipulation of Java objects and arrays. Native methods receive the JNI interface pointer as their first parameter. Without the JNI interface pointer, native methods would not be able to access functions in the JNI function table. The JNI interface pointer also provides a way to access Java fields and call Java methods. Because it points to per thread created JNI data, it cannot be shared between two threads. (MIT Education 2019; Oracle Documentation 2017.)

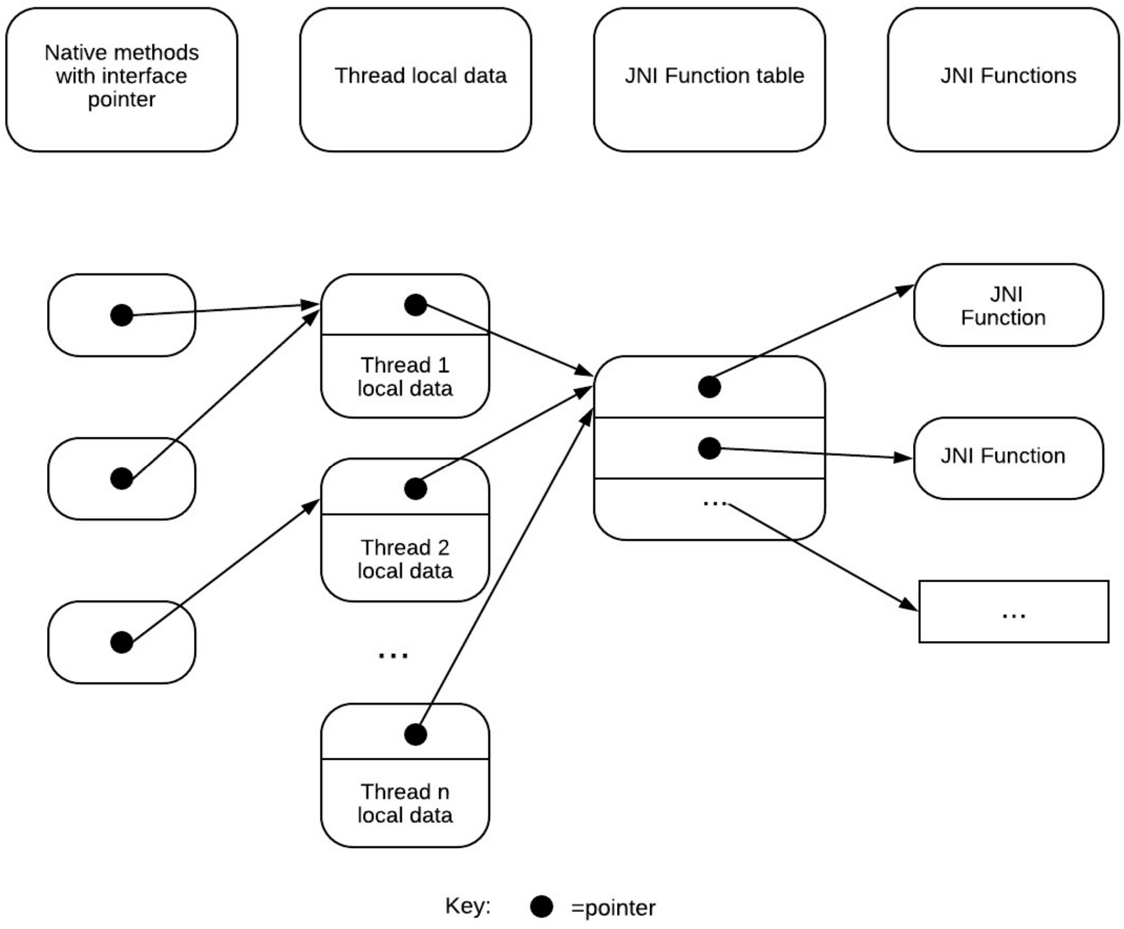


FIGURE 11. JNI interface Pointer

# 4 PRIMITIVE TYPES AND ARRAYS OF PRIMITIVE TYPES

This chapter concentrates on sending and retrieving primitive types and array types to and from the native libraries. Java Strings though not a primitive type, will be treated under primitive types. The Android Studio projects written in this chapter will attempt to send and retrieve Java primitives, strings and arrays to and from native code. These retrieved types will then be printed on the UI. Use shall be made of some basic JNI APIs concerned with primitive, string and array types. Because memory in the native side has to be managed, this will be another concern in this chapter.

## 4.1 Brief look at relationship between classes

In the PrimitiveTypesApp app there are 4 classes vis MainActivity, NativeMsClass, InfoDialog and InputValues (Gamua 2020). InputValues is the C++ class and therefore needs JNI to access some of its resources. Some types, however, like ints do not need JNI APIs. In the ArrayTypes app, there are also 4 classes vis MainActivity, NativeMsClass, InfoDilaog and InputValues (Gamua 2020). The creation of the diagram in figure 12 is possible even though  2 languages are in use in this project. UML is language agnostic (Booch, Rumbaugh, Jacobson 2005).

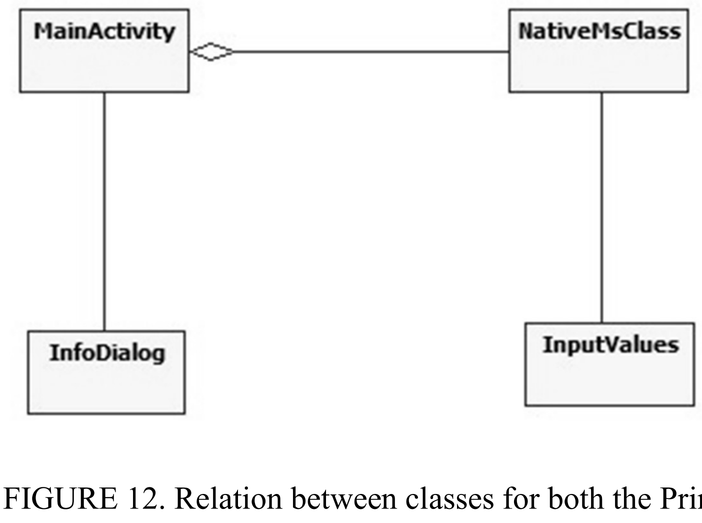
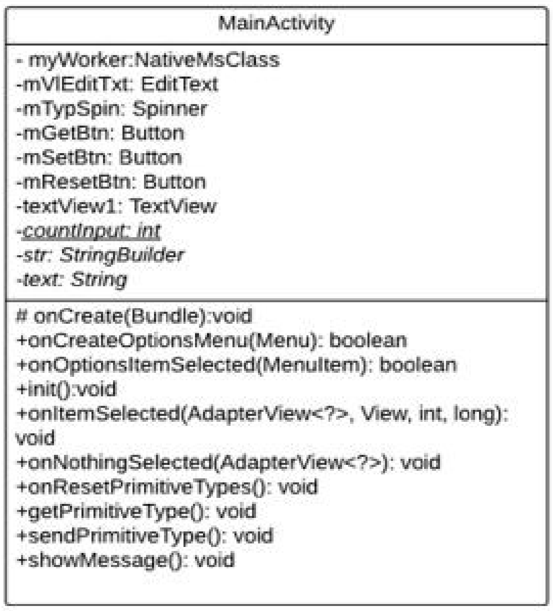


FIGURE 12. Relation between classes for both the PrimitiveTypeApp app and the ArrayType app

## 4.2 Detailed look at class diagrams

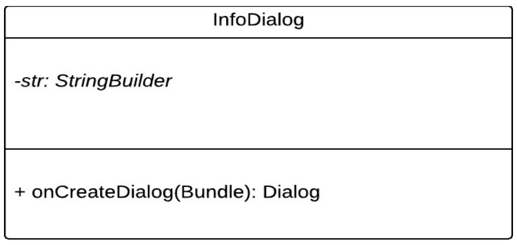
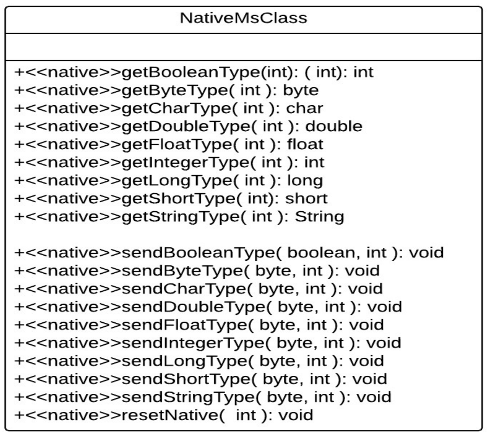
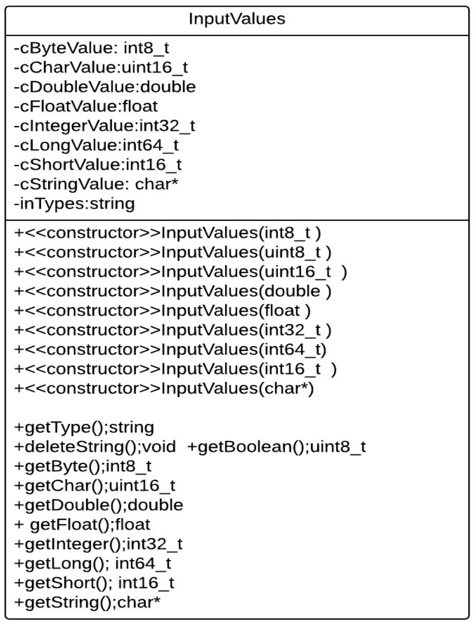
Table 3 presents the details of the classes for the PrimitiveTypeApp and table 4 the classes of the Array-Type app. The implementation of these classes in each app differ even though the main point is to send and retrieve data. On the native side which is of particular interest, the InputValues class of the PrimitiveType app has data members cBooleanValue, cByteValue, cCharValue, cDoubleValue. cFloatValue, cIntegerValue, cLongValue, cShortValue, inTypes and a cStringValue pointer type. The inTypes variable saves information about the type passed. The InputValues class also has constructors to set its data members when an object is instantiated in the native-lib. There are getters methods to get the saved values in the native-lib, a getType method to know the type information saved before calling a respective getter method and a deleteString method to help delete string references. The MainActivity class sends values to the native side via a sendPrimitiveType method that invokes a particular native method and gets values from the native side via a getPrimitiveType method that retrieves saved values from the native side. It also has MenuItem callback methods that either reset the user interface and free memory on the native side or bring up a user interface dialog. The NativeMsClass contains a declaration of all the native methods. The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. A detail description of how elements of one class interact with other elements of other classes is given in the section 4.7. (Gamua 2020.)

TABLE 3. Classes in the PrimitiveTypeApp

| Class | Functions in Brief |
|---|---|
|  MainActivity<br>- myWorker:NativeMsClass<br>-mVlEditTxt: EditText<br>-mTypSpin: Spinner<br>-mGetBtn: Button<br>-mSetBtn: Button<br>-mResetBtn: Button<br>-textView1: TextView<br>-countInput: int<br>-str: StringBuilder<br>-text: String<br># onCreate(Bundle):void<br>+onCreateOptionsMenu(Menu): boolean<br>+onOptionsItemSelected(MenuItem): boolean<br>+init():void<br>+onItemSelected(AdapterView<?>, View, int, long): void<br>+onNothingSelected(AdapterView<?>): void<br>+onResetPrimitiveTypes(): void<br>+getPrimitiveType(): void<br>+sendPrimitiveType(): void<br>+showMessage(): void | The MainActivity class sends values to the native side via the sendPrimitiveType method that invokes a particular native method and gets values from the native side via the getPrimitive-Type method that retrieves saved values from the native side. It also has MenuItem callback methods that either reset the user interface and free memory on the native side or bring up a user interface dialog. |

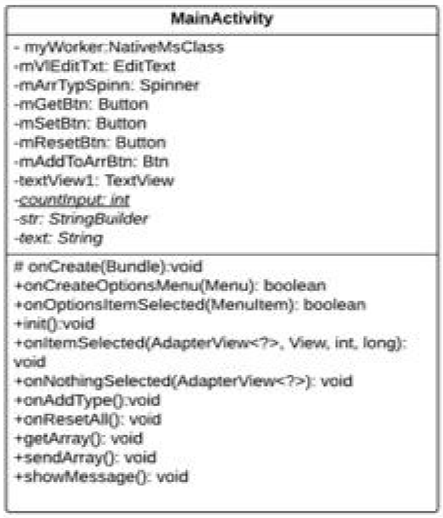
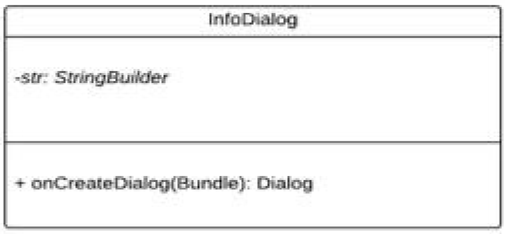
(Continues)

TABLE 3. (continued)

| | |
|---|---|
|  InfoDialog<br>*-str: StringBuilder*<br>+ onCreateDialog(Bundle): Dialog | The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. The dialog button is found in the menu and can be displayed anytime the user wants. |
|  NativeMsClass<br>+<<native>>getBooleanType(int): ( int): int<br>+<<native>>getByteType( int ): byte<br>+<<native>>getCharType( int ): char<br>+<<native>>getDoubleType( int ): double<br>+<<native>>getFloatType( int ): float<br>+<<native>>getIntegerType( int ): int<br>+<<native>>getLongType( int ): long<br>+<<native>>getShortType( int): short<br>+<<native>>getStringType( int ): String<br>+<<native>>sendBooleanType( boolean, int ): void<br>+<<native>>sendByteType( byte, int ): void<br>+<<native>>sendCharType( byte, int ): void<br>+<<native>>sendDoubleType( byte, int ): void<br>+<<native>>sendFloatType( byte, int ): void<br>+<<native>>sendIntegerType( byte, int ): void<br>+<<native>>sendLongType( byte, int ): void<br>+<<native>>sendShortType( byte, int ): void<br>+<<native>>sendStringType( byte, int ): void<br>+<<native>>resetNative( int ): void | The NativeMsClass contains a declaration of all the native methods. |
|  InputValues<br>-cByteValue: int8_t<br>-cCharValue:uint16_t<br>-cDoubleValue:double<br>-cFloatValue:float<br>-cIntegerValue:int32_t<br>-cLongValue:int64_t<br>-cShortValue:int16_t<br>-cStringValue: char*<br>-inTypes:string<br>+<<constructor>>InputValues(int8_t )<br>+<<constructor>>InputValues(uint8_t )<br>+<<constructor>>InputValues(uint16_t )<br>+<<constructor>>InputValues(double )<br>+<<constructor>>InputValues(float )<br>+<<constructor>>InputValues(int32_t )<br>+<<constructor>>InputValues(int64_t)<br>+<<constructor>>InputValues(int16_t )<br>+<<constructor>>InputValues(char*)<br>+getType();string<br>+deleteString();void +getBoolean();uint8_t<br>+getByte();int8_t<br>+getChar();uint16_t<br>+getDouble();double<br>+ getFloat();float<br>+getInteger();int32_t<br>+getLong(); int64_t<br>+getShort(); int16_t<br>+getString();char* | This class also has constructors that are called when an object is instantiated in the native-lib. It also has methods to help save the values passed from java and methods to retrieve those values from the native side. There are also met-hods to manage memory. |

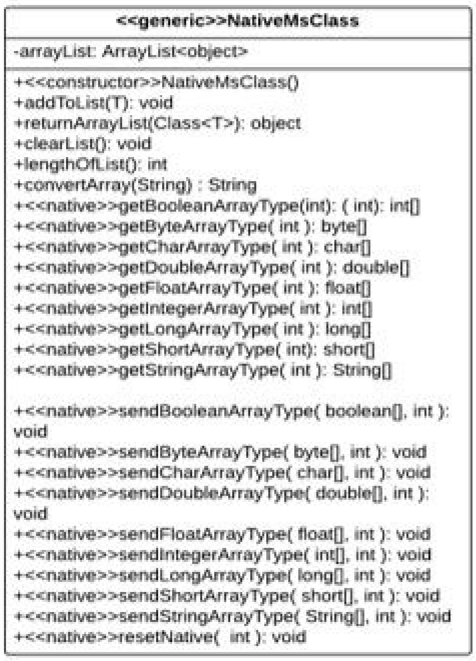
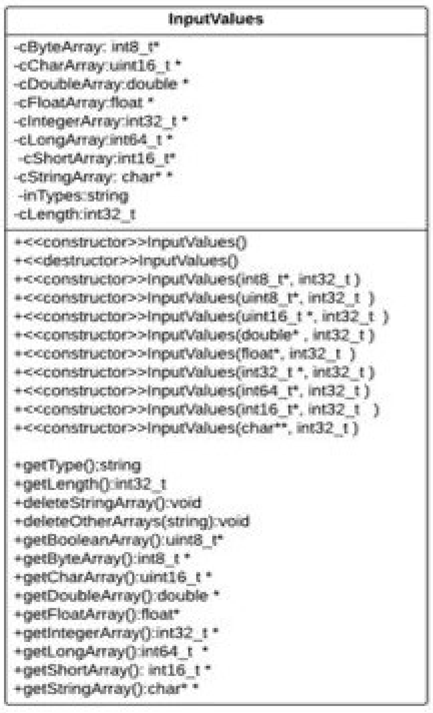
On the native side of the Array type, the InputValues class has pointer types cBooleanArray, cByteArray, cCharArray, cDoubleArray. cFloatArray, cIntegerArray, cLongArray, cShortArray and cStringArray. It also has an inType variable to save information about the reference type passed, and a cLength variable that saves information about the length of the array. Its constructors set these various pointer type references when an object of this class is created in the native-lib. There are getter methods to get the saved pointed references in the native-lib and the length of the arrays, a getType method to know the type of reference saved before calling a getter method and a deleteStringArray method to help delete string array references. The MainActivity class sends values to the native side via a sendArray method that invokes a particular native method and gets values from the native side via a getArray method that retrieves saved values from the native side. It also has MenuItem callback methods that either reset the user interface and also frees memory on the native side or bring up a user interface dialog. The onAdd method adds values to an arrayList created in the generic NativeMsClass. The generic NativeMsClass contains a declaration of all the native methods. It also contains an addToList method that adds array values to the arrayList, a returnArrayList that returns the arrayList and a convertArrayList that formats values retrieved from the native side. The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. A detail description of classes and how elements of one class interact with elements of other classes is given in the section 4.7. (Gamua 2020.)

TABLE 4. Classes in the ArrayType app

| Class | Functions in Brief |
|---|---|
|   MainActivity<br>- myWorker:NativeMsClass<br>-mVlEditTxt: EditText<br>-mArrTypSpinn: Spinner<br>-mGetBtn: Button<br>-mSetBtn: Button<br>-mResetBtn: Button<br>-mAddToArrBtn: Btn<br>-textView1: TextView<br>-countInput: int<br>-str: StringBuilder<br>-text: String<br># onCreate(Bundle):void<br>+onCreateOptionsMenu(Menu): boolean<br>+onOptionsItemSelected(MenuItem): boolean<br>+init():void<br>+onItemSelected(AdapterView<?>, View, int, long): void<br>+onNothingSelected(AdapterView<?>): void<br>+onAddType():void<br>+onResetAll(): void<br>+getArray(): void<br>+sendArray(): void<br>+showMessage(): void | The MainActivity class sends values to the native side via the sendArray method that invokes a particular native method and gets values from the native side via the getArray method that retrieves saved values from the native side. There are methods that help build arrays by calling on methods in the generic NativeMsClass. It also has MenuItem callback methods that either reset the user interface and thereby freeing memory on the native side or bring up a user interface dialog. |
|   InfoDialog<br>-str: StringBuilder<br>+ onCreateDialog(Bundle): Dialog | The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. The dialog button is found in the menu and can be displayed anytime the user wants. |

(Continues)

TABLE 4. (Continued)

| | |
|---|---|
| <br>**<<generic>>NativeMsClass**<br>-arrayList: ArrayList<object><br>+<<constructor>>NativeMsClass()<br>+addToList(T): void<br>+returnArrayList(Class<T>): object<br>+clearList(): void<br>+lengthOfList(): int<br>+convertArray(String) : String<br>+<<native>>getBooleanArrayType(int): ( int): int[]<br>+<<native>>getByteArrayType( int ): byte[]<br>+<<native>>getCharArrayType( int ): char[]<br>+<<native>>getDoubleArrayType( int ): double[]<br>+<<native>>getFloatArrayType( int ): float[]<br>+<<native>>getIntegerArrayType( int ): int[]<br>+<<native>>getLongArrayType( int ): long[]<br>+<<native>>getShortArrayType( int): short[]<br>+<<native>>getStringArrayType( int ): String[]<br>+<<native>>sendBooleanArrayType( boolean[], int ): void<br>+<<native>>sendByteArrayType( byte[], int ): void<br>+<<native>>sendCharArrayType( char[], int ): void<br>+<<native>>sendDoubleArrayType( double[], int ): void<br>+<<native>>sendFloatArrayType( float[], int ): void<br>+<<native>>sendIntegerArrayType( int[], int ): void<br>+<<native>>sendLongArrayType( long[], int ): void<br>+<<native>>sendShortArrayType( short[], int ): void<br>+<<native>>sendStringArrayType( String[], int ): void<br>+<<native>>resetNative(  int ): void | The generic NativeMsClass contains a declaration of all the native methods. It also contains an addToList method that adds array values to the arrayList, a returnArrayList that returns the arrayList and a convertArrayList that formats values retrieved from the native side. |
| <br>**InputValues**<br>-cByteArray: int8_t*<br>-cCharArray:uint16_t *<br>-cDoubleArray:double *<br>-cFloatArray:float *<br>-cIntegerArray:int32_t *<br>-cLongArray:int64_t *<br>-cShortArray:int16_t*<br>-cStringArray: char* *<br>-inTypes:string<br>-cLength:int32_t<br>+<<constructor>>InputValues()<br>+<<destructor>>InputValues()<br>+<<constructor>>InputValues(int8_t*, int32_t )<br>+<<constructor>>InputValues(uint8_t*, int32_t )<br>+<<constructor>>InputValues(uint16_t *, int32_t )<br>+<<constructor>>InputValues(double* , int32_t )<br>+<<constructor>>InputValues(float*, int32_t )<br>+<<constructor>>InputValues(int32_t *, int32_t )<br>+<<constructor>>InputValues(int64_t*, int32_t )<br>+<<constructor>>InputValues(int16_t*, int32_t )<br>+<<constructor>>InputValues(char**, int32_t )<br>+getType():string<br>+getLength():int32_t<br>+deleteStringArray():void<br>+deleteOtherArrays(string):void<br>+getBooleanArray():uint8_t*<br>+getByteArray():int8_t *<br>+getCharArray():uint16_t *<br>+getDoubleArray():double *<br>+getFloatArray():float*<br>+getIntegerArray():int32_t *<br>+getLongArray():int64_t *<br>+getShortArray(): int16_t *<br>+getStringArray():char* * | This class also has constructors to set values when an object of this class is created in the native-lib. It also has methods to help save the values passed from java and methods to retrieve those values from the native side. It has methods to manage memory |

## 4.3 Brief look at user interfaces

In this section, a brief look will be taken at the user interfaces. How to enter values and types for both the PrimitiveTypeApp and the arrayType app will be looked at. Strings are treated under primitive types. For primitive types, values are entered under the value field and the type of value entered is selected from the menu of the spinner. An integer value of 20 is entered as shown in table 5. (Gamua 2020.)

TABLE 5. How to Enter values into the PrimitiveTypeApp

| | |
|---|---|
| value | 20 |
| Type | Integer |

For the PrimitiveType app, the SEND button is used to enter values one at a time. The SEND Button has an onClickListener attached that calls the sendPrimitiveType() method. Depending on the type chosen the sendPrimitiveType() method will activate the required case. In case there is a mismatch of a value and its type, an exception will be caught, and the user will be informed that there is a mismatch of value and type. The GET button recovers the values that have been added from the native class through the getPrimitiveType() method and displays on the user interface. There are two items in the menu whose showAsAction is never. These are RESET and the INFO. RESET resets the user interface and cleans the memory on the C++ side. Figure 13 shows the user interface for the PrimitiveTypeApp. (Gamua 2020.)

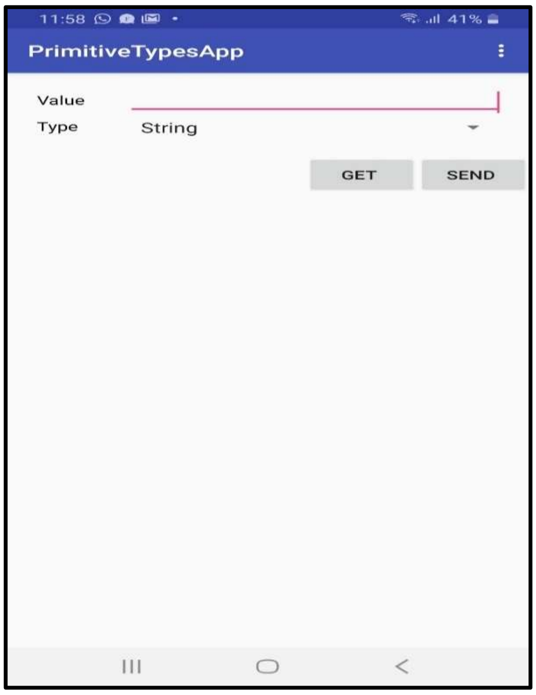


FIGURE 13. User interface to enter and retrieve primitive types

### 4.3.1 Array types

Values are entered under the value field and the type of value entered is selected from the  menu of the spinner. Consider an integer array such as {20,40,80}. To enter this array, 20 is entered into the value field and the ADD button clicked. The same process is followed for 40 and 80 being careful to keep the type as IntegerArrayType. Table 6 shows the first input of the value 20. The ADD button must clicked after each entry as only then is the value added to the current array by the addToList method. (Gamua 2020.)

TABLE 6.  First value in array entered and ADD button is clicked

| | |
|---|---|
| value | 20 |
| Type | IntegerArrayType |

The ADD button has an onClickListener that calls the onAdd() method that creates an array in the NativeMsClass. After creating the array with the ADD button, the SEND ARRAY should be pressed to send the array to the native side. The SEND ARRAY Button has an onClickListener attached that calls the sendArray method to send this created array. Depending on the type chosen the sendArray method will activate the required case that will then transfer the array data to the native class. In case there is a mismatch of a value and its type, an exception will be caught, and the user will be informed that there is a mismatch of value and type. The GET ARRAY button recovers values from the native class through the getArray method and displays on the user interface. There are two items in the menu whose showAsAction is never. These are RESET and the INFO. RESET resets the user interface and cleans the memory on the C++ side.  Figure 14 shows the user interface for the ArrayTypes app. (Gamua 2020.)

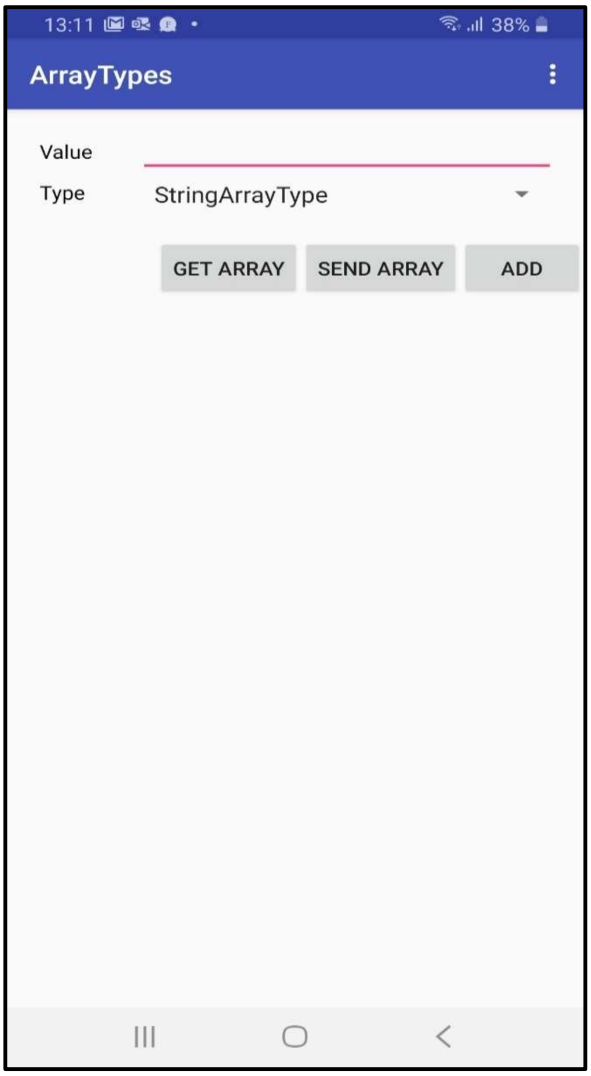
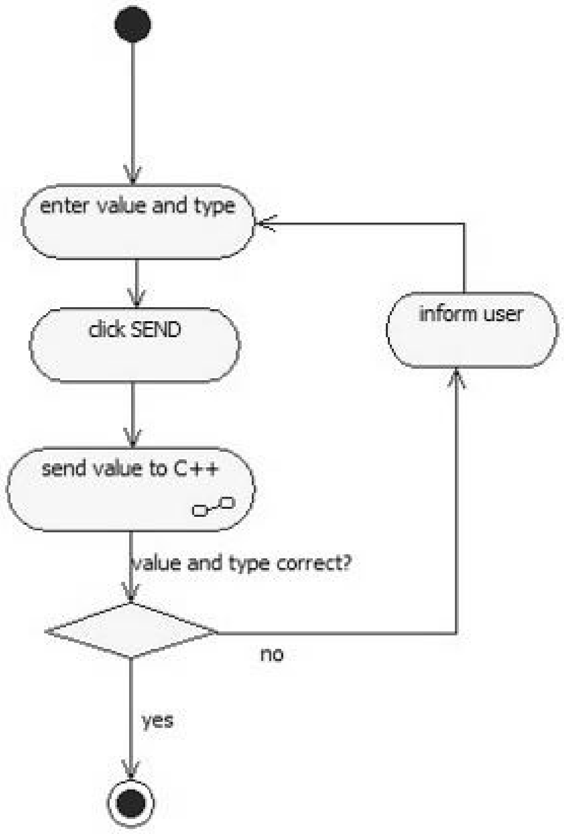


FIGURE 14. User interface to enter and retrieve array types

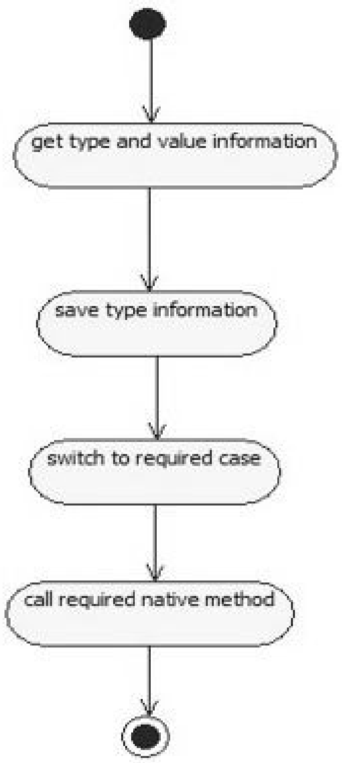
## 4.4 Activity Diagrams

The activity diagrams are presented below. All activities have sub activities that are presented alongside. The activity diagrams are drawn for both the send and get use cases. If there is a mismatch for value and type in the send use case the user will be informed. If there are no values saved on the native side and the user clicks GET or GET ARRAY he will be informed no values are saved. The tables are given headings for each use case.

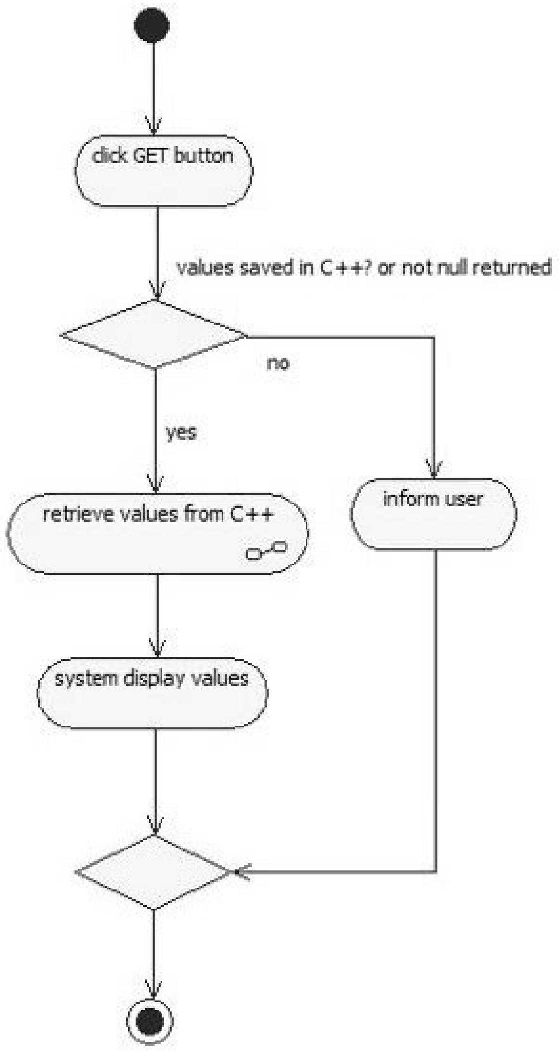
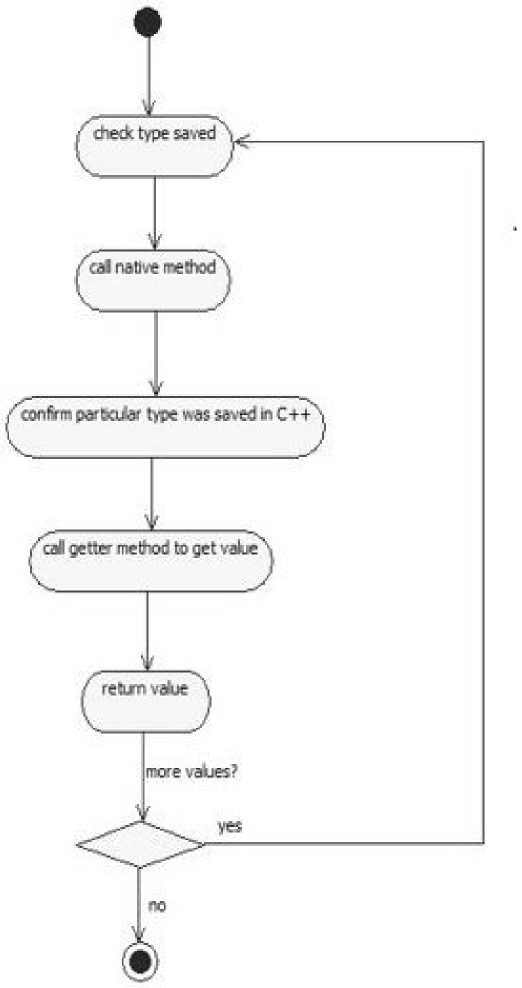
SEND use case for PrimitiveTypeApp app

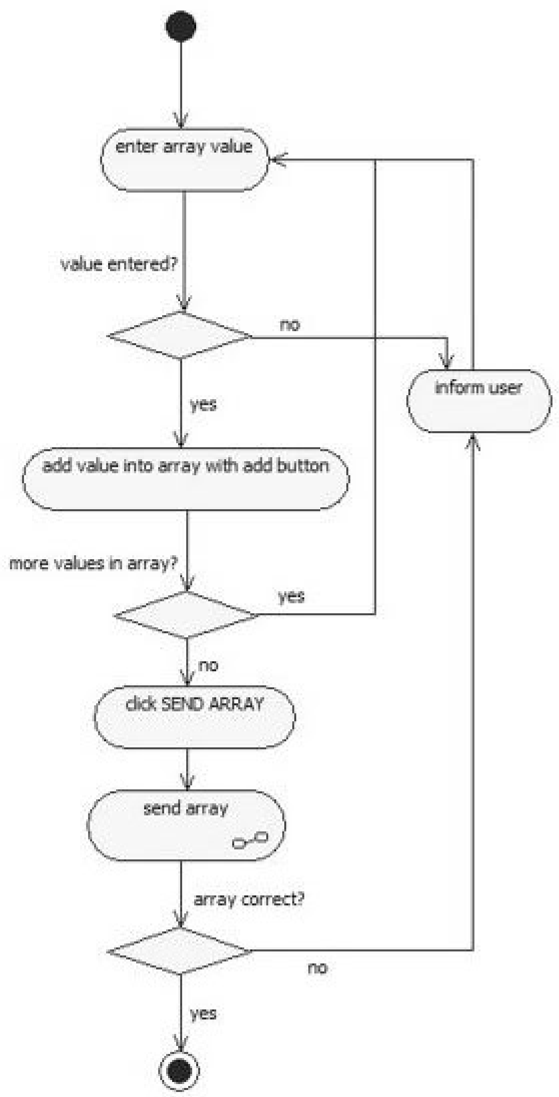
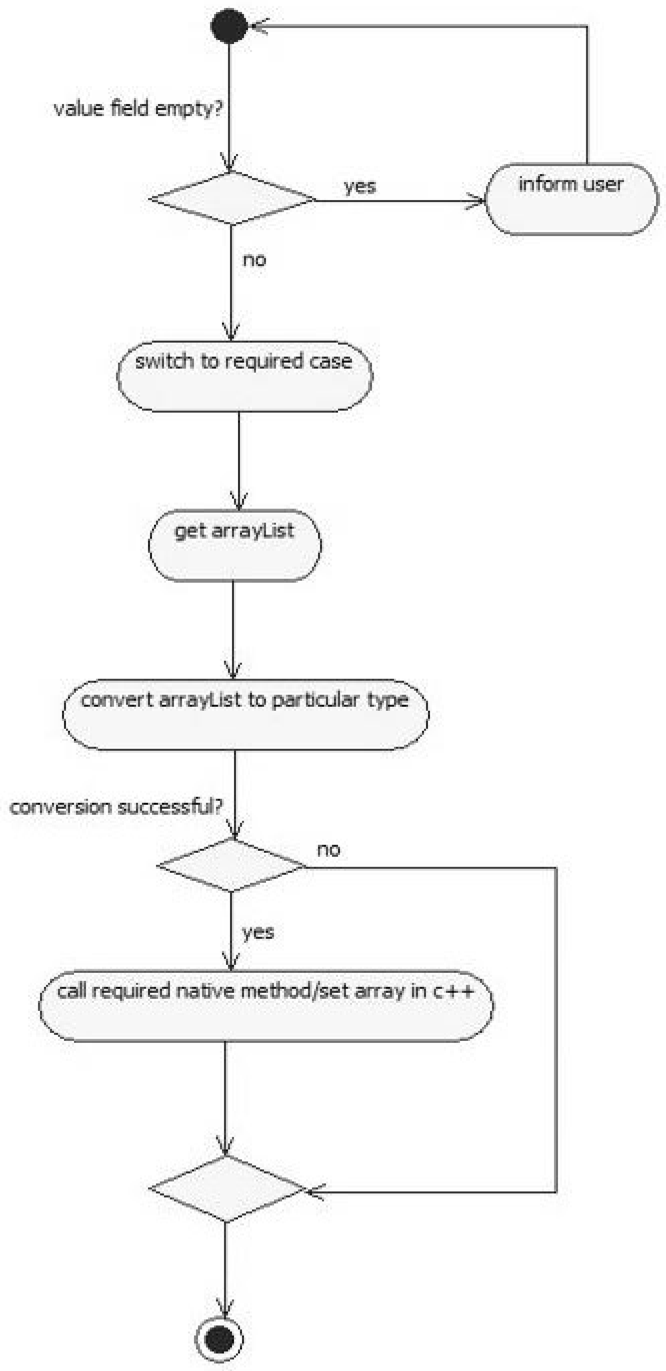


a) Activity Diagram to send to the native side

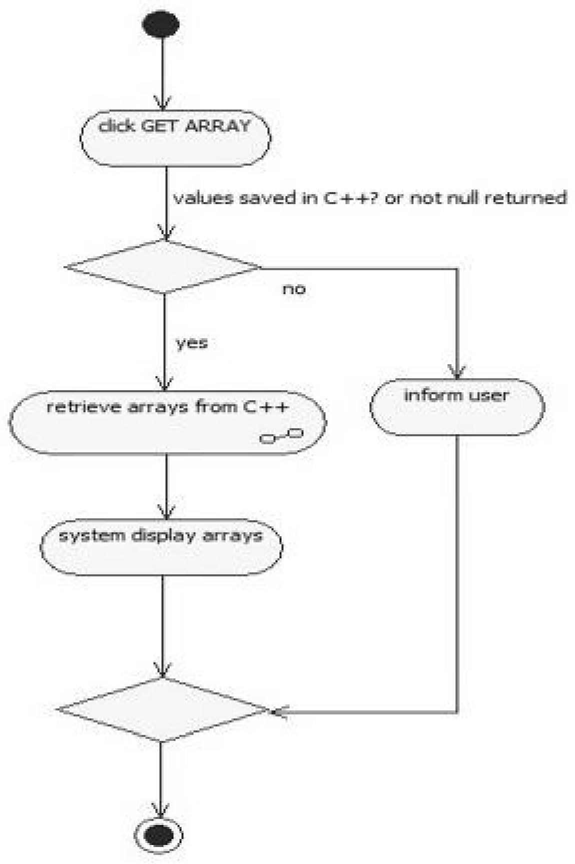


b) Send value to C++ sub activity

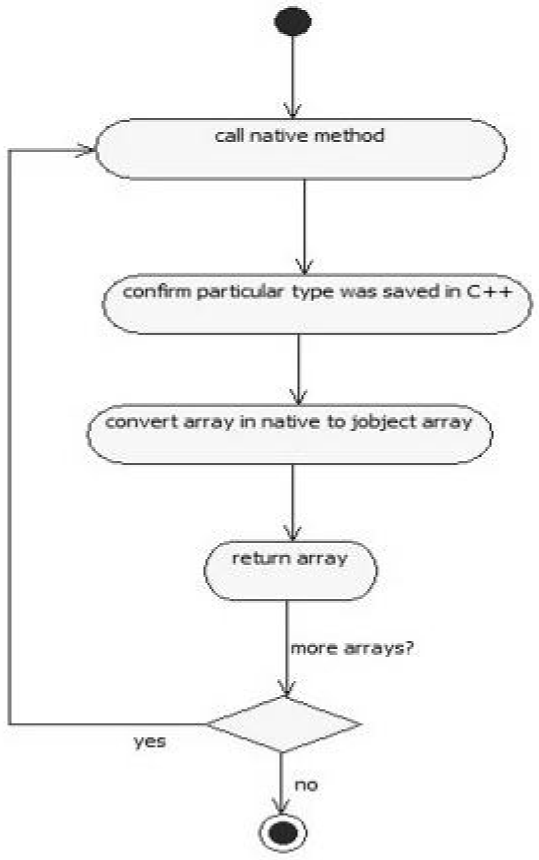
| GET use case for PrimitiveTypeApp app | |
|---|---|
|  click GET button<br>values saved in C++? or not null returned<br>no<br>yes<br>retrieve values from C++<br>inform user<br>system display values<br>a) activity diagram on clicking GET |  check type saved<br>call native method<br>confirm particular type was saved in C++<br>call getter method to get value<br>return value<br>more values?<br>yes<br>no<br>b) retrieve values from C++ sub activity |

| SEND ARRAY use case for ArrayTypes app | |
|---|---|
|   enter array value<br>value entered?<br>no<br>inform user<br>yes<br>add value into array with add button<br>more values in array?<br>yes<br>no<br>click SEND ARRAY<br>send array<br>array correct?<br>no<br>yes a) Activity Diagram to send to the native side |   value field empty?<br>yes<br>inform user<br>no<br>switch to required case<br>get arrayList<br>convert arrayList to particular type<br>conversion successful?<br>no<br>yes<br>call required native method/set array in c++ b) send Array sub activity |

GET ARRAY use case for ArrayTypes app



a) Activity Diagram on clicking GET ARRAY


call native method
confirm particular type was saved in C++
convert array in native to jobject array
return array
more arrays?
yes
no

b) retrieve values from C++ sub activity

## 4.5 Understanding the programs

Upon launching of the application, all parameters are initialized and in the init() method and InfoDialog class object is instantiated. A visible outcome of this is that there is a dialog window that pops up any time the app is launched. This window directs the user on how to use the app. The native-library is loaded in the NativeMsClass class. On the native side the JNI_onload is called once. It caches global variable settings. The version number which contains an integer is then returned. Version numbers can be JNI_VERSION_1_1, JNI_VERSION_1_2, JNI_VERSION_1_4, JNI_VERSION_1_6, JNI_VERSION_1_8, JNI_VERSION_1_9 and JNI_VERSION_1_10. The program in this project uses JNI_VERSION_1_6. (MIT Education 2003; Oracle Documentation 2017; Gamua 2020.)

### 4.5.1 String and primitive types

After entering a value in the value field and selecting the type from the menu of the spinner, the SEND button is pressed/clicked to enter the value. The SEND button has an onClickListener attached that calls the sendPrimitiveType method. In this method there is a String array called checkType that keeps track of the types entered. The native class methods are called depending on the case and the countInput which counts the number of inputted values is incremented. (Gamua 2020.)

For Boolean types myWorker.sendBooleanType() is called with arguments (true, countInput) or (false, countInput). The arguments depend on whether the user enters 1 or 0. The sendBooleanType method which is a natively declared method in the NativeMsClass calls its corresponding native method in native-lib.cpp. In native-lib.cpp, a new object is created. The jboolean type which is used as a parameter is automatically casted to its corresponding native type. The JVM ensures that true=1 and false=0. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

For Byte types myWorker.sendByteType() is called with arguments (Byte.parseByte(inputString), countInput). On the native side the jbyte value nByte passed is simply casted when creating the new object. Floats, Integers, Longs, Short, and Double types all follow the same pattern as Byte types. On the native side, they are also casted to their respective types and there is no need to use JNI APIs to manipulate these types. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

myWorker.sendCharType() is called with arguments (inputString.charAt(0), countInput) for Char types. On the native side the jchar value nChar passed is simply casted when creating the new object. myWorker.sendStringType(inputString, countInput) is called for String types. On the native side the JNI API must be called to manipulate the string reference passed. Java strings are objects so they cannot just be casted. To copy the JNI string to a native buffer use must be made of the JNI APIs GetStringUTFLength and GetStringUTFRegion. The first method gets the length of a string and the second copies it to a C++ memory buffer knowing the length and the buffer name. A '\0' is appended to the buffer for string termination. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

If the GET button is pressed, the getPrimitiveType method is called to retrieve a primitive type. For Boolean types, myWorker.getBooleanType(i) is called if checkType[i] is a Boolean, i being a variable which increments to the number of entries. On the native side the incomingValue[jObjectNum] object gets its type and compares to see if the inTypes variable was set as a BooleanType when the object was created. If yes, a 1 or 0 as jint is returned since Booleans are set as 1 or 0 on the C++ side. On the Java side, if 1 or 0 is returned, it is appended to the stringBuilder str. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

If the GET button is pressed and a Byte type had been entered, myWorker.getByteType(i) is called if checkType[i] is a Byte, i being a variable which increments to the number of entries done. On the native side the incomingValue[jObjectNum] object gets its type and compares to see if the inTypes variable was set as a ByteType when the object was created. If yes, the Byte type is returned as a jbyte. On the Java side, it is appended to the stringBuilder str. Floats, Integers, Longs, Short, and Double types all follow the same pattern as Byte types albeit with their respective JNI types. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

If the GET button is pressed and a Char type had been entered myWorker.getCharType(i) is called if checkType[i] is a Char. On the native side the incomingValue[jObjectNum] object gets its type and compares to see if the inTypes variable was set as a CharType when the object was created. If yes, the Char type is returned as a jchar. On the Java side, it is appended to the stringBuilder str. For String types, myWorker.getStringType(i) is called if a String had been entered and checkType[i] is a String. On the native side the incomingValue[jObjectNum] object gets its type and compares to see if the inTypes variable was set as a StringType when the object was created. If yes, a pointer variable that points to the returned C++ string of the object is created. Use is made of the JNI API method NewStringUTF() to create a jstring object that is returned. (Gamua 2020.)

### 4.5.2 Array type

For array types, there are not two but three buttons that permit the input and output of data. For any array, use is made of the ADD button to create and the SEND ARRAY button to send the array reference to the native side. Clicking on GET ARRAY button will retrieve the array from the native side. The ADD button has an onClickListener that calls the onAddType method. This method takes the primitive value to be added to the array and calls the addToList method of the NativeMsClass through the myWorker object. The ADD button must be clicked successfully after each primitive or string type which makes up the array. The NativeMsClass in this case is of generic type and the addToList method is also of generic type. The addToList method adds the primitive values to be added to an arrayList. After all the primitive types have been entered, clicking the SEND ARRAY button calls the sendArray method. In that method the arrayList is returned through the myWorker object and the returnArrayList method that takes a generic class type. It returns only arrays of a particular type and not combinations of types. As an example, consider selecting to enter an integer array using the menu of the spinner. An integer array such as {4,7,10,4} will be accepted and not {4, 7, 10.9, 4}. The user only gets informed after clicking the SEND ARRAY button as an exception is caught and displayed. (Gamua 2020.)

For Boolean types, myWorker object with the help of the returnArrayList method returns an arrayList of type Object if the “SEND ARRAY” button is clicked. The arrayList is then converted into Boolean types and saved in the newly created Boolean array. The Boolean array is then sent to the native class by calling the sendBooleanArrayType method. In the native-lib.cpp the corresponding method creates an array pointer of type uint8_t*. This points to Booleans on the heap with a length of length. The array elements are extracted into their JNI types using the GetBooleanArrayElements API and they are pointed to in memory by a pointer arrayTmp. To convert to a C++ bool type the elements pointed to by arrayTmp are compared to JNI_TRUE. A pointed jboolean element that is true will return a 1 while a jboolean that is false will return a 0. This is the easiest way to copy from the jboolean array type to a C++ bool array type. (Gamua 2020.)

For Byte types, myWorker object with the help of the returnArrayList method returns an arrayList of type Object if the “SEND ARRAY” button is clicked. The arrayList is then converted into byte types and saved in the newly created Byte array. The Byte array is then sent to the native class by calling the sendByteArrayType method. On the native side the JNI API GetArrayLength is used to get the length of the JNI array type passed from the Java side. A byte array is then created with the new keyword having the exact same length as the array sent. This array is of type int8_t as this is the type that will rightly save a byte. The JNI array is copied to the array created using the GetByteArrayRegion method of the JNI API. A new object is created with constructor parameters being the copied array and its length. Floats, Integers, Longs, Short, and Double types all follow the same pattern as Byte types albeit with their respective JNI types and C++ types. (Gamua 2020.)

For Char types, myWorker object with the help of the returnArrayList method returns an arrayList of type Object if the “SEND ARRAY” button is clicked. The arrayList is converted into chars and saved in the newly created char array. The char array is sent to the native class by calling the sendCharArray-Type method. On the native side the JNI API GetArrayLength is used to get the length of the JNI array type passed from the Java side. A char pointer reference is created with the new keyword having the exact same length as the array received from Java. This pointer will point to types of uint16_t. uint16_t is the right type to save a char. The JNI array is copied using the GetCharArrayRegion to the heap and the pointer reference array created with "new" made to reference this string on the heap. A new object is created with constructor parameters being the copied array and its length. (Gamua 2020.)

Java String types are not same as primitive types. Whether they be arrays or not, their treatment is different. In the case of String arrays, myWorker object with the help of the returnArrayList method returns an arrayList of type Object if the "SEND ARRAY" button is clicked. The arrayList is then converted to Strings and saved in the newly created String array. The String array reference is sent to the native side by calling the sendStringArrayType method. On the native side, the array length is first determined using the GetArrayLength API. A double char array pointer is created to house the individual strings in this array. Because a string is an object, use is made of the GetObjectArrayElement API method to extract the individual Strings in the for loop that follows. For each String in the array extracted, its length is determined using the GetStringUTFLength method. A reference of a string memory location on the heap with the new keyword is created using the array reference created before the for loop. The array reference refers to a char sequence of characters on the heap whose length is given by GetStringUTFLength plus 1. Subsequent references in the for loop will refer to each string extracted. For example, array[0] refers to the first string extracted, array[1] refers to the second string extracted etc. A new object is then created with constructor parameters being the copied array reference and its length. (Gamua 2020.)

If the GET ARRAY Button is clicked the getArray method is called. If one of arrays that were created prior and sent to the native side is a Boolean array, myWorker.getBooleanArrayType(i) is called if the null check is ruled out. On the native side, if the object is found to have been saved as a BooleanArrayType, a jbooleanArray is created using the NewBooleanArray API. This method takes the length of the array as one parameter. The array created is filled with Boolean elements using the SetBooleanArrayRegion API method. The array reference is returned to the Java side as a jbooleanArray. Floats, Integers, Longs, Short, Double, Bytes and Chars all follow the same pattern albeit with their respective JNI types. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

If the GET ARRAY Button is clicked the getArray method is called. If one of arrays that were created prior and sent to the native side is a String array, myWorker.getBooleanArrayType(i) is called if the null check is ruled out. On the native side, a check is made to see if the object was saved as a StringArrayType. If so, a new jobjectArray is created with parameters being the length of the saved string and the jclass global variable StringClass which tells the API the class of array object to create. For each string reference that was saved on the heap, a new string is made from the returned pointed string. These strings are then added subsequently to the jobjectArray variable. After this process finishes, the jobjectArray is returned. (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

## 4.6 Memory

Memory management is handled both on the Java side and on the native side. On the Java side, memory cleanup is the responsibility of the garbage collector. On the native side however, memory must be managed by the programmer. Management of memory on the native side is carried out using pointers, destructors and various JNI types. There exist JNI types for blocking the garbage collector from performing a clean-up of memory references. (MIT Education 2019; Oracle Documentation 2017.)

### 4.6.1 Primitive types

There is just one pointer that is used on the native side. This is the char pointer which is used to create the string. After clicking/pressing the RESET menu button, this char pointer reference needs to be deleted and then subsequently the reference to the object must be deleted in case the object was a StringType. In case the object was not a StringType object, only the reference to the object is deleted. (Gamua 2020.)

### 4.6.2 Array types

Here the object is deleted with the help of a destructor. Options exist to delete the pointer references for all the types. For a String array type, the references are deleted with the help of the deleteStringArray() method. In the deleteStringArray() method, all the references pointing to the strings in the heap are first deleted and then the reference to the references is deleted. For all the other array types, the array references are simply deleted. (Gamua 2020.)

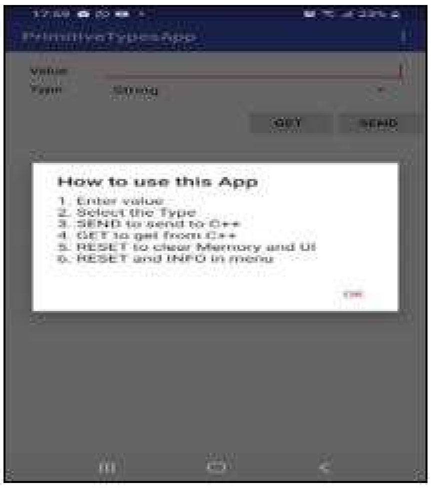
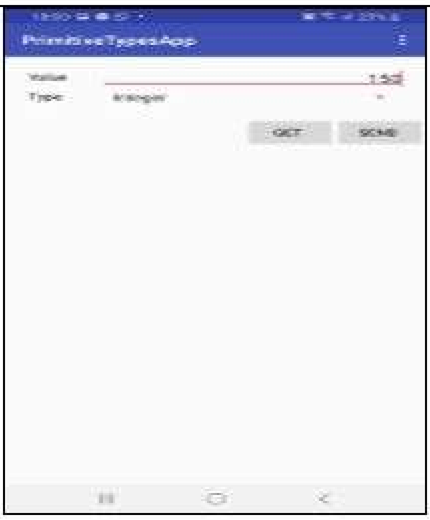
### 4.6.3 JNI API

The JNI family of methods Get<Primitive>ArrayElements creates a pointer to Java array elements. These pointers cannot be garbage collected. For example the GetBooleanArrayElements() creates a pointer to array elements of boolean type. In order to allow the garbage collector to clean the memory, a corresponding Release<Primitive>ArrayElements must be used after a Get<Primitive>ArrayElements. For example, for GetBooleanArrayElements, use is made of ReleaseBooleanArrayElements to allow the garbage collector to clean the memory. The garbage collector is also prevented from memory cleanup when use is made of GetStringUTFChars and GetStringChars. A corresponding use must be made of ReleaseStringUTFChars and ReleaseStringChars to allow the garbage collector to clean the memory (MIT Education 2019; Oracle Documentation 2017.)

## 4.7 Results

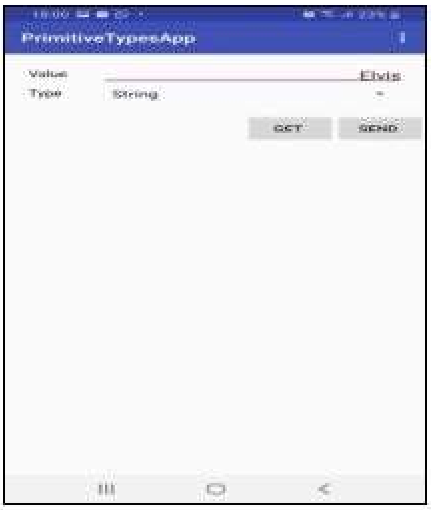
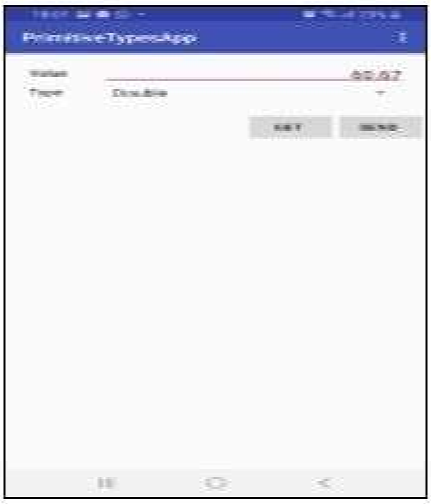
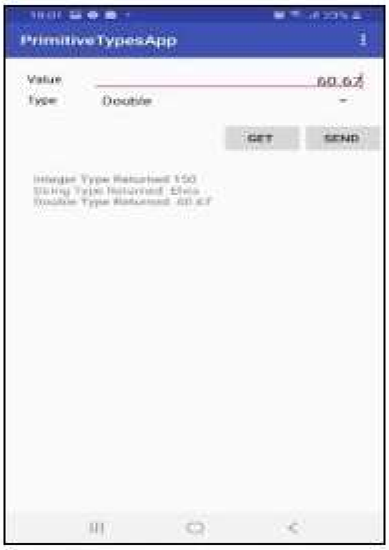
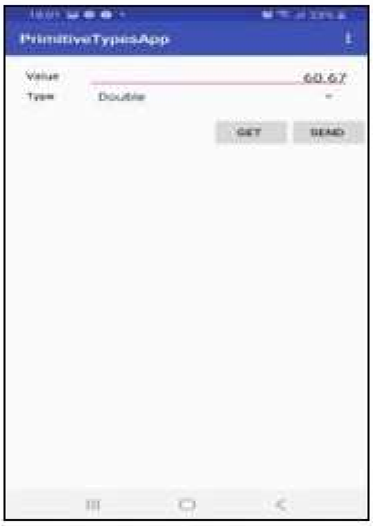
The results for the primitiveType app is presented below (Table 7). The input and output operations are shown. RESET and INFO are in the menu. The initial output when the app is opened is shown with a display of a dialogue. The dialogue gives directives on app usage. Values are then inputted and re-trieved. The RESET button clears the UI after each successive input-output operation. (Gamua 2020.)

TABLE 7. Results for the primitiveType app

| | |
|---|---|
|   PrimitiveTypesApp<br>Value<br>Type<br>String<br>GET<br>SEND<br>OK How to use this App<br>1. Enter value<br>2. Select the Type<br>3. SEND to send to C++<br>4. GET to get from C++<br>5. RESET to clear Memory and UI<br>6. RESET and INFO in menu<br><br>a)initial dialog |   PrimitiveTypesApp<br>Value<br>152<br>Type<br>Integer<br>GET<br>SEND<br><br>b) integer input |

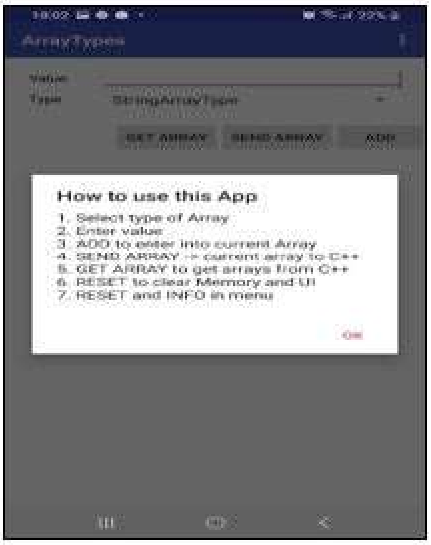
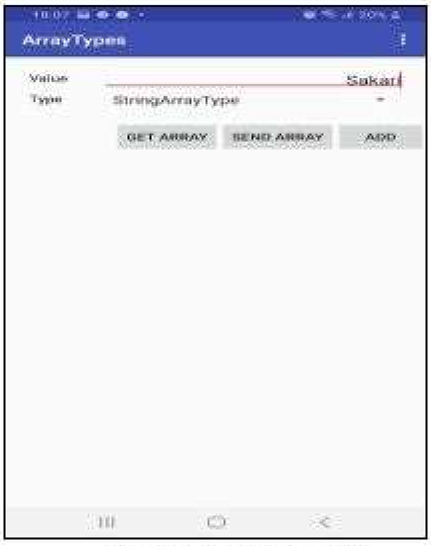
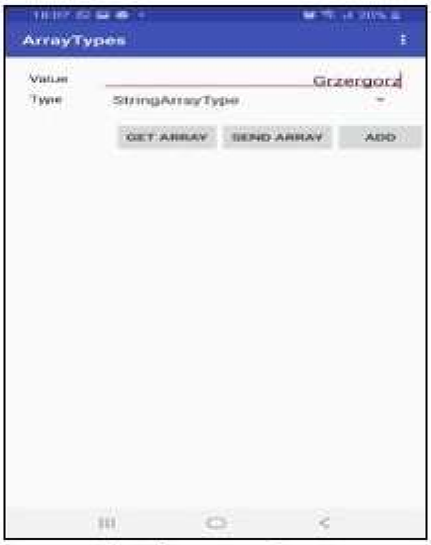
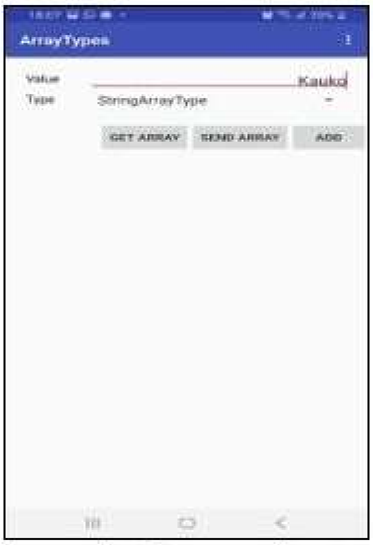
(Continues)

TABLE 7. (Continued)

| | |
|---|---|
|  PrimitiveTypesApp<br>Value Elvis<br>Type String<br> <br>c) string input |  PrimitiveTypesApp<br>Value 60.67<br>Type Double<br> <br>d) double input |
|  PrimitiveTypesApp<br>Value 60.67<br>Type Double<br> <br>Integer Type Returned 150<br>String Type Returned: Elvis<br>Double Type Returned: 60.67<br>e) clicking "GET" |  PrimitiveTypesApp<br>Value 60.67<br>Type Double<br> <br>f) clicking "RESET" |

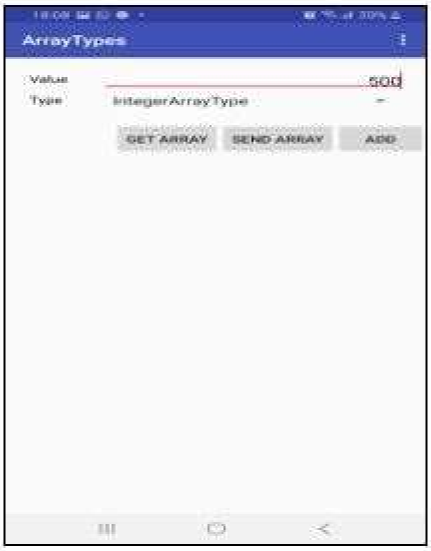
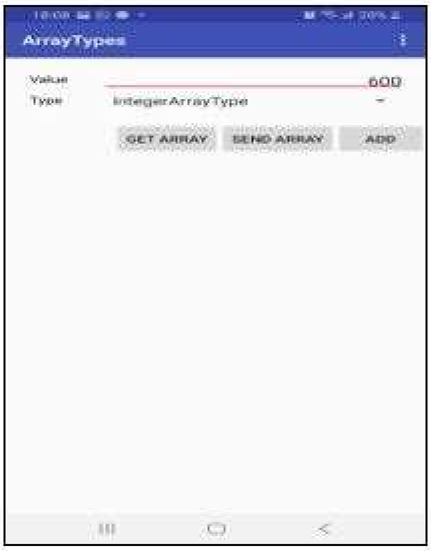
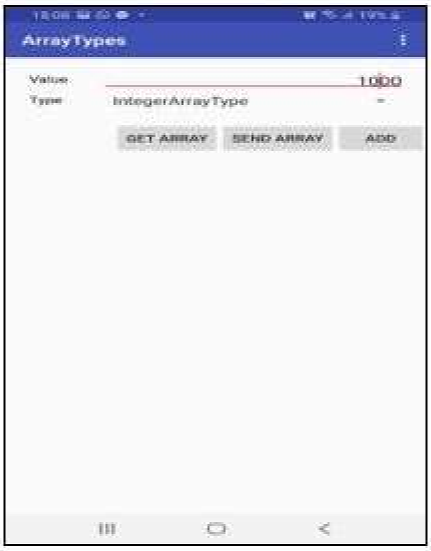
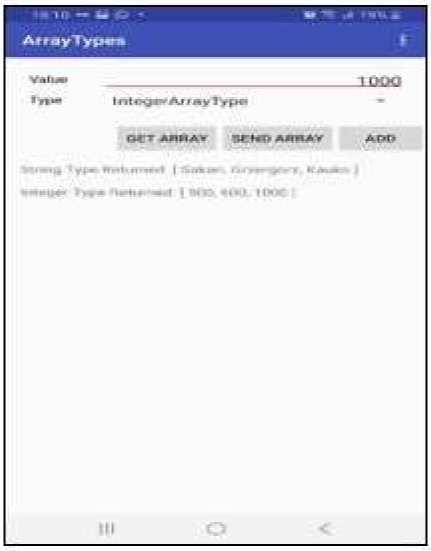
The results for the ArrayType app is presented below (TABLE 8). The input and output operations are shown. In addition there is an "ADD" button to create an array before sending to the native side. The user is informed he has not created an array in case he clicks "SEND ARRAY" before "ADD". The initial output when the app is opened is shown with a display of a dialogue. The dialogue gives directives on app usage. Values are then inputted and retrieved. The RESET button clears the UI after each successive input-output operation. (Gamua 2020.)

TABLE 8. Results for the ArrayType app

| | |
|---|---|
| <br>ArrayTypes<br>Value<br>Type StringArrayType<br>How to use this App<br>1. Select type of Array<br>2. Enter value<br>3. ADD to enter into current Array<br>4. SEND ARRAY -> current array to C++<br>5. GET ARRAY to get arrays from C++<br>6. RESET to clear Memory and UI<br>7. RESET and INFO in menu<br> <br>a) initial dialog | <br>ArrayTypes<br>Value Sakari<br>Type StringArrayType<br> <br>b) string array input |
| <br>ArrayTypes<br>Value Grzergorz<br>Type StringArrayType<br> <br>c) string array input | <br>ArrayTypes<br>Value Kauko<br>Type StringArrayType<br> <br>d) string array input |

(Continues)

TABLE 8. (continued)

| | |
|---|---|
|   ArrayTypes<br>Value<br>500<br>Type<br>IntegerArrayType<br>GET ARRAY<br>SEND ARRAY<br>ADD e) integer array input |   ArrayTypes<br>Value<br>600<br>Type<br>IntegerArrayType<br>GET ARRAY<br>SEND ARRAY<br>ADD f) integer array input |
|   ArrayTypes<br>Value<br>1000<br>Type<br>IntegerArrayType<br>GET ARRAY<br>SEND ARRAY<br>ADD g) integer array input |   ArrayTypes<br>Value<br>1000<br>Type<br>IntegerArrayType<br>GET ARRAY<br>SEND ARRAY<br>ADD String Type Returned: [ Sakari, Grzegorz, Kauko ]<br>Integer Type Returned: [ 500, 600, 1000 ]<br>h) after clicking "GET ARRAY" |

# 5 OBJECT TYPES AND ARRAYS OF OBJECTS

In this chapter manipulating Java objects and accessing Java data members by the JNI API will be explored. An attempt will be made to send and retrieve Java objects to and from the native side. Object arrays will also be sent and retrieved from the native side. To do these operations use shall be made of the various JNI APIs for manipulating Java objects and Java arrays of objects. JNI APIs will further permit us to manipulate Java objects and arrays of objects from the native side. Other APIs explored in this chapter will permit the access of an object's data members from the native side. Memory management will be the final concern in this chapter.

## 5.1 Brief look at relationship between classes

In the ObjectTypes app there are 5 classes vis MainActivity, NativeMsClass, Language, InfoDialog and InputValues classes. InputValues is the C++ class and therefore needs JNI to set and access some of its resources. Worth noting is the fact that the InputValues class is an aggregator class to the Language class. The MainActivity class is also an aggregator class to the NativeMsClass class. The relationship between the classes is presented in Figure 15. (Gamua 2020.)

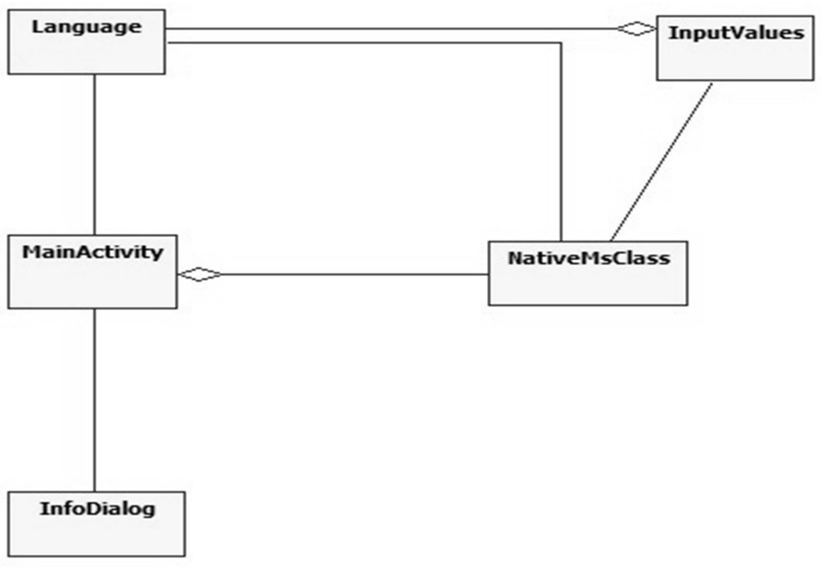


FIGURE 15. Relationship between classes in the ObjectTypes app

In the arrayOfObjects app there are 5 classes vis MainActivity, NativeMsClass, Language, InfoDialog and InputValues. In addition, there is an interface called ComparingListener. Worth noting is the fact that the InputValues class is an aggregator class to the Language class. The NativeMsClass is a compositor class to the ComparingLister interface. The MainActivity class is an aggregator class to the NativeMsClass class. It implements the ComparingLister interface. The relationship between the classes is presented below (Figure 16). (Gamua 2020.)

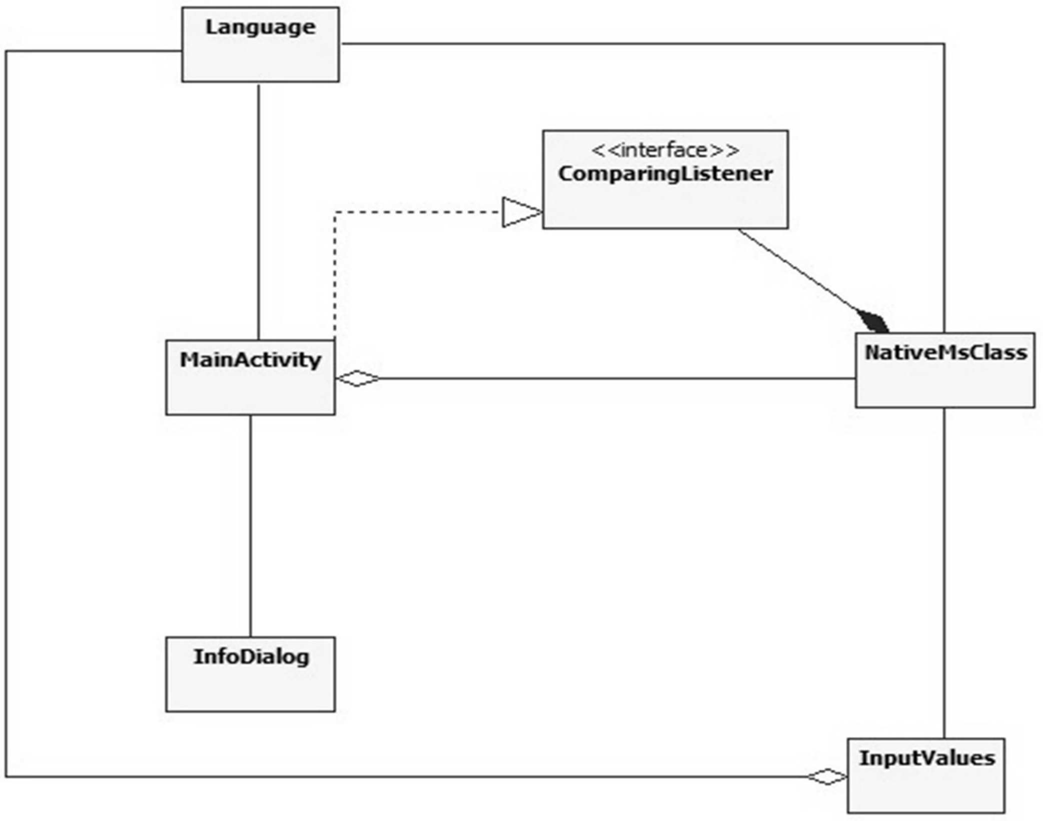
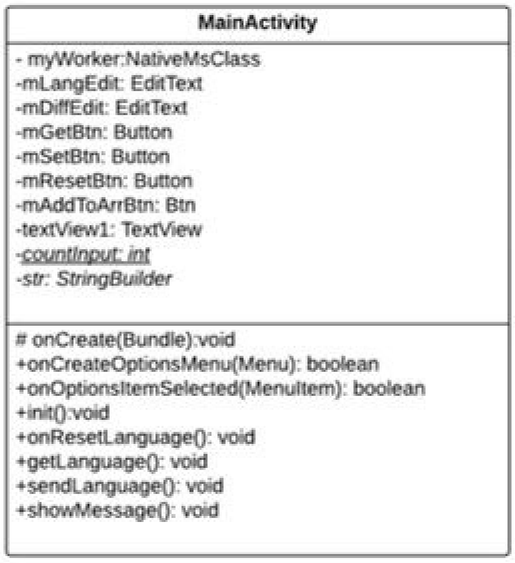
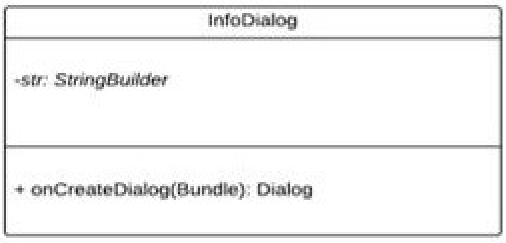
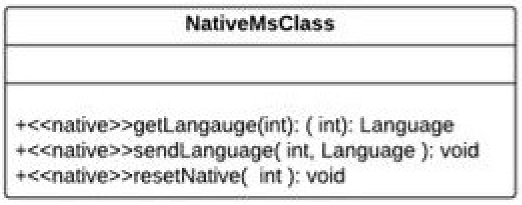


FIGURE 16. Relationship between classes in the arrayOfObject app

## 5.2 Detail look at classes

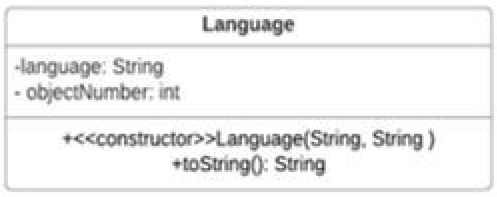
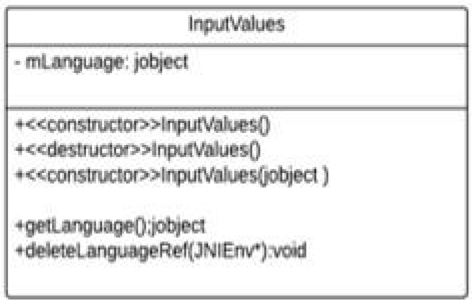
The details of the classes are shown in table 9 and table 10. MainActivity class of the ObjectType app has private members that reference the various Views that are shown on the user interface. The countInput counts the number of inputs entered. MainActivity has MenuItem callback methods that either reset the user interface and free memory on the native side or bring up a user interface dialog. Memory is freed by calling the Java method onResetLanguage which further calls the native method resetNative. The sendLanguage and getLanguage methods are both used to send and retrieve information from the native end by respectively calling the sendLanguage and getLanguage native methods of the NativeMsClass. The Language class has members to count the object number entered, the name of the programming language entered and the difficulty level. The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. The mLanguage variable in the InputValues class is of a type called a jobject. A jobject is a native reference to a Java object. This reference can be sent back to the Java side using the getLanguage method of the NativeMsClass. (Gamua 2020.)

TABLE 9. Classes in the objectType app

| Class | Functions in Brief |
|---|---|
|  MainActivity<br>- myWorker:NativeMsClass<br>-mLangEdit: EditText<br>-mDiffEdit: EditText<br>-mGetBtn: Button<br>-mSetBtn: Button<br>-mResetBtn: Button<br>-mAddToArrBtn: Btn<br>-textView1: TextView<br>-countInput: int<br>-str: StringBuilder<br># onCreate(Bundle):void<br>+onCreateOptionsMenu(Menu): boolean<br>+onOptionsItemSelected(MenuItem): boolean<br>+init():void<br>+onResetLanguage(): void<br>+getLanguage(): void<br>+sendLanguage(): void<br>+showMessage(): void | The MainActivity class of the ObjectType app has private members that reference the various Views that are shown on the user interface. The countInput counts the number of inputs entered. MainActivity also has MenuItem callback methods that either reset the user interface and free memory on the native side or bring up a user interface dialog. Memory is freed by calling the Java method onResetLanguage which further calls the native method resetNative. The sendLanguage and getLanguage methods are both used to send and retrieve values from the native end by respectively calling the sendLanguage and getLanguage native methods of the NativeMsClass |
|  InfoDialog<br>-str: StringBuilder<br>+ onCreateDialog(Bundle): Dialog | The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. The dialog button is found in the menu and can be displayed anytime the user wants. |
|  NativeMsClass<br>+<<native>>getLangauge(int): ( int): Language<br>+<<native>>sendLanguage( int, Language ): void<br>+<<native>>resetNative( int ): void | The NativeMsClass contains a declaration of all the native methods. |

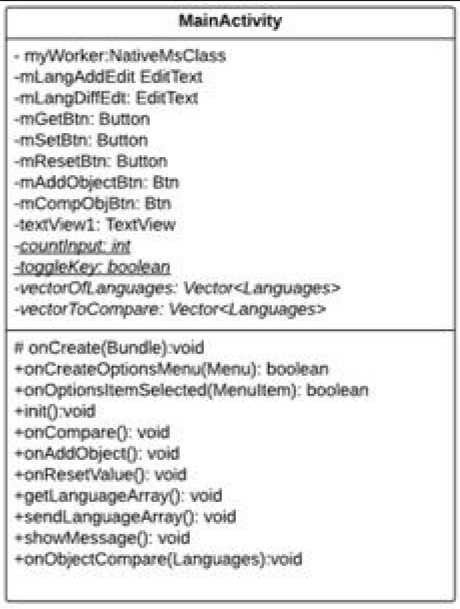
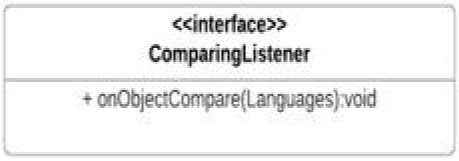
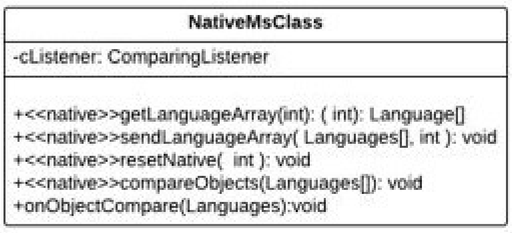
(Continues)

TABLE 9. (Continued)

| | |
|---|---|
|  Language<br>-language: String<br>- objectNumber: int<br>+<<constructor>>Language(String, String )<br>+toString(): String | The Language class has members to count the object number entered, the name of the programming language entered and the difficulty level. |
|  InputValues<br>- mLanguage: jobject<br>+<<constructor>>InputValues()<br>+<<destructor>>InputValues()<br>+<<constructor>>InputValues(jobject )<br>+getLanguage():jobject<br>+deleteLanguageRef(JNIEnv*):void | This class also has constructors to make an object in the native-lib. It has methods to help save the objects passed from java and methods to retrieve those objects from the native side. There are also methods to manage memory |

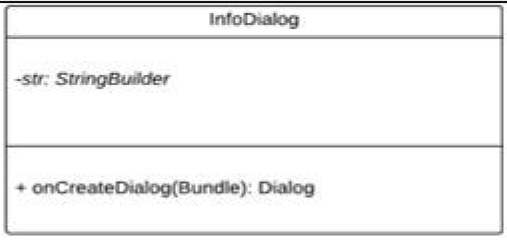
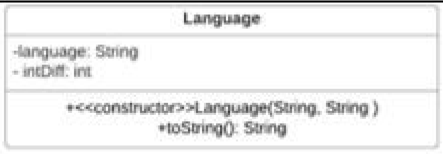
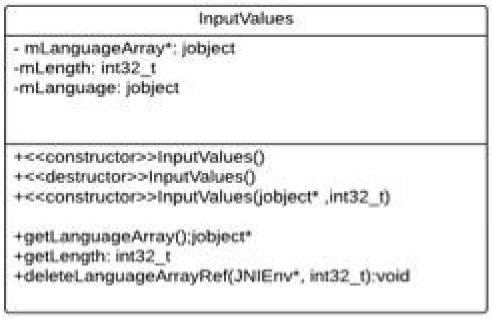
The MainActivity class of the arrayOfObjects app has in addition a toggleKey variable that tracks if values have been entered. There is a vectorOfLanguages Vector variable used to aggregate a list of the objects. This is done using the onAddObect method. The getLanguageArray and sendLanguageArray are used to send and retrieve array references from the native side. The onResetValue resets the native side by deleting all array references and then clears the user interface. onObectCompare aggregates the most difficult language in each array after it has been compared on the native side and sends back via the use of ComparingListener interface. The maximums of each array are printed when the onCompare method is executed. The Language class is similar to the Language class of the objectType app. The NativeMsClass defines the native methods. The native method compareObect is called to compare objects each time an array of object is entered. Additionally, the NativeMsClass' onObjectCompare method overrides the interface method onObjectCompare. The former method of the NativeMsClass is called from the Native-lib when the maximum in each array is computed. This information will then be sent to the MainActivity's onObectCompare method to add to the vectorToCompare Vector variable. (Gamua 2020.)

TABLE 10. Classes in the arrayOfObects app

| Class | Functions in Brief |
|---|---|
|  MainActivity<br>- myWorker:NativeMsClass<br>-mLangAddEdit EditText<br>-mLangDiffEdt: EditText<br>-mGetBtn: Button<br>-mSetBtn: Button<br>-mResetBtn: Button<br>-mAddObjectBtn: Btn<br>-mCompObjBtn: Btn<br>-textView1: TextView<br>-countInput: int<br>-toggleKey: boolean<br>-vectorOfLanguages: Vector<Languages><br>-vectorToCompare: Vector<Languages><br># onCreate(Bundle):void<br>+onCreateOptionsMenu(Menu): boolean<br>+onOptionsItemSelected(MenuItem): boolean<br>+init():void<br>+onCompare(): void<br>+onAddObject(): void<br>+onResetValue(): void<br>+getLanguageArray(): void<br>+sendLanguageArray(): void<br>+showMessage(): void<br>+onObjectCompare(Languages):void | The MainActivity class of the arrayOfObjects app has a toggleKey variable that tracks if values have been entered. There is a vectorOfLanguages Vector variable used to aggregate a list of the objects. This is done using the onAddObject method. The getLanguageArray and sendLanguageArray are used to send and retrieve array references from the native side. The onResetValue resets the native side by deleting all array references and then clears the user interface. onObectCompare aggregates the most difficult language in each array after it has been compared on the native side and sends back via the use of ComparingListener interface. These maximums of each array are printed when the onCompare method is executed. |
|  <<interface>><br>ComparingListener<br>+ onObjectCompare(Languages):void | Interface implemented by MainActivity and NativeMsClass |
|  NativeMsClass<br>-cListener: ComparingListener<br>+<<native>>getLanguageArray(int): ( int): Language[]<br>+<<native>>sendLanguageArray( Languages[], int ): void<br>+<<native>>resetNative(  int ): void<br>+<<native>>compareObjects(Languages[]): void<br>+onObjectCompare(Languages):void | NativeMsClass declares the native methods and overrides the interface method onObjectCompare. The later method of the NativeMsClass is called from the Native-lib when the maximum in each array is computed. The information about the maximum will then be sent to the MainActivity's onObectCompare method to add to the vectorToCompare Vector variable. |

(Continues)

TABLE 10. (Continued)

| | |
|---|---|
|  InfoDialog<br>-str: StringBuilder<br>+ onCreateDialog(Bundle): Dialog | The InfoDialog class has a StringBuilder object that helps to build a dialog to be displayed on the user interface. The dialog button is found in the menu and can be displayed anytime the user wants. |
|  Language<br>-language: String<br>- intDiff: int<br>+<<constructor>>Language(String, String )<br>+toString(): String | The Language class has members to count the object number entered, the name of the programming language entered and the difficulty level. |
|  InputValues<br>- mLanguageArray*: jobject<br>-mLength: int32_t<br>-mLanguage: jobject<br>+<<constructor>>InputValues()<br>+<<destructor>>InputValues()<br>+<<constructor>>InputValues(jobject* ,int32_t)<br>+getLanguageArray():jobject*<br>+getLength: int32_t<br>+deleteLanguageArrayRef(JNIEnv*, int32_t):void | This class also has constructors to make an object of the class in the native-lib. It has methods to help save the array of objects passed from java and methods to retrieve those arrays of objects from the native side. There are also methods to manage memory |

## 5.3 Brief look at user interface

The user interfaces are simple. The user interface of the ObjectTypes app is shown in figure 17. The input is a capture of a single object. There are two input fields both representing the data members of an object as shown in figure 17. Language is the type of programming language and Difficulty is the difficulty level of the language. The SEND LANGUAGE button is used to enter values one at a time. The SEND LANGUAGE Button has an onClickListener attached that calls the sendLanguage method. The user will be informed if he fails to enter a value. The sendLanguage method with the help of myWorker object calls a native method to pass on the object reference to the native libraries. The GET LANGUAGE button displays the objects that have been added to the native C++ class through the NativeMsClass' getLanguage method and the MainActivity's getLanguage method. There is RESET and INFO in the menu. RESET resets the app by deleting the references to all the objects and INFO gives information on app usage. (Gamua 2020.)

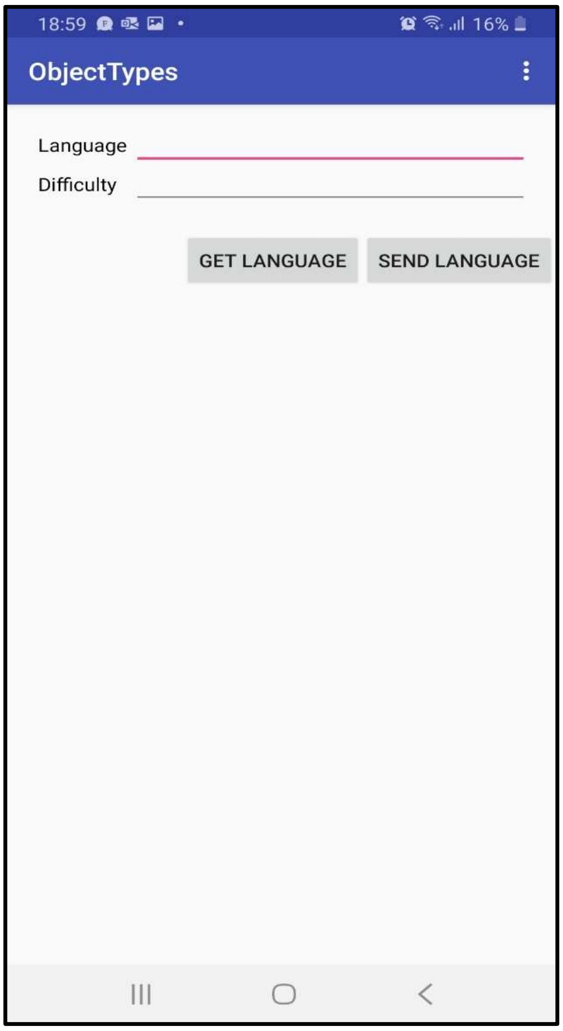


FIGURE 17. User Interface of ObjectType App

The user interface of the arrayOfObjects app is shown in figure 18. On entering a language and its difficulty level, the ADD button is clicked to add this language to the current array of Language objects. This is done by adding to the vector of languages variable vectorOfLanguages. Upon adding a Language object reference, the SEND LANG button becomes enabled immediately but not the COM button which compares the language difficulty level in each array. On clicking SEND LANG, the language array references are sent to the native-lib and hence the native C++ class via the NativeMsClass sendLanguageArray method with the help of the myWorker object.  The compareObjects method also sends the references to the native-lib.cpp library where the objects are compared, and the most difficult languages sent back to be saved in the vectorToCompare vector variable of MainActivity class. Clicking the COM button displays the most difficult objects in an array. The GET LANGS button displays the saved object arrays. There are RESET and INFO items in the menu. RESET resets the app by deleting the references to all the objects and the arrays and INFO gives information on app usage. (Gamua 2020.)

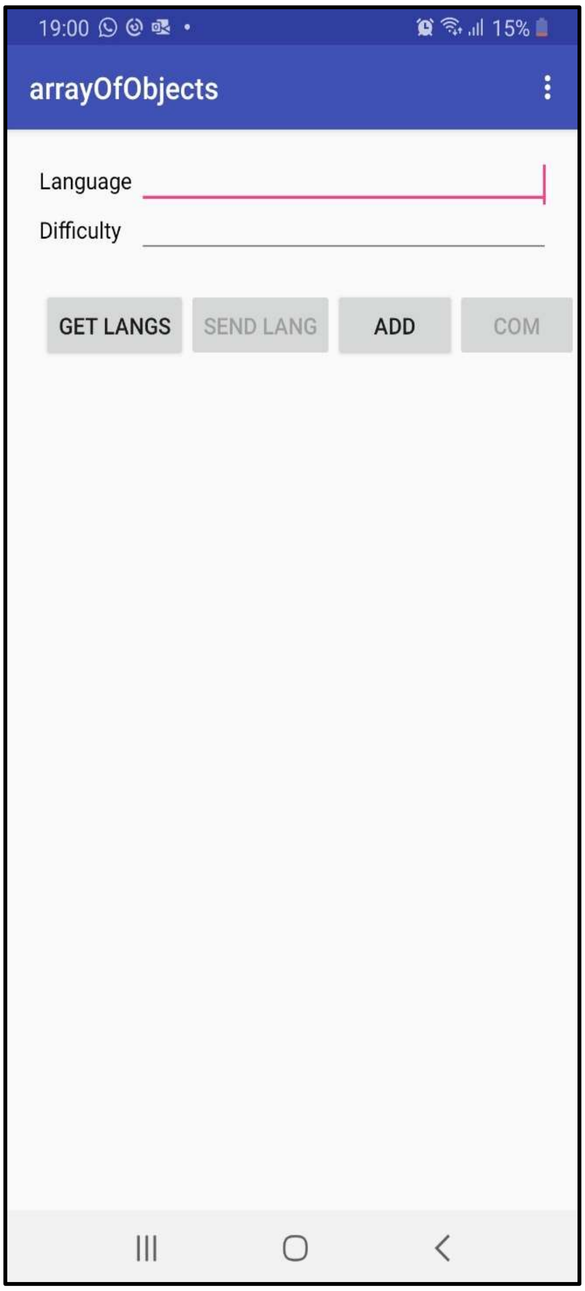
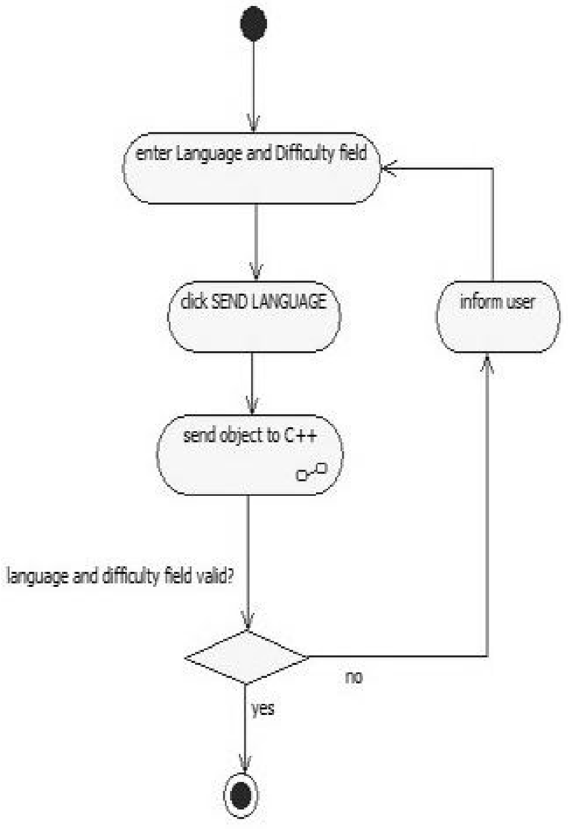
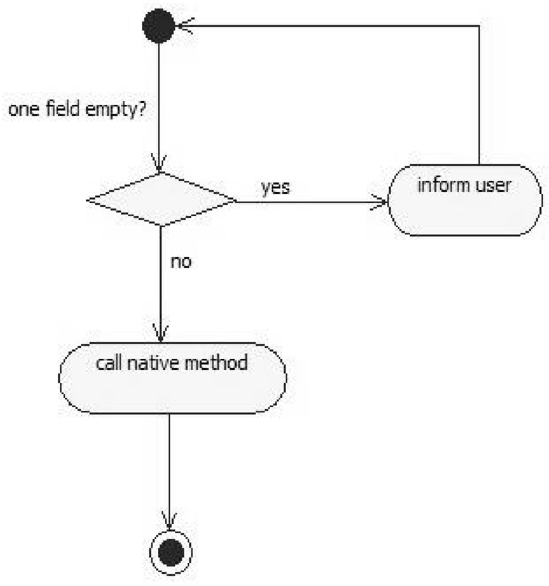
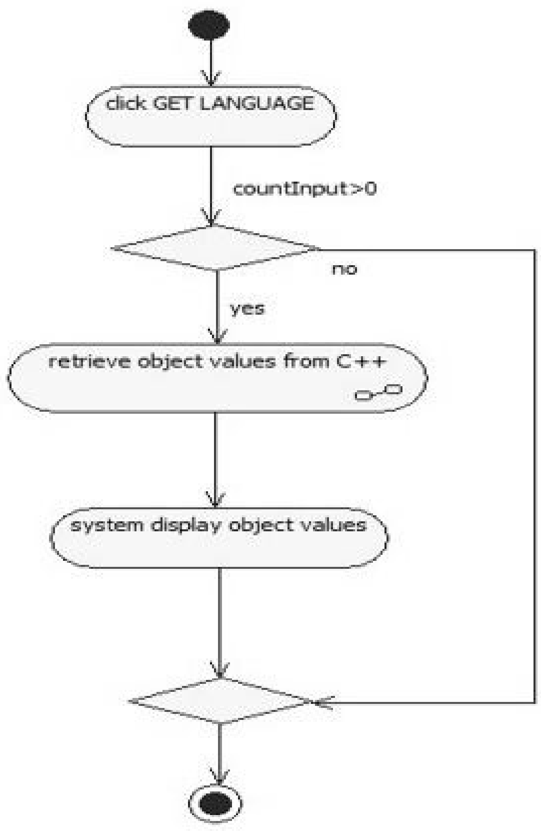
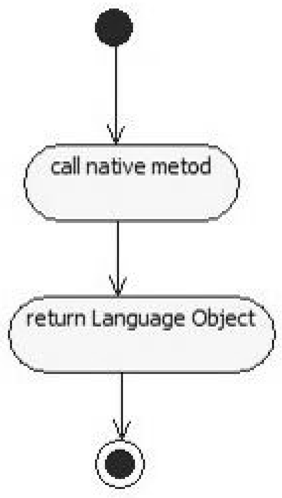


Figure 18. User interface of array of objects app

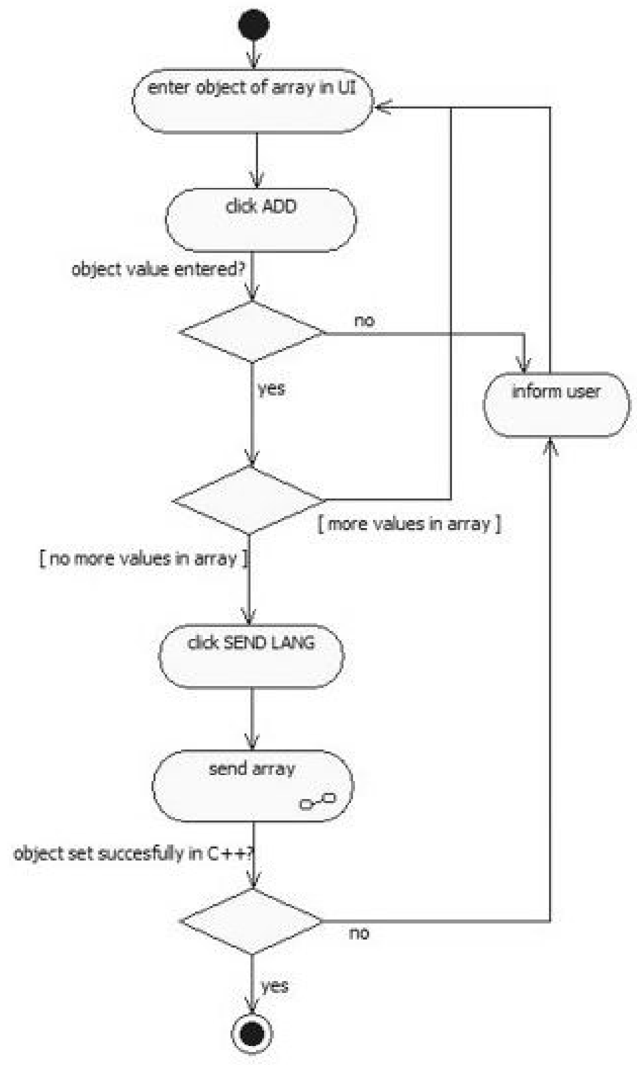
## 5.4 Activity diagrams

The activity diagrams are presented below. All activities have sub activities that are presented alongside. The activity diagrams are drawn for both the send and get use cases. If the fields are not entered rightly and the user tries to send, he will be informed he is performing an illegal operation. If an array has not been created and the user attempts to send, he will also be informed the operation he is attempting to perform is illegal. The tables are given headings for each use case.

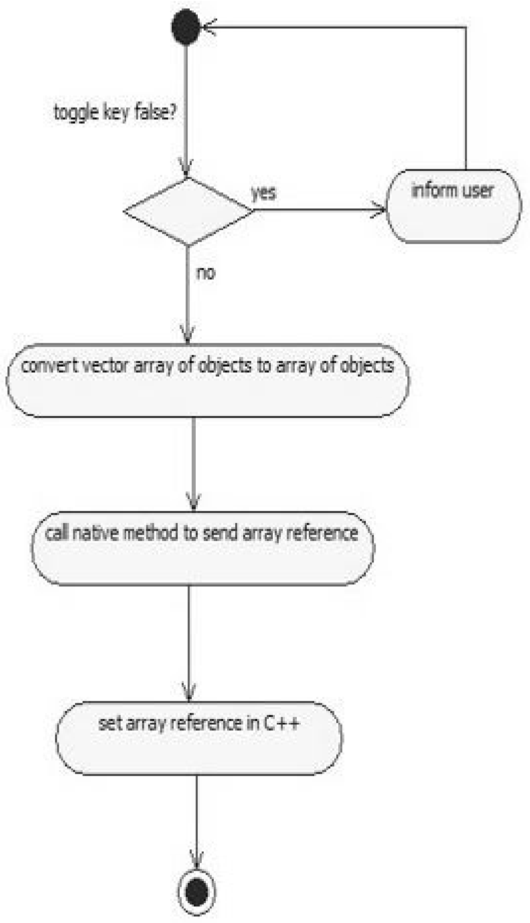
| Activity Diagram to send for the ObjectType app | |
|---|---|
|   enter Language and Difficulty field<br>click SEND LANGUAGE<br>inform user<br>send object to C++<br>language and difficulty field valid?<br>no<br>yes a) Activity Diagram to send an object to native side |   one field empty?<br>yes<br>inform user<br>no<br>call native method b) send object to C++ sub activity |

| Activity Diagram on Clicking GET LANGUAGE for ObjectType app | |
|---|---|
|  click GET LANGUAGE<br>countInput>0<br>no<br>yes<br>retrieve object values from C++<br>system display object values a) Activity Diagram on clicking GET LANGUAGE |  b) retrieve object from C++ sub activity |

Activity Diagram to send for arrayOfObjects app

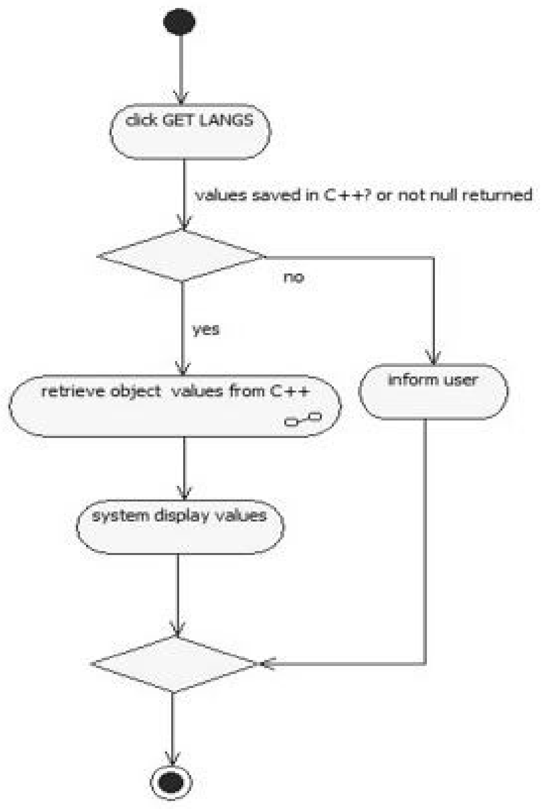


a) Activity Diagram to send to the native side

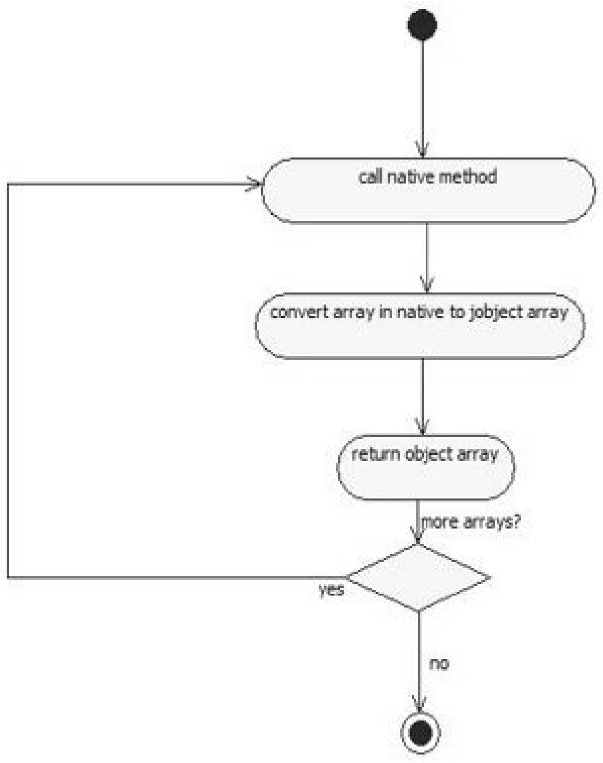

toggle key false?
yes
inform user
no
convert vector array of objects to array of objects
call native method to send array reference
set array reference in C++

b) Send array sub activity

Activity Diagram on Clicking GET LANGS for arrayOfObjects app


click GET LANGS
values saved in C++? or not null returned
no
yes
retrieve object values from C++
inform user
system display values

a) Activity Diagram on clicking GET LANGUAGE



b) retrieve object values from C++ sub activity

## 5.5 Forming the method name and signature of a Java method

The understanding of this section is vital. Understanding of programs will be incomplete without mentioning how a Java method can be called from C/C++. Reference should be made to code usage in the native-libs of either the arrayOfObjects app or the NativeThreads app of chapter 6. The JNI operations that permit calling Java methods belong either Call<type>Method Routines, Call<type>MethodA Routines or Call<type>methodV Routines. Methods from any of these families are used to call Java instance methods. (MIT Education 2019; Oracle Documentation 2017.)

In order to use any methods in the routines mentioned above a pointer to the GetMethodID method must be obtained. GetMethodID has as parameters a JREF index for the class in question, the name of the method as a string and the internal type signature for that particular method. Constructor names are specified as <init>. The JREF index points to the class where the method belongs. This index is got by either using the FindClass or GetObjectClass API. To look up the particular method being called, the methods have type signatures which are treated in the next paragraph. In the native methods use is made of the methods in the family of operations above to call the Java method. The parameters to be passed to the family of operations are the JNI jobject representation of the object that called the native method in the first place, the methodID which is the returned ID upon use of the GetMethodID and the actual arguments of the Java method in question.

For the Java Virtual Machine to look up the particular method being called, the methods have type signatures. Type signatures ensure that the specific method is called. They include the input and return type of the method in question. The onObjectCompare method in the arrayOfObects app's NativeMsClass has a Languages class variable as input parameter and void as return type. The method's signature is Lcom/example/gamuatachu/arrayofobjects/Languages;)V where com/example/gamuatachu/arrayofobjects represents the package name (Gamua 2020). The signatures are shown below (TABLE 11). (MIT Education 2019; Oracle Documentation 2017.)

TABLE 11. JVM type signatures

| Signature | Java Type |
|---|---|
| Z | Boolean |
| B | Byte |
| C | Char |
| S | Short |
| I | Int |
| J | Long |
| F | Float |
| D | Double |
| L fully-qualified-class ; | fully-qualified-class |
| [ type | type[] |
| ( arg-types ) ret-type | method type |

## 5.6 Understanding the programs

After entering a value in the Language field and the Difficulty field in the objectTypes app, the SEND LANGUAGE button is pressed to enter the values. The SEND LANGUAGE button has an onClickListener attached that calls the sendLanguage method. If both fields are entered rightly, the myWorker object calls the NativeMsClass' native sendLanguage method and this method calls its corresponding method in the native-lib.cpp library. If the fields are entered wrongly a message is displayed indicating wrong entry of input type. The countInput variable is incremented each time an entry is successfully added. In the corresponding sendLanguage method in the native-lib.cpp a new global variable is created using the NewGlobalRef JNI API. This prevents the reference from being garbaged collected and also permits it to be seen outside its scope. A new object is created with this new globally created variable. The mLanguage member of this object cannot be NULL since the reference it refers to has been prevented from being garbage collected by the use of NewGlobalRef. If there is no need for this reference to be kept global, the JNI API DeleteGlobalRef can be used as can be seen from the deleteLanguageRef method of objectType.cpp. The GET LANGUAGE button activates the getLanguage method. The objects saved are then appended to the MainActivity's StringBuilder str by calling the NativeMsClass' getLanguage method with the help of the myWorker object. The stringBuilder is then displayed on the UI using textView1. (Gamua 2020.)

In the arrayOfObjects app the ADD button permits objects to be entered into the current array. The toggleKey ensures at least one object is entered into the array. The ADD button has an onClickListener attached. This listener activates the onAdd method to add the entered Language object to the vector of languages variable vectorOfLanguages. The SEND LANG button becomes enabled after an object is entered to an array but the COM button which compares the language difficulty level in each array remains disabled since it should only become enabled if there are objects that have been sent to the native-lib.cpp and hence the native C++ class. It is only when objects are present in the native side that comparison of objects in an array can be done. The Boolean variable toggleKey becomes true when an array is created. This variable checks if an object has been added to an array before it is sent to the native libraries. If toggleKey is false, a message will be generated upon a click/press on SEND LANG letting you know there are no arrays created. (Gamua 2020.)

On clicking SEND LANG, the toggleKey variable is checked. If it is true, then a Language object has been added to an array. The vector of Language objects are then converted to an array of Language objects whose reference is passed to the Language array variable lan. The vector of Language variable is cleared. The myWorker object calls on the native method sendLanguageArray to pass on this reference of array objects to C++. The same reference is also sent via the compareObjects method to the native-lib.cpp library where the objects in each array are compared. The COM button becomes enabled. (Gamua 2020.)

On the native side of the arrayOfObjects app two methods are called: Firstly, the sendLanguageArray method is called. In this method, the length of the array is determined by the GetArrayLength API. A pointer of type jobject is created using the length of the determined array. This pointer reference will be incremented as an array to hold a reference to the successive references of Language objects. The successive Language references in an array are gotten using the GetObjectArrayElement. A new global reference variable is created for each reference. A new InputValue object reference is finally created with the reference of references (or the reference of the first array member) and the length of the array is passed to an IncomingValue array object. Secondly, the native-lib's method compareObjects is called. This method allows for comparison of objects within an array each time an array of objects is entered by clicking the SEND LANGS button. In this method, the length of the array is first determined, the first object or reference within the array is also determined and saved as a jobject variable. In the for loop that proceeds a NULL reference is assigned to a second Language object variable. This variable will hold the reference of the second object in the array. Two jchar variables, languageValue1 and languageValue2 are declared. These can also be declared as jints. In order to access the Java int fields of Language objects, the field ID of the int field member intDiff has to be determined. This is done using the GetFieldID and the GetIntField APIs. In JNI each data member of a Java class is regarded as a field and is associated with an ID. This ID permits the getting (access) or the setting (changing) of data members of objects. The GetFieldID gets the field ID of a class member (for example, its usage in this case returns the ID of intDiff) while the GetIntField gets the particular int field of an object. The if condition that follows guarantees that the object is not NULL. Inside the if statement, the int field of the second Language object in the array is extracted. This is then compared with the first to determine which is larger. This process to filter the larger value continues for all the elements in the Language array. The onObjectCompare method in the NativeMsClass is finally called to pass on the reference of the Language object with the largest difficulty. In order to call the onObjectCompare method, use is made of one of JNI's API CallVoidMethod methods. This method takes as parameters the environment variable, the method ID of the method to be called and whose value had been determined and cached in the JNI_onLoad method and the actual parameters to be passed to the called method. In the NativeMsClass, the onObjectCompare method passes this object to the MainActivity's onObjectCompare where it is added to the vectorToCompare variable. (Gamua 2020.)

The GET LANGS button permits the recovery of saved arrays. myWorker.getLanguageArray(i) is called as many times as there are Language arrays. In the corresponding method on the native side, the incomingValue reference object for the jObjectNum being tested is checked to see if it returns a jobject reference to a Language array. If so, a jobjectArray reference is created with the help of the JNI API

NewObjectArray. One of the parameters of this API is the variable LanguagesClass which is a reference object to the Language class. This reference had been cached as a global variable in the JNI_onLoad method and in JNI, its type is a jclass. The jobjectArray reference is then set to reference the different array object elements using the SetObjectArrayElement API. The jobjectArray is finally returned, appended to the StringBuilder object str and displayed with the help of a TextView object when all arrays have been appended. (Gamua 2020.)

After clicking the SEND LANG button, the COM button becomes activated permitting the printing on the UI of the most difficult Language object within each array. The most difficult Language objects are heretofore saved in the vectorToCompare vector variable. Clicking/pressing the COM button calls the onCompare() method which converts vectorToCompare to a Language array, appends a string version of each object in the array to the StringBuilder object str and finally prints on the UI. (Gamua 2020.)

## 5.7 Memory management

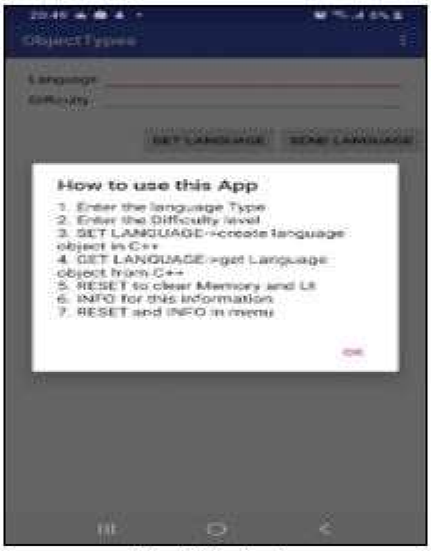
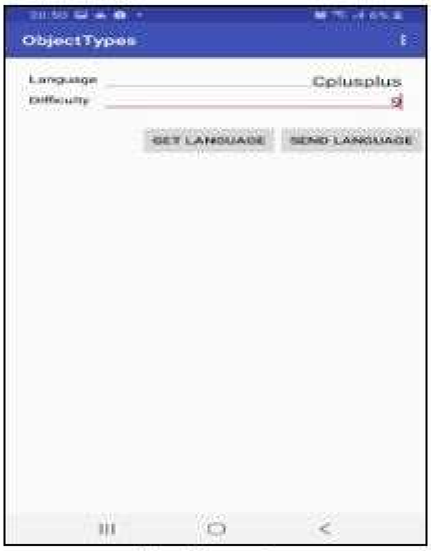
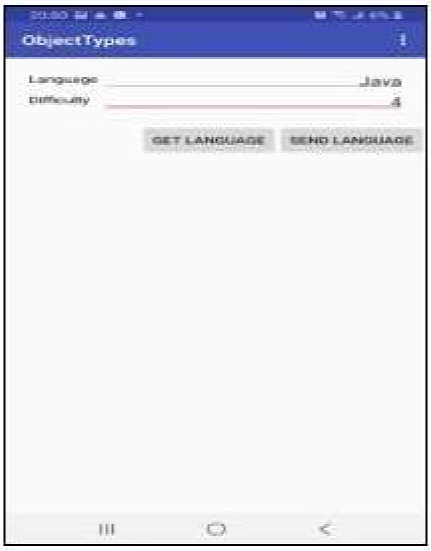
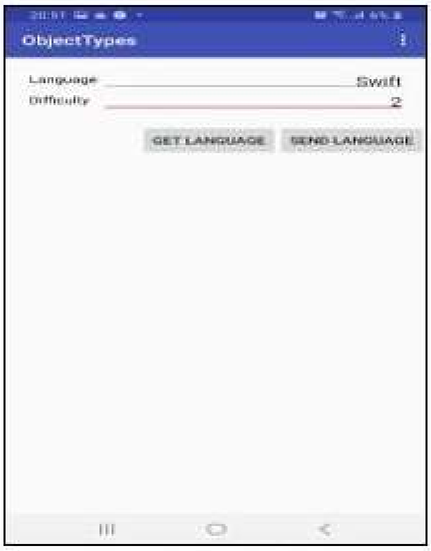
If the RESET menu item of the ObjectTypes app is clicked, the attached onClickListener activates the onResetLanguage method. This method verifies that there is an inputted Language object and calls the native method resetNative via the myWorker object. This call is made as many times as the number Language objects inputted. For each of those times, there is a verification in the native-lib's resetNative method that an incomingValue object reference returns a jobject Language reference. If yes, it deletes the Language reference and then deletes the object. (Gamua 2020.)

If the RESET menu item of the arrayOfObjects is clicked, the attached onClickListener activates the onResetValue method. This method verifies that there is an inputted array of Language objects and calls the NativeMsClass' native resetNative method via the myWorker object. This call is made as many times as there were arrays entered. For each of those times, there is a verification in the native-lib.cpp's method resetNative that an incomingValue object reference returns a jobject Language pointer reference. If yes, a call is made to the deleteLanguageArrayRef method to delete the object references that were pointed to by jobject pointer reference and in this same deleteLanguageArrayRef method the jobject pointer reference is deleted. After deleting these references, the incomingValue's object reference is deleted in the native-lib.cpp's resetNative method. (Gamua 2020.)

## 5.8 Results

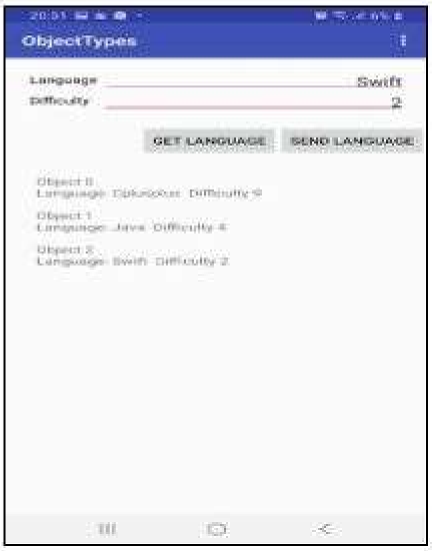
The results for the ObjectType app is presented below (TABLE 12). The input and output operations are shown. RESET and INFO are in the menu. The initial output when the app is opened is shown with a display of a dialogue. The dialogue gives directives on app usage. Values are then inputted and retrieved. The RESET button clears the UI after each successive input-output operation.

TABLE 12. Results for ObjectType app

| | |
|---|---|
|  ObjectTypes Language Difficulty GET LANGUAGE SEND LANGUAGE How to use this App 1. Enter the language Type 2. Enter the Difficulty level 3. SET LANGUAGE->create language object in C++ 4. GET LANGUAGE->get Language object from C++ 5. RESET to clear Memory and UI 6. INFO for this information 7. RESET and INFO in menu OK a)initial dialog |  ObjectTypes Language Cplusplus Difficulty 9 GET LANGUAGE SEND LANGUAGE b) object input |
|  ObjectTypes Language Java Difficulty 4 GET LANGUAGE SEND LANGUAGE c) object input |  ObjectTypes Language Swift Difficulty 2 GET LANGUAGE SEND LANGUAGE d) object input |

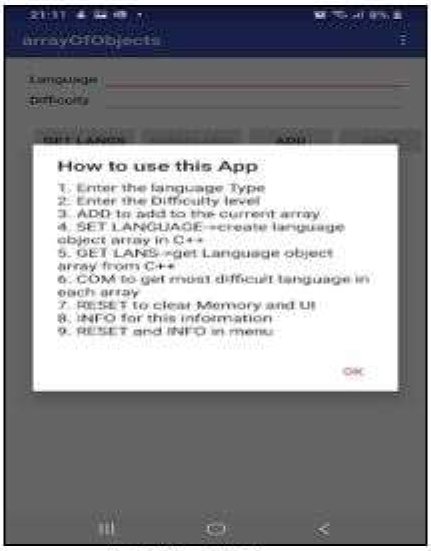
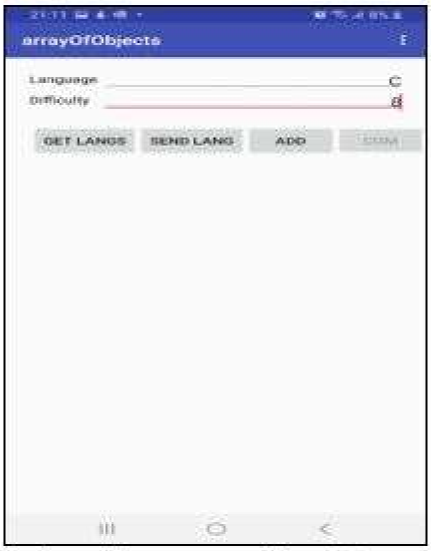
(Continues)

TABLE 12. (Continued)

| | |
|---|---|
|  ObjectTypes<br>Language Swift<br>Difficulty 2<br>GET LANGUAGE SEND LANGUAGE<br>Object 0<br>Language: Cplusplus Difficulty 9<br>Object 1<br>Language: Java Difficulty 4<br>Object 2<br>Language: Swift Difficulty 2<br>e) after pressing GET LANGUAGE | |

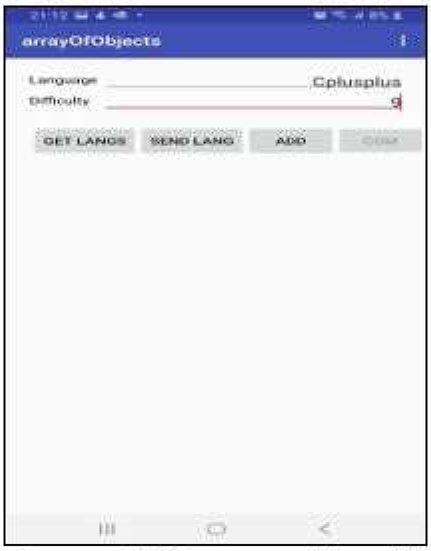
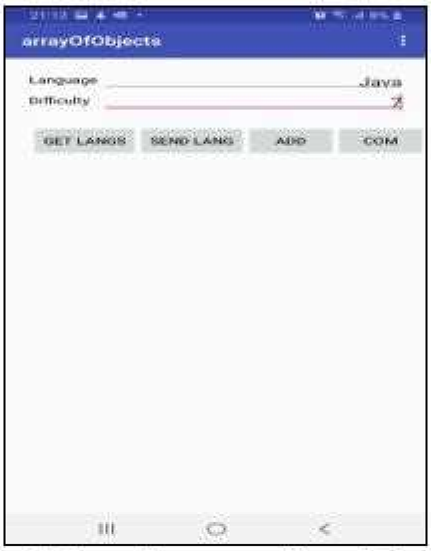
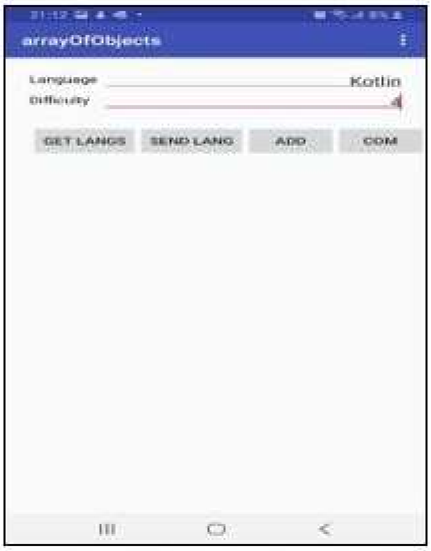
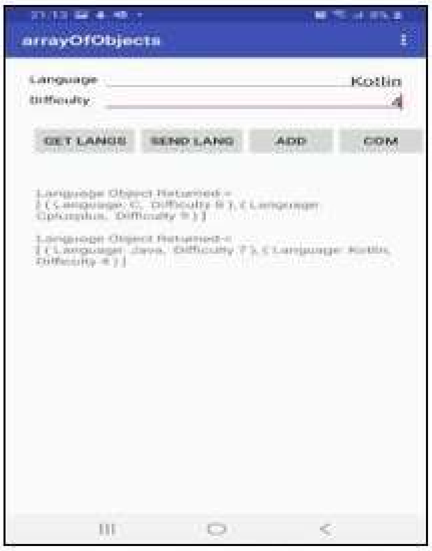
The results for the ArrayOfObjects app is presented below (TABLE 13). The input and output operations are shown. In addition there is an "ADD" button to create an array before sending to the native side and a COM Button to compare the difficulty of Language objects in an array. The initial output when the app is opened is shown with a display of a dialogue. The dialogue gives directives on app usage. The RESET button clears the UI after each successive input-output operation.

TABLE 13. Results for arrayOfObjects app

| | |
|---|---|
|  arrayOfObjects<br>Language<br>Difficulty<br>How to use this App<br>1. Enter the language Type<br>2. Enter the Difficulty level<br>3. ADD to add to the current array<br>4. SET LANGUAGE->create language object array in C++<br>5. GET LANGS->get Language object array from C++<br>6. COM to get most difficult language in each array<br>7. RESET to clear Memory and UI<br>8. INFO for this information<br>9. RESET and INFO in menu<br>OK<br>a) initial dialog |  arrayOfObjects<br>Language C<br>Difficulty 8<br>GET LANGS SEND LANG ADD COM<br>b) first object entered into 1st array |

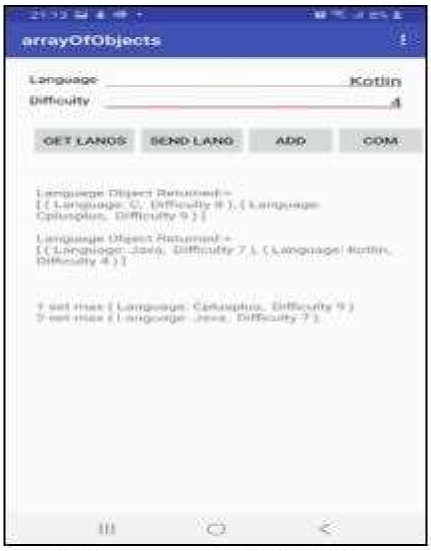
(Continues)

TABLE 13. (Continued)

| | |
|---|---|
|   arrayOfObjects<br>Language<br>Cplusplus<br>Difficulty<br>9<br>GET LANGS<br>SEND LANG<br>ADD<br>COM c) second object entered into 1st array |   arrayOfObjects<br>Language<br>Java<br>Difficulty<br>7<br>GET LANGS<br>SEND LANG<br>ADD<br>COM d) first object entered into 2nd array |
|   arrayOfObjects<br>Language<br>Kotlin<br>Difficulty<br>4<br>GET LANGS<br>SEND LANG<br>ADD<br>COM e) second object entered into 2nd array |   arrayOfObjects<br>Language<br>Kotlin<br>Difficulty<br>4<br>GET LANGS<br>SEND LANG<br>ADD<br>COM Language Object Returned = [ { Language: C, Difficulty 8 }, { Language: Cplusplus, Difficulty 9 } ] Language Object Returned= [ { Language: Java, Difficulty 7 }, { Language: Kotlin, Difficulty 4 } ] f) after pressing "GET LANGS" |

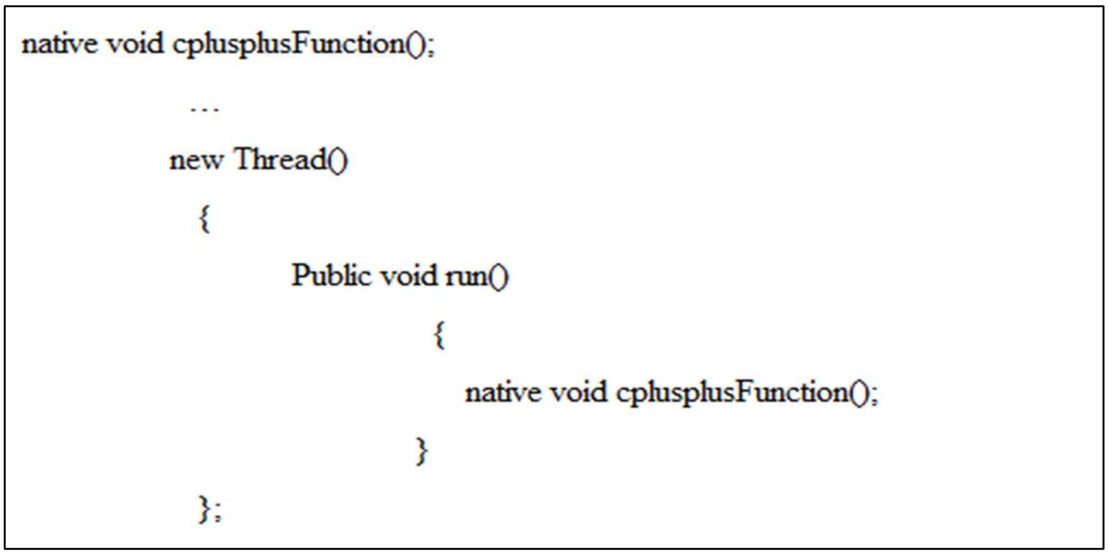
(Continues)

TABLE 13. (Continued)

| | |
|---|---|
|  arrayOfObjects<br>Language Kotlin<br>Difficulty 4<br>GET LANGS SEND LANG ADD COM<br>Language Object Returned=<br>[ ( Language: C, Difficulty 8 ), ( Language: Cplusplus, Difficulty 9 ) ]<br>Language Object Returned=<br>[ ( Language: Java, Difficulty 7 ), ( Language: Kotlin, Difficulty 4 ) ]<br>1 set max ( Language: Cplusplus, Difficulty 9 )<br>2 set max ( Language: Java, Difficulty 7 )<br>g) after pressing "COM" | |

# 6 THREADS

Threads are particularly important in programming. In native development threads can only be explicitly created. When support for C++ is included in a project, the Java Native Interface (JNI) does not create a new thread. The Java thread that the Java method executes in, is the same thread that the native method executes in. In case there is need for a new thread, it must be explicitly created. Creation of a thread can be done in two ways. The first method is by wrapping the C++ method that needs to be executed in a thread in a created Java thread. This is shown below in code snippet 6. (MIT Education 2019; StackOverflow 2017.)



```
native void cplusplusFunction();
            ...
            new Thread()
              {
                        Public void run()
                                  {
                                     native void cplusplusFunction();
                                  }
              };
```

CODE SNIPPET 6. Native method wrapped and executed in created java thread

The second variant of native thread creation is to create the thread on the native side using the POSIX PThread API. In this case, a new thread is created but it needs to be attached to the Java Virtual Machine and the interface pointer can then be extracted in case it is needed. This is the type that will be used in this project. Note should be made that the interface pointer cannot be shared between threads but the JavaVM can be shared. In this chapter a look is going to be taken at how a native thread can be created (MIT Education 2019; Emory College of Arts and Sciences 2020; IBM Documentation 2020.)

## 6.1 User interface

The user interface is simple and shown in figure 19. There is an EditText that allows the user to enter numbers. This number is the maximum number primes can be printed up to. After inputting the number, the user clicks on the START button and the system calculates the primes up to that number and presents on the user interface. There is no menu containing RESET and INFO as the apps treated in chapters 4 and 5. There is however a RESET button on the main UI. The RESET button clears the main UI.

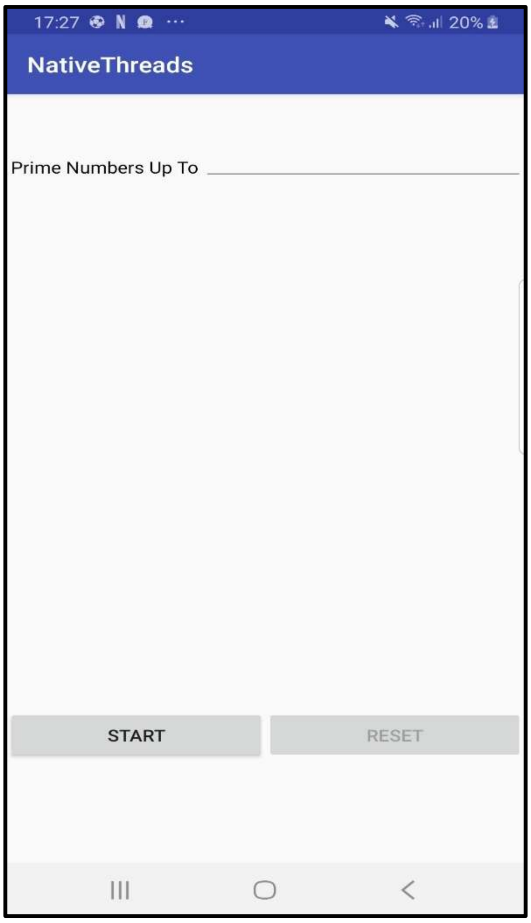


FIGURE 19. User interface for NativeThread app

## 6.2 Simple class diagram relationship

The class diagram is shown in figure 20. The main points to note of here is that the NativeMsClass declares the native methods and implements the NativeListener interface methods. In the NativeListener interface methods are declared and these methods will be implemented in the MainActivity class. The NativeMsClass is a compositor class to the NativeListener interface. There is a ThreadVariable struct on the native side that is used to hold some variables. These variables are used by the thread and the thread methods.(Gamua 2020.)

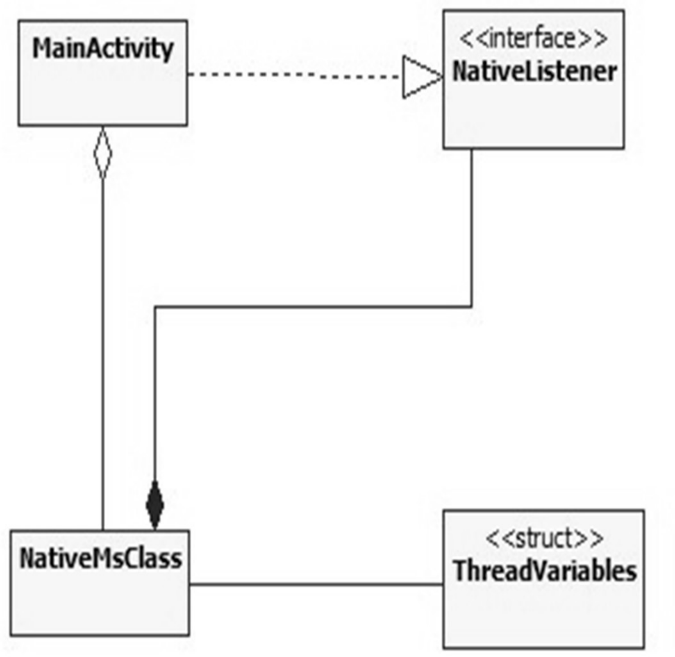
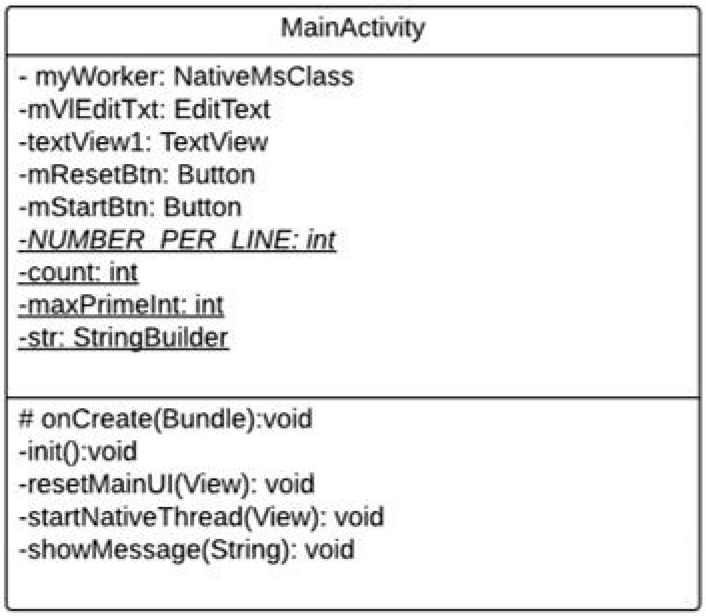
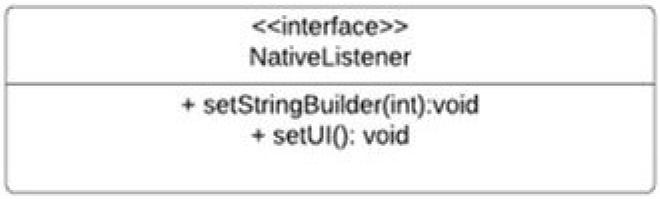


FIGURE 20. Simple Class Diagram Relationship

## 6.3 Detail look at classes

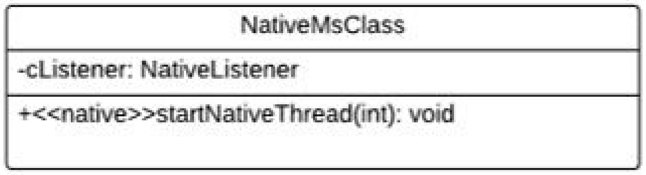
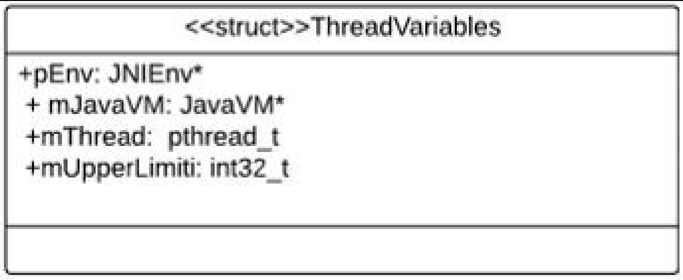
The class diagrams are shown in table 14. MainActivity has private data members that reference the various Views that are shown on the user interface. The NUMBER PER LINE defines the number of primes to be outputted per line. The count variable keeps track of the total number of primes outputted. maxPrimeInt saves the highest number the computation will be done up to. MainActivity also has Java methods that are used to call native methods from the NativeMsClass. The resetMainUI method clears the user interface. There is no need to free memory in the native side when the user interface is cleared as the ThreadVariable struct's pointer object defined in the runThread method of the native-lib is automatically deleted after the thread created in the native-lib is detached. The startNativeThread starts the native method. The MainActivity class also implements NativeListener's methods. The NativeListener interface is implemented by the NativeMsClass. NativeMsClass also has a declaration of the native methods, a method to send information back to the MainActivity in order to build a string that holds all the primes and a method that would be called to output on the user interface. The ThreadVariables struct has members to hold the JNI interface pointer, a pointer to the Java Virtual Machine, a variable of the pthread_t class to hold the ID of the created thread and a variable to hold the upper computational limit which is entered by the user. Various methods exist in the native-lib to start the thread, run it and compute the prime. These methods are detailly explained in the section 6.4. (Gamua 2020.)

TABLE 14. Details of classes NativeThread app

| Class | Functions in Brief |
|---|---|
|  MainActivity; - myWorker: NativeMsClass; -mVlEditTxt: EditText; -textView1: TextView; -mResetBtn: Button; -mStartBtn: Button; -NUMBER_PER_LINE: int; -count: int; -maxPrimeInt: int; -str: StringBuilder; # onCreate(Bundle):void; -init():void; -resetMainUI(View): void; -startNativeThread(View): void; -showMessage(String): void | MainActivity has private data members that reference the various Views that are shown on the user interface. The NUMBER PER LINE defines the number of primes to be outputted per line. The count variable keeps track of the total number of primes outputted. maxPrimeInt saves the highest number the computation will be done up to. MainActivity also has Java methods that call native methods from the NativeMsClass. The resetMainUI method clears the user interface. |
|  <<interface>> NativeListener; + setStringBuilder(int):void; + setUI(): void | Interface implemented by MainActivity and NativeMsClass. |

(Continues)

TABLE 14. (continued)

| | |
|---|---|
|  NativeMsClass<br>-cListener: NativeListener<br>+<<native>>startNativeThread(int): void | NativeMsClass has a declaration of the native methods, an over-rided method to send information back to the MainActivity in order to build a string that holds all the primes and another overrided method that will be called from native-lib to output. |
|  <<struct>>ThreadVariables<br>+pEnv: JNIEnv*<br>+ mJavaVM: JavaVM*<br>+mThread: pthread_t<br>+mUpperLimiti: int32_t | The ThreadVariables struct has members to hold the JNI interface pointer, a pointer to the Java Virtual Machine, a variable of the pthread_t class to hold the ID of the created thread and a variable to hold the upper computational limit which is entered by the user. |

## 6.4 Program description

Upon launching the application, the START button is enabled while the RESET button is disabled. There are onClickListeners attached to the START and RESET button. Upon entering an integer type greater than or equal to 2 and clicking the START button the onClickListener attached to the START button is called. This executes the method startNativeThread which calls the startNativeThread of the NativeM-sClass with the help of the myWorker object. In the startNativeThread method of MainActivity, the START button is disabled. (Gamua 2020.)

### 6.4.1 In native-lib.cpp

The startNativeThread creates a global variable out of the current object. This global variable is saved in the myLock jobject. This will be used to synchronise or pass a monitor to in the critical section when the thread is started. The native thread is started with the method startThread. The JavaVM object must be passed to this method as this will be used to retrieve the interface pointer. In the startThread(JavaVM, nInteger) method, the JavaVM object is saved in one of the members of the ThreadVariable struct. The number which the primes will be printed up to is also saved into one of the members of the ThreadVariable struct. The lAttributes reference of the pthread_attr_t struct reference contains the properties of the new thread. The thread is then started using the syntax below (CODE SNIPPET 7). (MIT Education 2019; The Fossies Software Archive 2020; Gamua 2020.)

```
int pthread_create(pthread_t *threadID,
        const  pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg);
```

CODE SNIPPET 7. Syntax to start native thread

The properties of the new thread are contained in the variable attr. The threadID is the identifier for the new thread. The start_routine is the name of the new function that the new thread will execute. In the case of this project it is runThread. The start_routine is passed a parameter of type void *. The return type of start_routine is void*. The argument that will be passed to the start_routine function is arg. In the case of this project the void pointer passed must be casted back to a ThreadVariable pointer. If the thread is created successfully, the pthread_create() method returns 0. (Emory College of Arts and Sciences 2020.)

```
void *  start_routine( void * arg )
   {
      ....
   }
```

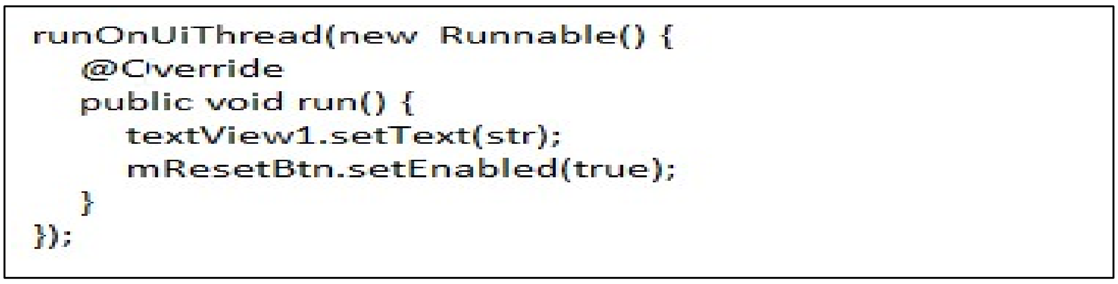
CODE SNIPPET 8. start_routine of native thread

In the runThread method found in native-lib.cpp the void pointer is casted back to a ThreadVariable pointer. An object of the JavaVMAttachArgs structure is used to further input information about the thread. If the thread is successfully created, AttachCurrentThread function returns JNI_OK and the JNI interface pointer is saved in the JNIEnv argument address specified in the AttachCurrentThread function.

The GetMethodID API is then used to identify the setStringBuilder and setUI methods of the NativeMsClass. The parameters of the GetMethodID are a class descriptor jclass object which was heretofore made global and identifies a class, the name of the method and the signature of the method. In the while loop the monitor is passed to the myLock object to synchronise. Between the MonitorEnter and MonitorExit the primes are calculated, and a call is made using the JNI API CallVoidMethod to the setStringBuilder method of the NativeMsClass which then calls the setStringBuilder of the MainActivity class through the interface object cListener. In the setStringBuilder of the MainActivity class, the primes are added to the StringBuilder object str. In the last iteration of the while loop the setUI method of NativeMsClass is called. This method then calls the setUI method of the MainActivity through the cListener object. This call enables the RESET button and prints the calculated primes. On leaving the while loop, the JavaVM object must be called to detach the thread. (MIT Education, 2019; Oracle Documentation 2017; Gamua 2020.)

### 6.4.2 On the Java side

On the Java side the init method of the MainActivity class resets the main variables when the app is launched. In the startNativeThread method of the same class, the native method startNativeThread declared in the NativeMsClass is called using the myWorker object. The setStringBuilder method of mainActivity builds the stringBuilder that would be eventually displayed on the UI. There is one method worth mentioning in the MainActivity class. The setUI method. This method needs to call the runOnUIThread method as shown below (CODE SNIPPET 10). This is the only way the view, in this case the RESET button can be touched.(Gamua 2020.)



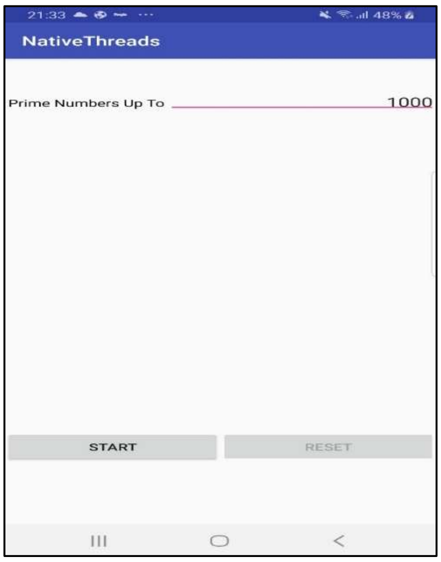
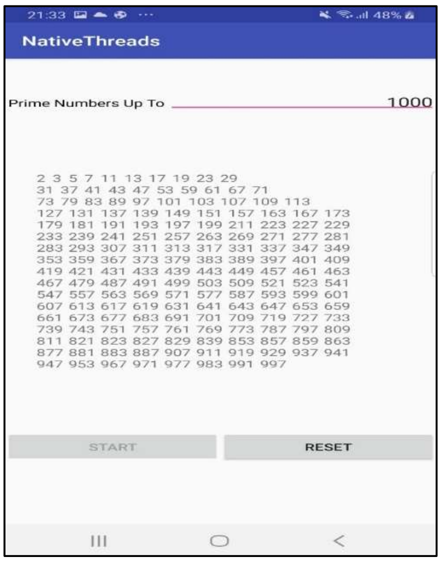
```
runOnUiThread(new Runnable() {
   @Override
   public void run() {
      textView1.setText(str);
      mResetBtn.setEnabled(true);
   }
});
```

CODE SNIPPET 10. Call to runOnUiThread in setUI method of MainActivity class

## 6.5 Results

Presented below in table 15 is a result. It shows the primes calculated up to 1000. The primes are calculated and displayed 10 per line. Upon launching the app, the START button is enabled, and the RESET button disabled. The START button is disabled after it is clicked, and the RESET button enabled. The RESET button clears the UI and cleans the memory on the native side. The results are displayed inside a ScrollView.

TABLE 15. Results for native thread

| | |
|---|---|
|  a) limit entered |  b) output |

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997
```

# 7 STRUCTURES

Copying or Mapping C or C++ struct members to Java is a feat worth achieving. Java does not have a struct data structure so copying the C++ struct data structure to a corresponding Java data structure cannot be talked of. An attempt can be made at mapping struct members to a corresponding Java class member. As this chapter will eventually show, there is no automatic way to copy C++ struct members to Java in such a way that the C++ struct variable can be referenced and manipulated from Java. Code has to be written manually that will copy or map each individual C++ field of the struct to a Java field class member. This is what is being attempted by this chapter.

## 7.1 User interface and activity diagram

The user interface on launching the activity is presented below (Figure 21). In the user interface, there is a TextView that displays the output from a stringBuilder object. Below the TextView, there are three buttons. The first button PRINT OBJ prints the current object. The second button FILL N GET UPD OBJ passes this current Java object to the C++ native side so that the Java object's field could be set by the struct's members and sent back to the Java class to be printed. The RESET button is the last button on the far right of the UI.

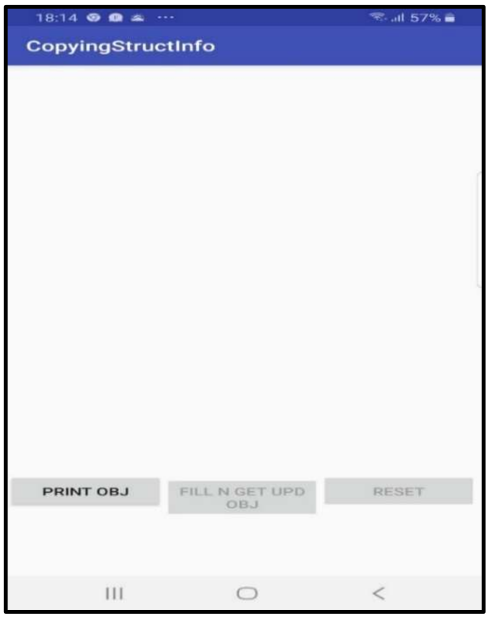


FIGURE 21. User interface for CopyingStructInfo app

In this final chapter, there is no need to create an activity diagram. The activity is simple and sequential following the pattern of the buttons from left to right. When the app is launched the PRINT OBJ button is active while the other 2 buttons are disabled. Clicking this button prints the current Java object of the ITDepartmentInfo class that was instantiated in the init() method of the Java's MainActivity class. Immediately thereafter the FILL N GET UPD OBJ button becomes enabled while the other two buttons are disabled. Clicking this button will send the current Java object to the native side for its members to copy or be set by the struct's members. Clicking this button also enables the RESET button. Clicking the RESET clears the TextView and the Stringbuilder object, instantiates a new ITDepartmentInfo class object and deletes the current struct object. (Gamua 2020.)

### 7.2 Brief look at relationship between classes

The class diagram is shown below (Figure 23). The main points to note of here is that the NativeMsClass defines the native methods and implements the StructCopyListener. In the StructCopyListener interface methods are declared and these methods will be implemented in the MainActivity class and the NativeMsClass. The NativeMsClass is a compositor class to the StructCopyListener interface. There is a LecturerInfo struct. This is found on the native side. (Gamua 2020.)

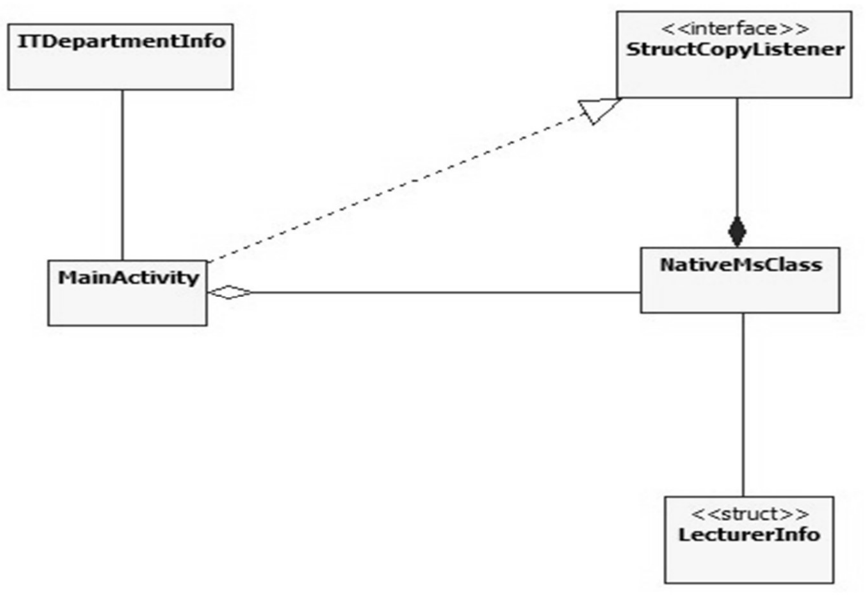
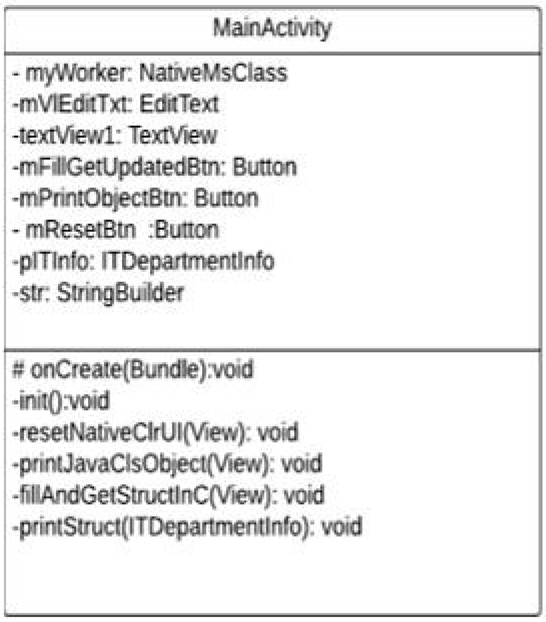


FIGURE 23. Relationship between classes

## 7.3 Detailed look at classes

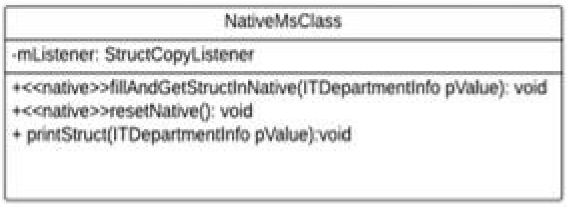
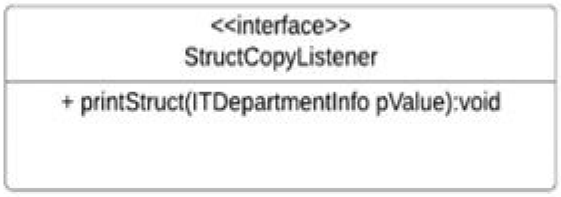
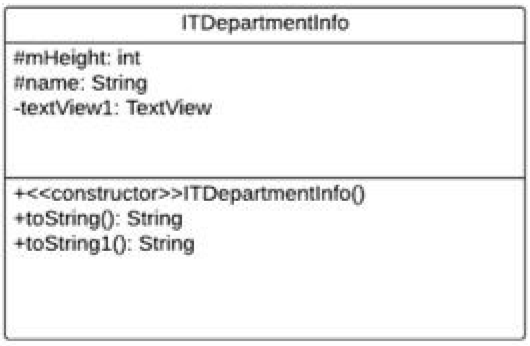
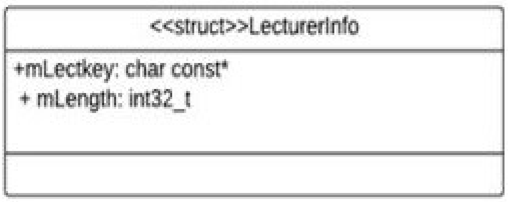
The details of the classes are shown in table 16. The MainActivity's resetNativeClrUI clears the user interface and calls the native method resetNative to clear the memory used in the native library. It also instantiates a new ITDepartmentInfo object. printJavaClsObject prints the information which is held by the pITInfo object. The fillAndGetStructInC calls the fillAndGetStructInNative. The later method is passed the pITInfo object. This object's members will be edited in the native-lib by replacing them with the LecturerInfo struct object's members. The MainActivity's printStruct method is automatically called from the native side when this editing is over. This method prints the new state of the pITInfo object. MainActivity also implements the StructCopyListener interface. The ITDepartmentInfo is a simple class with members to hold the lecturer's name and height. The NativeMsClass has a declaration of the native methods and implements the StructCopyListener interface. The LecturerInfo struct has a pointer to a constant char as member to hold the name of the lecturer and an int member for the height of the lecturer. Note should be made that there must be a one to one correspondence between the struct members and the members of the Java class receiving the struct's members. This is the reason why the class ITDepartmentInfo has 2 class members just like the struct. (Gamua 2020.)

TABLE 16. Classes in CopyingStructInfo app

| class | Functions in Brief |
|---|---|
|   MainActivity<br>- myWorker: NativeMsClass<br>-mVlEditTxt: EditText<br>-textView1: TextView<br>-mFillGetUpdatedBtn: Button<br>-mPrintObjectBtn: Button<br>- mResetBtn :Button<br>-pITInfo: ITDepartmentInfo<br>-str: StringBuilder<br># onCreate(Bundle):void<br>-init():void<br>-resetNativeClrUI(View): void<br>-printJavaClsObject(View): void<br>-fillAndGetStructInC(View): void<br>-printStruct(ITDepartmentInfo): void | The MainActivity's resetNativeClrUI clears the user interface and calls the native method resetNative to clear the memory used in the native library. It also instantiates a new ITDepartmentInfo object. printJavaClsObject prints the information which is held by the pITInfo object. The fillAndGetStructInC calls the fillAndGetStructInNative of the NativeMsClass. The later method is passed the pITInfo object. This object will be edited in the native-lib and sent back to Java |

(Continues)

TABLE 16. (Continued)

| | |
|---|---|
|  NativeMsClass<br>-mListener: StructCopyListener<br>+<<native>>fillAndGetStructInNative(ITDepartmentInfo pValue): void<br>+<<native>>resetNative(): void<br>+ printStruct(ITDepartmentInfo pValue):void | The NativeMsClass has a declaration of the native methods and implements the StructCopyListener interface. |
|  <<interface>><br>StructCopyListener<br>+ printStruct(ITDepartmentInfo pValue):void | Interface implemented by MainActivity and NativeMsClass |
|  ITDepartmentInfo<br>#mHeight: int<br>#name: String<br>-textView1: TextView<br>+<<constructor>>ITDepartmentInfo()<br>+toString(): String<br>+toString1(): String | The ITDepartmentInfo is a simple class with members to hold the lecturer's name and height. This information will be sent to the native -lib to be edited |
|  <<struct>>LecturerInfo<br>+mLectkey: char const*<br>+ mLength: int32_t | The LecturerInfo struct has a pointer to a constant char as member to hold the name of the lecturer and an int member for the height of the lecturer. |

## 7.4 Program description

On clicking the PRINT OBJ button program execution goes to the printJavaClsObject method because of the onClickListener attached to this button. Here the current Java object is added to the StringBuilder object str and printed on the textView textView1. This button immediately becomes disabled and the next button FILL N GET UPD OBJ becomes enabled while the RESET button stays disabled. A visible outcome of clicking this is that the current objects information is printed on the output. (Gamua 2020.)

On clicking the FILL N GET UPD OBJ program execution goes to the fillAndgetStructInC method. Here 3 things happen. The myWorker object calls the NativeMsClass' fillAndGetStructInNative method. This in turn launches the corresponding native method in the native-lib.cpp. The FILL N GET UPD OBJ button then becomes disabled and the RESET button becomes enabled. A corresponding result on the user interface is that the struct members that have now been copied to the Java class object members are printed on the output. (Gamua 2020.)

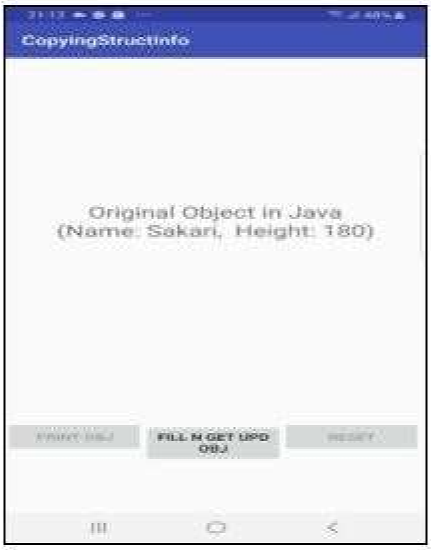
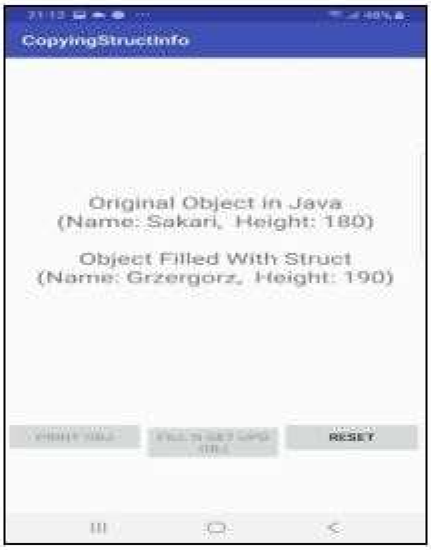
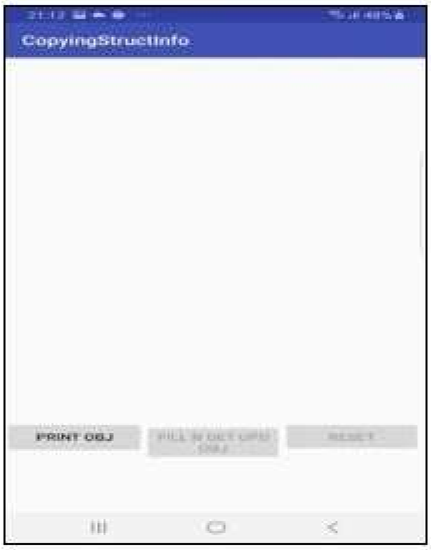
On clicking the RESET button, program execution passes to the resetNativeClrUI method. Here the TextView object textview1 is cleared, the Stringbuilder object is cleared, a new ITDepartmentInfo class object instantiated, the native method resetNative is called and the RESET button is disabled while the PRINT OBJ is enabled. A visible outcome is that textView1 is cleared, therefore clearing the user interface. (Gamua 2020.)

The native-lib is connected with the mainActivity through the MainActivity's fillAndgetStructInC method and the resetNativeClrUI methods that connect to the native-lib via methods from the nativeMsClass. Clicking the FILL N GET UPD OBJ launches the fillAndgetStructInC method. In this method the myWorker object calls the NativeMsClass' fillAndGetStructInNative native method. On the native side a new LecturerInfo struct object is instantiated in the native-lib's native fillAndGetStructInNative method. The fillInfo method is used to fill information about this struct variable. This information will be used to set the various members of the ITdepartmentInfo class' object members that is being referenced on the native side. In order to access the Java int field of the ITDepartmentInfo class, the field ID of the int field mHeight must be determined. This is done with the GetFieldID API which takes as parameters a reference object to the class in question. This reference had been cached as a global variable in the JNI_onLoad method. Its type is a jclass. The next parameter is the field name and finally the last parameter is a Java type of the field name indicated by the "I". After getting the field ID, the setIntField API is used to set the particular int field of the object whose reference was passed in JNI. A new string is created using the NewStringUTF API. This takes as parameter the string that is being referenced by the pointer member of the struct object.  The GetFieldID is used to get the field ID of the Java String class member. The setObjectField API is used to set the String member to the string which was created using the NewStringUTF API. The CallVoidMethod method of the JNI API is then used to call the printStruct method of the NativeMsClass. The CallVoidMethod method takes as parameters the environment variable, the method ID of the method to be called whose value had been determined and cached in the JNI_onLoad() method and the actual parameters to be passed to the called method. Clicking the RESET button will launch the resetNative method of the NativeMsClass which will then launch the corresponding native method in the native library native.lib. The function in this library does just one thing, delete the struct object.   (MIT Education 2019; Oracle Documentation 2017; Gamua 2020.)

## 7.5 Results

Presented below is one result (TABLE 17). In the first image from the left, the PRINT OBJ button has been clicked displaying the current object. This button becomes disabled immediately after it is clicked. The FILL N GET UPD OBJ button becomes enabled. Clicking this button generates the second image on the right. In the second image the object has been filled with the contents of the struct in C++ on the native side. The first 2 buttons are then disabled and only the RESET button is enabled. RESET clears the UI.

TABLE 17. Results of copying structs

| | |
|---|---|
|   CopyingStructInfo<br>PRINT OBJ<br>FILL N GET UPD OBJ<br>RESET Original Object in Java<br>(Name: Sakari, Height: 180)<br>a) Original object in Java |   CopyingStructInfo<br>PRINT OBJ<br>FILL N GET UPD OBJ<br>RESET Original Object in Java<br>(Name: Sakari, Height: 180)<br>Object Filled With Struct<br>(Name: Grzergorz, Height: 190)<br>b)object filled with struct in C++ |
|   CopyingStructInfo<br>PRINT OBJ<br>FILL N GET UPD OBJ<br>RESET c) reset | |

# 8 CONCLUSION

In order to develop apps, a programmer can rely only on languages like Java or Kotlin. In these languages the programmer is presented with a vast development toolset to do almost anything he/she wants to do. Apps therefore can be programmed exclusively in these languages. But as mentioned in the introductory chapter, code leveraging to C++ adds more power to the programmer's arsenal. In addition to the reasons given in the introductory chapter to use NDK for app development, other considerations need to be taken into account when developing apps. Such considerations may warrant the use of NDK. Such considerations are explained in the following paragraphs. (Android Developers Documentation 2019).

If an app were to be developed that has to intensively use the CPU for certain tasks, it will be a good idea to leverage part of the code that uses much CPU time to C++. If this is not done, this will affect the performance of the app as the CPU will run more cycles in the Java code and less for the hybrid code. Java compiles to bytecode and to run on an Android platform, a JVM has to be run contrary to C++ that is compiled to the machine code of the phone. This means the CPU has to work harder to run a Java only program than to run the hybrid program.

The performance of apps that use much memory will be improved if sections of such apps are written in C/C++. If such apps were written purely in Java, more memory will be needed, and this will ultimately stand on the way of performance. A JVM has to periodically run the garbage collector and this is an expensive process that requires memory and takes up CPU time. In an app like a mobile game, the frames have to be updated periodically. This update requires more memory and CPU usage so if a JVM also has to run the garbage collector at the same time that a frame is being updated, the performance will surely be affected. This is also true for apps that perform physics simulations.

If an encryption algorithm is to run on a mobile platform, it is evident that the prospects for such traditional Java source code protection to perform below the expected target, are quite considerable. Encryption in Java is accomplished with the custom class loader encryption and codes from the custom class loader can be easily decompiled. Encryption and decryption code algorithm, when run using C/C++ may be much safer and difficult to crack. In this light, it would be wise to draw on the loftier qualities of C/C++ to achieve a much reliable encryption and decryption, whenever the situation demands in mobile apps.

## REFERENCES

Android Developers Documentation, 2019. Android Runtime (ART) and Dalvik. Available at: https://source.android.com/devices/tech/dalvik/ Accessed: 22 October 2019.

Android Developers Documentation, 2019. Getting Started with NDK. Available at: https://developer.android.com/ndk/guides Accessed: 20 October 2019.

Android Developers Documentation, 2019. Configure Your Build. Available at: https://developer.android.com/studio/build Accessed: 20 October 2019.

Android Developers Documentation, 2019. JNI Tips. Available at: https://developer.android.com/training/articles/perf-jni Accessed: 11 January 2020.

Booch, G., Rumbaugh, J. & Jacobson, I., 2005. Unified Modelling Language User Guide. Second edition. Palo Alto: Addison Wesley.

CMake, 2019. Build with CMake. Build With Confidence. Available at: https://cmake.org/ Accessed: 25 October 2019.

Emory College of Arts and Sciences, 2020. Parallel Programming with PThread API. Available at: http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/91-pthreads/join-threads.html Accessed: 20 February 2020.

Gamua, E., 2020. ElvisGamua/playing_with_jni. Available at: https://github.com/ElvisGamua/playing_with_jni
Accessed: 3 July 2020.

Geeks for Geeks , 2020. Unified Modeling Language (UML): Activity Diagram. Available at: https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/
Accessed: 15 January 2020.

IBM Documentation, 2020. Library Functions. Available at: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxbd00/keyword.htm Accessed: 16 February 2020.

Krajci, I. & Cummings, D., 2013. Android on X86: An Introduction to Optimising for Intel Architecture. First edition. New York: Apress.

Liang, D., 2015. Introduction to Java Programming. 10th Edition. New Jersey: Prentice Hall.

Liang, S., 1999. The Java Native Interface: A Programmers Guide and Specification. First edition. Palo Alto: Addison Wesley.

MIT Education, 2019. Invoking Java Virtual Machine.
Available at: https://web.mit.edu/javadev/doc/tutorial/native1.1/implementing/invo.html Accessed: 1 February 2020.

MIT Education, 2019. JNI Functions.
Available at: https://web.mit.edu/java_v1.5.0_22/distrib/share/docs/guide/jni/spec/functions.html Accessed: 8 December 2019.

MIT Education, 2019. The Java Native Interface.
Available at: http://web.mit.edu/javadev/doc/tutorial/native1.1/implementing/index.html Accessed: 15 October 2019.

MIT Education, 2019. Threads and Native Methods.
Available at: https://web.mit.edu/javadev/doc/tutorial/native1.1/implementing/sync.html Accessed: 8 December 2019.

Oracle Documentation, 2019. Java Native Interface Specification Contents.
Available at: https://docs.oracle.com/javase/10/docs/specs/jni/index.html Accessed: 17 September 2019.

StackOverflow, 2017. JNI Thread Model?.
Available at: https://stackoverflow.com/questions/38378901/jni-thread-model Accessed: 7 October 2019.

The Fossies Software Archive, 2020. Pthreads: Data Structures.
Available at: https://fossies.org/dox/pthreads-3.14/annotated.html Accessed: 4 February 2020.

Vogel, L. & Scholz, S., 2012. Building Android Applications with Gradle: Gradle Tutorial. Available at: https://www.vogella.com/tutorials/AndroidBuild/article.html. Accessed:  November 2019.