BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 6

DCA3201

MOBILE APPLICATION DEVELOPMENT

Unit 6

Android Runtime

Table of Contents

1. INTRODUCTION

Hey there, future Android developers!

So, you’ve been diving into the world of Android development, and by now, you’ve probably heard terms like “Dalvik,” “ART,” or “JIT” tossed around. If you’re scratching your head wondering what these acronyms even mean, you’re in the right place! Unit 6 is all about demystifying the Android Runtime. Why should you care? Well, understanding the Android Runtime is like getting to know the engine that powers a car. Sure, you can drive without knowing what’s under the hood, but understanding the mechanics can make you a better, more efficient driver.

In the tech world, knowledge is power. Knowing how Android Runtime works can help you write better apps, optimize performance, and even debug like a pro. Ready to dive in?

1.1 Learning Objectives

By the end of this unit, you should be able to:

- ❖ *Remember the key differences between Dalvik Virtual Machine and Android Runtime (ART).*
- ❖ *Understand the role of Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation in Android Runtime.*
- ❖ *Apply the knowledge of core libraries in Android.*
- ❖ *Analyze the significance of Java Interoperability Libraries in Android Runtime.*
- ❖ *Evaluate the impact of the Android Runtime on the performance and efficiency of Android applications.*
- ❖ *Create an optimized Android application that leverages the capabilities of the Android Runtime.*

Ready to jump-start your journey into the fascinating world of Android Runtime? Let’s get started!

2. DALVIK VIRTUAL MACHINE V/S ANDROID RUNTIME (ART)

The concept of a virtual machine (VM) is pivotal in the realm of Android development. A VM acts as an abstraction layer between the compiled application code and the hardware, interpreting or compiling bytecode into native instructions understood by the device's CPU. This abstraction facilitates app portability across various hardware configurations, one of Android's key strengths.

Originally, Android employed the Dalvik Virtual Machine (DVM) for this purpose. However, it was eventually replaced by Android Runtime (ART) to address several performance and efficiency issues. To grasp why this transition was necessary, it's crucial to understand the features, advantages, and disadvantages of both.

2.1 Features of Dalvik Virtual Machine

Just-In-Time Compilation (JIT): Dalvik used a Just-In-Time (JIT) compiler, converting bytecode to native code at runtime. While this strategy reduced the app's installation size, it had performance costs, especially during the initial execution of the app.

When an Android app runs, the JIT compiler translates the app's bytecode into native machine code for the device's processor, doing this translation on-the-fly, or "just in time," as the app runs.

Garbage Collection: Dalvik had a less efficient garbage collection system compared to ART. While garbage collection took place, the application would often freeze, leading to a subpar user experience.

Memory and Battery Efficiency: Dalvik was generally less efficient in terms of memory usage and battery consumption. It was designed at a time when Android devices had more limited resources, and as such, it didn't scale well with the increasing capabilities of modern hardware.

Pros of JIT:

- Memory Efficiency: Since JIT compiles only the portions of bytecode that are needed at runtime, it saves storage space.

- Dynamic Optimization: JIT can perform optimizations based on runtime conditions, potentially leading to more efficient code execution.

Cons of JIT:

- Startup Latency: The on-the-fly compilation leads to a delay in the app's initial startup time.
- CPU Overhead: The continuous compilation process consumes CPU cycles, affecting the overall performance and battery life.

2.2 Features of Android Runtime (ART)

Unlike Dalvik, ART uses Ahead-of-Time (AOT) compilation, fundamentally changing how apps are executed.

Ahead-of-Time Compilation (AOT): ART introduced an Ahead-of-Time (AOT) compiler, which converts bytecode to native code during app installation. This shift dramatically improved app execution speed, although it did increase the initial installation time and the installed app size.

Improved Garbage Collection: ART features an improved, more efficient garbage collection system that minimizes app pauses, resulting in a smoother user experience.

64-bit Support: With the advent of 64-bit processors in mobile devices, ART was designed to be compatible with 64-bit instructions, unlike Dalvik, which was confined to 32-bit architectures.

Profile-Guided Compilation: ART employs profile-guided compilation, using app usage data to continually optimize the compiled code. This adaptive strategy improves both speed and efficiency.

Pros of AOT:

- Faster Execution: Since the code is pre-compiled, apps launch and run faster.
- Optimized Garbage Collection: AOT allows for more effective garbage collection, reducing pauses and stutters in app performance.

Cons of AOT:

- Increased Installation Time: The pre-compilation step adds to the installation time of apps.
- Larger App Size: Pre-compiled native code takes up more storage space than bytecode.

2.3 Comparative Analysis: Dalvik vs ART

- Performance: ART outperforms Dalvik in terms of execution speed, thanks to its AOT compiler and optimized garbage collection. Apps launch faster and run more smoothly on ART.
- Efficiency: ART is more memory- and battery-efficient than Dalvik. Its intelligent runtime optimizations, such as profile-guided compilation, allow it to adapt to the user's specific needs, thereby conserving resources.
- Compatibility: Transitioning from Dalvik to ART required developers to ensure that their apps were compatible with the new runtime. However, ART's benefits generally outweighed the challenges posed by this transition.
- Future-Proof: ART is better suited for the demands of modern Android devices, including those with 64-bit architectures. It is designed to be extensible, laying the groundwork for future enhancements to the Android platform.

2.4 Memory Management: Dalvik vs ART

Both Dalvik and ART have their strategies for managing memory, an essential aspect of any runtime environment.

2.4.1 Dalvik's Memory Management:

- Zygote: Dalvik used a process called Zygote to handle memory allocation. Zygote preloads classes and resources common to multiple apps, aiming to reduce the startup time.
- Garbage Collection: Dalvik employed a "stop-the-world" garbage collector, meaning that all application threads would be paused during garbage collection, leading to potential performance hitches.

2.4.2 ART's Memory Management:

- Improved Garbage Collection: ART uses a generational garbage collector that categorizes objects into young and old generations, optimizing the garbage collection process and reducing app pauses.

By understanding the evolution from Dalvik to ART, developers can better appreciate the current capabilities and constraints of Android development. The shift to ART was more than just an upgrade; it was a transformation that elevated Android's performance, efficiency, and future adaptability.

3. EXPLORING CORE LIBRARIES

3. 1 Android's Native Libraries

The Android operating system incorporates a host of native libraries that extend the capabilities of the Android Runtime (ART). These libraries offer functions for everything from low-level device control to advanced graphics rendering, making them vital for Android application development.

- OpenGL ES: This library is vital for the rendering of high-end graphics in Android applications. The 'ES' stands for 'Embedded Systems,' emphasizing its optimization for mobile and embedded devices. This library is often used in gaming apps and other apps requiring complex graphics rendering.
- SQLite: Android incorporates SQLite as its database management system. It is a serverless, transactional SQL database engine, and its lightweight nature makes it ideal for mobile devices where resources are limited.
- WebKit: This open-source web browser engine enables Android apps to display web content. WebKit is also the backbone of many popular web browsers, offering robust HTML5 and CSS support.
- SSL: The Secure Sockets Layer is a security protocol used in Android to establish encrypted links between a server and a client—typically a web server and a browser, or a mail server and a mail client.

3.2 Core Java Libraries

Android's architecture is deeply tied to Java, and this connection is evident in the array of core Java libraries integrated within Android Runtime. These libraries offer functionalities similar to those in standard Java SE (Standard Edition) and are crucial for basic programming tasks.

- java.util: This package provides the data structures fundamental to Java programming, including collections like ArrayLists, HashMaps, and Sets, which are often used for data manipulation and storage in Android apps.

- java.io and java.nio: These are input/output libraries that handle file operations, data streams, and networking, among other things. These libraries are essential for any form of data transfer or manipulation within Android apps.
- java.net: This is the Java package for network operations and encapsulates classes that handle network protocols like HTTP and FTP. Networking is a fundamental aspect of many Android applications, whether they are client-server apps, peer-to-peer apps, or any other architecture that requires remote data access.

3.3 Custom Libraries in Android

Developers are not limited to the libraries provided by Android. They can also incorporate custom libraries for specialized functionalities. Many third-party libraries offer advanced features, improved performance, or more straightforward implementations of complex tasks. Some popular examples include:

- Retrofit: A type-safe HTTP client for Android and Java, simplifying the process of connecting to REST-based web services.
- Glide: An image loading and caching library focused on smooth scrolling, making it easier to handle image-intensive applications.
- Room: A persistence library that offers an abstraction layer over SQLite, simplifying database operations and ensuring type safety.

By leveraging these libraries, developers can not only speed up the development process but also introduce robust and optimized features in their applications.

4. UNDERSTANDING JAVA INTEROPERABILITY LIBRARIES

4.1 Java Native Interface (JNI)

The Java Native Interface (JNI) is a crucial part of Android's Java interoperability toolkit. It serves as a bridge between Java code and native code written in languages like C or C++. This capability allows developers to leverage high-performance, low-level code for specific tasks, like complex mathematical calculations or direct hardware interactions.

Basics of JNI: The JNI provides a set of specifications that allow Java code running within a Java Virtual Machine (JVM) to interoperate with applications and libraries written in other languages like C and C++.

- JNI in Android: Android takes advantage of JNI to enable the Java application framework to interact seamlessly with native libraries like OpenGL for graphics or SQLite for database management. This combination of high-level Java code and low-level native code provides the flexibility and performance needed for a wide range of mobile applications.
- Security Implications: While JNI allows for powerful functionality, it also opens up additional security considerations. Native code is not subject to the same sandboxing and security constraints as Java code, making it a potential attack vector if not managed carefully.

4.2 Android NDK and SDK

The Android Native Development Kit (NDK) and Software Development Kit (SDK) are two essential tools for Android development, each with distinct purposes and capabilities.

- Android SDK: This is the primary tool for Android application development and is oriented toward Java programming. The SDK provides all the Java libraries, API documentation, and developer tools needed to build, test, and debug Android apps.
- Android NDK: This toolset complements the SDK by providing libraries and tools for embedding native code (C, C++) in Android applications. While most apps won't need

the NDK, it is invaluable for CPU-intensive tasks that require the performance of native code.

4.3 Use Cases for Java Interoperability

Java interoperability is not always necessary, but there are specific use-cases where it becomes invaluable.

- Performance Optimization: Tasks like image processing, data encryption, or real-time audio and video processing can benefit from the performance optimization that native code offers.
- Legacy Code: Organizations that have pre-existing libraries written in C or C++ can use JNI to leverage this code within Android apps, saving time and effort.
- Hardware-Level Access: Native code can provide more direct access to hardware components, offering functionalities that may not be accessible through Java APIs.
- Scientific Computing: High-performance computing tasks, often encountered in scientific applications, can be more effectively handled with native code than with Java.

By understanding when and how to use Java interoperability libraries and native code, developers can make informed choices about the architecture and implementation of their Android applications. This knowledge allows for the creation of apps that can exploit the full range of capabilities offered by modern mobile devices, combining the ease and portability of Java with the power and flexibility of native programming languages

5. SUMMARY

Congratulations on making it to the end of this in-depth exploration of Android Runtime! If this chapter were a journey, you've certainly covered some significant ground.

We kicked things off by diving into the evolutionary tale of Android's runtime environment, from the Dalvik Virtual Machine to the Android Runtime (ART). You've seen how Android transitioned from Dalvik's Just-In-Time (JIT) compilation to ART's Ahead-Of-Time (AOT) compilation to meet the ever-growing demands of modern mobile devices. You've understood the pros and cons of each system, from JIT's dynamic but CPU-intensive operations to AOT's fast but storage-hungry nature.

We then took a tour of the core libraries that are the building blocks of any Android application. Whether it's rendering stunning graphics with OpenGL ES, managing databases with SQLite, or displaying web content using WebKit, these libraries are the unsung heroes that make your apps tick.

And let's not forget Java Interoperability! You've learned how Android utilizes Java Native Interface (JNI) to bridge the gap between high-level Java code and low-level native code. We also touched upon the use-cases for JNI and the Android Native Development Kit (NDK), showing you when to opt for native code for performance gains or for leveraging legacy libraries.

By now, you should be equipped with a solid understanding of:

- The transition from Dalvik to ART: Why it happened and how it affects app development.
- Compilation techniques: The nuances of JIT and AOT and their impact on app performance and size.
- Core Libraries: A rundown of the essential libraries in Android that you'll likely use in your app development journey.
- Java Interoperability: The role of JNI and NDK in enhancing your Android apps by bridging Java with native code.

With this knowledge under your belt, you’re not just an Android app developer; you’re an informed Android app developer. And that, dear learner, makes all the difference. So go ahead, apply what you’ve learned, and build apps that aren’t just functional but are also optimized and efficient. Here’s to many more coding adventures!

6. GLOSSARY

- Android Runtime (ART): The environment in which Android apps execute their code. ART uses Ahead-Of-Time (AOT) compilation to improve performance and efficiency.
- Dalvik Virtual Machine (DVM): The predecessor to ART, Dalvik was the runtime environment for older versions of Android. It used Just-In-Time (JIT) compilation.
- Just-In-Time Compilation (JIT): A runtime compilation technique used by Dalvik to translate bytecode into native machine code just as the app is running.
- Ahead-Of-Time Compilation (AOT): A compilation technique used by ART where bytecode is pre-compiled into native code during app installation.
- Bytecode: Intermediate code generated by the Java compiler, which is later converted into native machine code by the runtime environment.
- Native Code: Machine-level code that can be directly executed by the device's CPU.
- Zygote: A process in Dalvik's memory management system that preloads classes and resources common to multiple apps to reduce startup time.
- Garbage Collection: The automatic process of identifying and freeing up memory that is no longer in use by the application.
- Core Libraries: The set of standard libraries that provide essential functionalities in Android, such as SQLite for database management, OpenGL ES for graphics, and SSL for secure network communication.
- Java Native Interface (JNI): A programming framework that allows Java code to interact with code written in other languages like C and C++.
- Android Native Development Kit (NDK): A set of tools that allows developers to write parts of their app using native languages like C and C++.
- Android Software Development Kit (SDK): A collection of software tools and libraries that facilitate Android app development, primarily using Java or Kotlin.
- Debugging: The process of finding and resolving defects or problems within a computer program.
- Profiler: A tool used to analyze the program's run-time behavior, including CPU usage, memory allocation, and thread activity.

- CPU Overhead: The additional computational work that is performed by the CPU, often leading to reduced performance and increased battery consumption.
- Log Analysis: The process of examining logs generated by an application to identify patterns, errors, or other useful information.
- Real-Time Notifications: Instant alerts or messages that are sent to the user's device as events occur, often implemented through Broadcast Receivers in Android.
- Memory Allocation: The process of reserving memory space for storing variables and data during the execution of a program.
- OpenGL ES: A subset of the OpenGL computer graphics rendering application programming interface for rendering 2D and 3D graphics.
- SSL (Secure Sockets Layer): A protocol for secure network communication, commonly used in secure data transmission over the internet.

7. CASE STUDY

Optimizing a Social Media App

Background

Imagine you are part of a development team tasked with optimizing an existing Android-based social media application. The application has been in the market for a year and has garnered a reasonable user base. However, user reviews indicate that the app suffers from slow load times, frequent crashes, and high battery consumption.

The app was initially developed when Dalvik was the primary runtime for Android devices. It heavily relies on complex image and video rendering, as well as real-time notifications. Your team's mission is to optimize the app's performance, stability, and efficiency.

Case Scenarios

1. Slow Image Rendering: Users complain about the slow loading of images in their feed.
2. High Battery Drain: Battery consumption spikes dramatically when the app is in use.
3. App Crashes: Frequent crashes occur when users try to upload videos.
4. Delayed Notifications: Real-time notifications for likes, comments, and follows are significantly delayed.

Questions

1. Which runtime environment would you recommend for the app: Dalvik or ART? Justify your choice.
2. What specific libraries or tools can be used to speed up image rendering in the app?
3. How would you approach diagnosing the cause of the app's crashes?
4. What steps can be taken to reduce battery drain during the app's operation?
5. Can Java Native Interface (JNI) be useful for any of the case scenarios mentioned? If so, how?

Self-Assessment Questions - 1

1. Which runtime environment was primarily used in the early versions of Android?

   Section: Dalvik Virtual Machine vs Android Runtime (ART)

   a) A. ART
   b) B. JVM
   c) C. Dalvik
   d) D. .NET
2. What type of compilation does Dalvik Virtual Machine use?

   Section: Just-In-Time Compilation (JIT) in Dalvik

   a) A. Ahead-Of-Time (AOT)
   b) B. Just-In-Time (JIT)
   c) C. Post-In-Time (PIT)
   d) D. Real-Time (RT)
3. Wat is a drawback of Ahead-Of-Time (AOT) compilation in ART?

   Section: Ahead-of-Time Compilation (AOT) in ART

   a) A. Increased installation time
   b) B. Reduced app performance
   c) C. High CPU usage
   d) D. Slower garbage collection
4. Which library is commonly used for rendering graphics in Android apps?

   Section: Core Libraries

   a) A. SQLite
   b) B. OpenGL ES
   c) C. SSL
   d) D. WebKit
5. What is the primary function of the Java Native Interface (JNI)?

Section: Java Native Interface (JNI)

a) A. Database management
b) B. Network operations
c) C. Bridging Java code with native code
d) D. Web rendering

6. Which Android development kit is focused on Java programming?

Section: Android NDK and SDK

a) A. Android GDK
b) B. Android SDK
c) C. Android NDK
d) D. Android J2ME

7. What does JIT stand for?

Section: Just-In-Time Compilation (JIT) in Dalvik

a) A. Java Internal Testing
b) B. Just Install This
c) C. Just-In-Time
d) D. Java In Time

8. What type of garbage collection does ART use?

Section: Memory Management: Dalvik vs ART

a) A. Stop-the-world
b) B. Generational
c) C. Incremental
d) D. Parallel

9. Which library is used for secure network communications in Android?

Section: Core Libraries

a) A. SSL
b) B. OpenGL ES

c) C. SQLite
d) D. WebKit

10. What is Zygote in the context of Dalvik's memory management?

    Section: Memory Management: Dalvik vs ART

    a) A. A garbage collector
    b) B. A memory allocator
    c) C. A class preloader
    d) D. A heap manager

11. What is the primary advantage of using Android NDK?

    Section: Android NDK and SDK

    a) A. Easy integration with web services
    b) B. High-level abstractions
    c) C. Performance optimization for CPU-intensive tasks
    d) D. Quick app prototyping

12. Which database engine does Android primarily use?

    Section: Core Libraries

    a) A. MySQL
    b) B. PostgreSQL
    c) C. SQLite
    d) D. MongoDB

13. What is Ahead-Of-Time (AOT) compilation?

    Section: Ahead-of-Time Compilation (AOT) in ART

    a) A. Compilation during runtime
    b) B. Compilation before runtime
    c) C. Compilation after runtime
    d) D. Compilation at the end of runtime

14. Which toolset allows you to write native C and C++ code for Android?

Section: Android NDK and SDK

a) A. Android SDK
b) B. Android GDK
c) C. Android NDK
d) D. Android J2ME

15. What is the primary drawback of using Just-In-Time (JIT) compilation?

Section: Just-In-Time Compilation (JIT) in Dalvik

a) A. Larger app size
b) B. Slower app installation
c) C. CPU overhead during app execution
d) D. Incompatibility with older Android versions

8. TERMINAL QUESTIONS

1. Explain the key differences between Dalvik Virtual Machine and Android Runtime (ART).
2. Why did Android transition from Dalvik to ART? Discuss the advantages and limitations of each.
3. Describe how Just-In-Time (JIT) compilation works in Dalvik Virtual Machine.
4. What are the pros and cons of using Just-In-Time (JIT) compilation in Dalvik?
5. Explain the concept of Ahead-Of-Time (AOT) compilation in Android Runtime.
6. Discuss the advantages and disadvantages of Ahead-Of-Time (AOT) compilation.
7. How does memory management differ between Dalvik and ART?
8. Elaborate on the concept of Zygote in Dalvik's memory management.
9. List and describe some of the core libraries in Android and their functionalities.
10. What is the role of Java Native Interface (JNI) in Android Runtime?
11. Discuss the security implications of using Java Native Interface (JNI).
12. Explain the differences between Android SDK and Android NDK.
13. In what scenarios would you recommend using Android NDK over SDK?
14. How can Java Interoperability be beneficial in Android app development? Provide examples.
15. Describe the process of diagnosing app crashes in Android. What tools would you use?
16. How can one optimize image rendering in an Android app?
17. What steps can be taken to reduce battery drain in an Android application?
18. Explain the term 'Garbage Collection' in the context of Android Runtime. How does ART improve upon Dalvik in this aspect?
19. Discuss the use cases where Ahead-Of-Time (AOT) compilation would be more beneficial than Just-In-Time (JIT) compilation.
20. How does the choice of Android Runtime environment affect the user experience of an app?

9. ANSWERS

9.1 Case Study

1. ART (Android Runtime) would be the recommended runtime environment. ART provides Ahead-of-Time (AOT) compilation, which significantly improves app performance and responsiveness. It's particularly useful for an app that relies heavily on image and video rendering.
2. Libraries like OpenGL ES for graphics rendering and Glide for image loading and caching can be very effective in speeding up image rendering.
3. Diagnosing app crashes would likely involve a mixture of log analysis, debugging, and profiling tools available in the Android SDK. It might also be beneficial to look into Crash Reporting tools that can provide real-time insights into issues as they occur in the field.
4. Battery drain can be reduced by optimizing background processes, reducing the frequency of network calls, and efficiently managing resources like GPS and sensors. Utilizing Android's Battery Profiler can help identify the specific components that are consuming excessive battery.
5. JNI could be beneficial for real-time notifications and video processing. Native code can handle these tasks more efficiently, reducing both the time taken and the CPU cycles consumed, ultimately saving battery life.

9.2 Self-Assessment Questions

1. C. Dalvik
2. B. Just-In-Time (JIT)
3. A. Increased installation time
4. B. OpenGL ES
5. C. Bridging Java code with native code
6. B. Android SDK
7. C. Just-In-Time
8. B. Generational
9. A. SSL
10. C. A class preloader

11. C. Performance optimization for CPU-intensive tasks
12. C. SQLite
13. B. Compilation before runtime
14. C. Android NDK
15. C. CPU overhead during app execution

9.3 Terminal Questions

1. Refer to Section Dalvik Virtual Machine vs Android Runtime (ART)
2. Refer to Section Evolutionary Background
3. Refer to Section Just-In-Time Compilation (JIT) in Dalvik
4. Refer to Section Just-In-Time Compilation (JIT) in Dalvik
5. Refer to Section Ahead-of-Time Compilation (AOT) in ART
6. Refer to Section Ahead-of-Time Compilation (AOT) in ART
7. Refer to Section Memory Management: Dalvik vs ART
8. Refer to Section Memory Management: Dalvik vs ART
9. Refer to Section Core Libraries
10. Refer to Section Java Native Interface (JNI)
11. Refer to Section Java Native Interface (JNI)
12. Refer to Section Android NDK and SDK
13. Refer to Section Android NDK and SDK
14. Refer to Section Java Interoperability
15. Refer to Section Just-In-Time Compilation (JIT) in Dalvik (debugging tools are discussed in relation to JIT)
16. Refer to Section Core Libraries (OpenGL ES is discussed)
17. Refer to Section Just-In-Time Compilation (JIT) in Dalvik (CPU overhead is discussed)
18. Refer to Section Memory Management: Dalvik vs ART
19. Refer to Section Ahead-of-Time Compilation (AOT) in ART
20. Refer to Section Dalvik Virtual Machine vs Android Runtime (ART)