People's Democratic Republic of Algeria
Ministry of Higher Education & Scientific Research
University of Oum El Bouaghi
Faculty of Exact Sciences and Natural and Life Sciences
Department of Mathematics and Computer Science

Mobile Applications

Silem Abdelheq

Assistant Professor

silem.abdelheq@univ-oeb.dz

Table of Contents

List of Figures

List of Tables

Introduction

To the Reader

This document serves as a comprehensive synthesis of the Mobile Application Development course delivered at the University of Oum El Bouaghi. The field of mobile development is at once fascinating, fast-paced, and technically demanding. Building modern mobile applications is often considered one of the most dynamic challenges in computer science, as it requires a developer to master a diverse ecosystem ranging from fundamental software engineering principles to the specific constraints of handheld hardware.

To navigate this complexity, the student must be equipped with a varied toolkit. This includes foundational knowledge (application lifecycles, resource management, and asynchronous programming) as well as practical technical skills (UI/UX design, database persistence, and hardware integration like sensors and GPS).

In alignment with academic standards, this course is structured to address the multi-layered nature of mobile systems. The first part focuses on the Frontend and User Interface (UI), covering layout design, activity lifecycles, and event handling. The second part explores Data Management and Integration, including local persistence (SQLite/Room), background services, and network communication with remote APIs.

The documentation surrounding mobile development is vast and constantly evolving. Nevertheless, this manuscript draws from several foundational references, including the official Android Documentation, the architectural guidelines from Google, and seminal works on mobile software engineering. I have also integrated insights from industry-standard practices and modern pedagogical frameworks.

While every effort has been made to ensure the accuracy and pedagogical clarity of this work, perfection remains a human ideal and this report is no exception. Given the rapid shifts in mobile technology and the inherent pressure of academic preparation, some errors may persist. Therefore, I kindly invite all readers to signal any errors whether they be typographical, methodological, or technical to help improve future editions of this course.

Objectives

The purpose of this course is to equip students with foundational skills in designing and building applications and computer systems for mobile platforms. As smartphones have become integral to modern life, mobile apps now play a critical role across all user contexts—whether they are consumers (B2C), business partners (B2B), or internal employees (B2E). Additionally, this course focuses on mastering Android app development, exploring its tools and frameworks, and addressing the specialized techniques required for programming on smartphone hardware.

Chapter 1

Introduction to Mobile Applications

Introduction

Nowadays, smartphones have become an integral part of our daily lives due to the wide range of features they offer and their ease of use. Over the years, smartphones have evolved considerably, providing numerous capabilities that enhance everyday life and simplify common tasks. Mobile applications are at the core of this transformation, allowing users to interact directly with content on their smartphones simply by using their fingers.

Nowadays, smartphones have become an integral part of our daily lives due to the wide range of features they offer and their ease of use. Over the years, smartphones have evolved considerably, providing numerous capabilities that enhance everyday life and simplify common tasks. Mobile applications are at the core of this transformation, allowing users to interact directly with content on their smartphones simply by using their fingers.

Mobile applications are designed to operate on mobile devices such as smartphones and tablets, with the objective of providing proactive assistance to users, including access to essential services such as email and calendars. However, as demand for this type of software has increased, the mobile application industry has rapidly expanded to cover additional domains such as gaming, entertainment, logistics, healthcare, manufacturing, and many others.

Mobile applications can be classified into three main categories: native applications, web applications, and hybrid applications. Each category offers distinct advantages and disadvantages, enabling developers to choose the most suitable solution according to the objectives of their project and the characteristics of the target audience.

1.1 Chapter Objectives

This chapter introduces the ecosystem of mobile applications and evaluates their significance in the real world. At the end of this chapter, students will be able to:

- **Understanding the Impact of Mobile Apps:** Critique the socio-economic impact and the technological evolution of mobile platforms in the modern digital landscape.
- **Exploring Mobile App Categories:** Formulate a technical distinction between Native, Web, and Hybrid architectures based on performance requirements and resource constraints.
- **Comparing Mobile Operating Systems:** identify current mobile operating systems and understand the key differences between them.
- **Mapping the App Development Lifecycle:** Master the end-to-end lifecycle of a mobile product, from initial conceptualization through deployment and iterative maintenance.
- **Defining Essential App Qualities:** Establish rigorous criteria for successful applications, prioritizing user data protection, high-responsiveness, and technical scalability.

1.2 The Importance of Mobile Applications

In this section, we illustrate the importance of mobile phones and mobile applications in our lives. Understanding the history of a particular subject can help us appreciate its significance, its evolution over time, and its impact on society. By examining the past, we can also gain insight into the challenges and opportunities that have shaped the field and learn from the successes and failures of previous generations. History provides valuable context for understanding the present and preparing for the future.

In the case of mobile phones, studying their history helps us understand the rapid growth and evolution of the field, the impact of emerging technologies, and the changing expectations of users. It also highlights the challenges faced by developers and designers when creating engaging and useful applications that meet the needs of diverse audiences. By examining the history of mobile phones, we can better appreciate their importance in our daily lives and the role they play in shaping the digital landscape.

1.2.1 The History of Mobile Phones

The history of mobile phones dates to 1947, when Bell Labs invented the first mobile telephone system, known as the Mobile Telephone Service (MTS). This device was bulky, weighing more than 40 kilograms, and offered very limited coverage. Despite its limitations, it

represented a revolutionary achievement that paved the way for future developments in mobile communication technology. [1]

Over the years, mobile phone technology has continued to evolve at a remarkable pace. In 1973, Martin Cooper of Motorola demonstrated the first handheld mobile phone call using the DynaTAC prototype on the streets of New York City. A decade later, the device reached commercial availability in 1983 as the Motorola DynaTAC 8000X — significantly smaller and lighter than earlier vehicle-mounted systems, and offering sufficient battery life for practical everyday use [1], [2].

In 1991, the first commercial GSM (Global System for Mobile Communications) network was launched in Finland by Radiolinja, establishing the world's first standardized digital cellular infrastructure. GSM represented a decisive departure from the fragmented analog systems that had preceded it, enabling international roaming, encrypting voice transmissions, and substantially improving call quality — milestones that collectively defined a new era in mobile telephony [1], [2].

In 1992, IBM demonstrated the Simon Personal Communicator at the COMDEX trade show, with commercial release following in 1994. Simon integrated telephony with computing functionality, incorporating a touchscreen interface, a stylus, and the capacity to send and receive faxes and electronic mail. Although limited commercial adoption prevented it from achieving widespread impact, the Simon Personal Communicator is widely regarded as the conceptual forerunner of the modern smartphone.

The mobile industry underwent a fundamental transformation in 2007 with the introduction of Apple's iPhone. Combining a capacitive touchscreen interface, full web browsing capability, and support for third-party application integration, the iPhone redefined user expectations for mobile devices and established a design and functional paradigm that has since governed smartphone development [2].

In 2008, the HTC Dream became the first commercially available smartphone powered by Google's Android operating system, inaugurating an open and highly customizable mobile platform. The Android ecosystem subsequently grew into the world's most widely deployed mobile operating system, fundamentally altering the competitive landscape of the mobile industry.

By 2009, Samsung introduced the first smartphone incorporating an AMOLED (Active-Matrix Organic Light-Emitting Diode) display, delivering measurably superior color

reproduction and contrast ratios relative to conventional LCD technology. This advancement progressively established AMOLED as the prevailing display technology in high-end mobile devices.

The deployment of 4G LTE networks beginning in 2010 marked a further inflection point in mobile communication, enabling data transmission speeds sufficient to support high-definition video streaming, video conferencing, and latency-sensitive applications. This infrastructure rapidly became the global standard for mobile broadband connectivity.

In 2019, foldable smartphones entered the consumer market, introducing a form factor that allows the device display to fold, offering expanded screen real estate within a compact footprint. While adoption has remained limited due to elevated cost and early durability concerns, foldable devices represent a significant direction of ongoing innovation in mobile hardware design.

Today, mobile phones constitute an indispensable component of contemporary life, supporting communication, commerce, education, and entertainment on a global scale. Mobile technology has undergone a profound transformation over the past eight decades, evolving from large, vehicle-mounted radio systems to compact, multifunctional devices of remarkable capability. This trajectory of innovation shows no signs of abating, with emerging form factors, artificial intelligence integration, and successive network generations continuing to redefine the boundaries of mobile communication.

1.2.2 Co-evolution of Mobile Devices and Applications

We have already described the relationship between mobile phones and mobile applications in the previous section. The following discussion highlights their evolution over time and demonstrates the existence of this close relationship.

Mobile phones and mobile applications are strongly interconnected and cannot be considered independently. This strong connection allows them to reinforce and evolve together over time. In other words, the growth of mobile phones has driven the evolution of mobile applications, and vice versa.

Indeed, the expansion of mobile applications is closely linked to the evolution of mobile devices. As mobile phones have become ubiquitous in everyday life, and with the rapid growth of smartphones, mobile applications have become an essential part of daily routines. Mobile

applications are designed to work seamlessly with mobile devices by leveraging their features and capabilities to deliver enhanced user experiences.

Conversely, the development of increasingly sophisticated mobile applications has contributed to the evolution of mobile phones themselves. As applications have become more complex and demanding, mobile device manufacturers have responded by improving the hardware and software capabilities of their devices. This has led to the development of faster processors, higher-quality cameras, improved battery life, and more advanced sensors, all of which have enhanced the overall user experience of both mobile phones and mobile applications.

1.3 Mobile Operating Systems

Mobile operating systems play a central role in the functioning of mobile devices and mobile applications. A mobile operating system is the software platform that manages the hardware resources of a mobile device and provides the necessary services for running applications. It acts as an intermediary between the hardware components (processor, memory, sensors, communication modules) and the applications developed by programmers.

Unlike desktop operating systems, mobile operating systems are designed to operate under strict constraints, such as limited battery capacity, reduced memory, and variable network connectivity. As a result, they must be optimized for energy efficiency, responsiveness, and security while ensuring a smooth user experience.

Several mobile operating systems have emerged over the years, but only a few have achieved widespread adoption. These operating systems differ in terms of architecture, supported hardware, application ecosystems, and development tools.

1.3.1 Android Operating System

Android is a mobile operating system based on the Linux kernel, designed primarily for touch devices such as smartphones and tablets. The operating system was developed by Android Inc. which was founded in 2003 and acquired by Google in 2005. Google transformed Android into open-source software under the Apache license with the project name "Android Open-Source Project" (AOSP) and brought together several partners around Android called the "Open Handset Alliance" (OHA) to develop open standards for mobile devices. [3], [4]

Today, Android is the world's most widely used mobile operating system, with a large community of developers writing applications (or "apps") that extend the functionality of devices. Android has been the world's best-selling operating system for smartphones since 2011 and for tablets since 2013. As of January 1, 2021, the Google Play Store boasted over 3 million apps.[5]

Android supports a wide range of devices, including smartphones, tablets, smart TVs, smartwatches, and embedded systems. Applications developed for Android are distributed mainly through the Google Play Store, although alternative distribution platforms also exist.

In addition, Android offers a wide range of features for users, including customization, security, compatibility with thousands of different devices, and easy access to Google Play Store to download applications. What's more, Android also offers regular updates to improve the performance and security of its operating system.

1.3.2 iOS Operating System

iOS is a mobile operating system developed by Apple exclusively for its devices, such as the iPhone and iPad. Unlike Android, iOS is a closed-source platform, meaning that Apple maintains strict control over both the hardware and software components of its ecosystem [6], [7].

Applications for iOS are developed using specific tools and programming languages provided by Apple and are distributed through the Apple App Store. This controlled environment allows Apple to enforce strict quality, security, and performance standards.

iOS is known for its smooth user experience, strong security model, and tight integration between hardware and software. However, its closed nature limits customization compared to Android.

1.3.3 Harmony OS

Harmony OS is a distributed operating system developed by Huawei, designed to operate across a wide range of devices, including smartphones, tablets, smart TVs, wearables, and Internet of Things (IoT) devices. Unlike traditional mobile operating systems, Harmony OS adopts a unified and distributed architecture that enables seamless interaction between multiple devices [8], [9].

One of the main objectives of Harmony OS is to provide a consistent user experience across different device categories while optimizing performance and resource usage. Applications

developed for Harmony OS can be designed to adapt dynamically to various screen sizes and hardware configurations.

Harmony OS represents a modern approach to mobile and distributed operating systems and illustrates emerging trends in mobile computing, particularly in multi-device integration and ecosystem-oriented design.

1.4 Types of Mobile Applications

Mobile applications can be classified into several categories based on their development technologies, execution environment, and level of integration with the operating system. This classification helps developers choose the most appropriate approach according to the objectives of the application, performance requirements, and target users. [10], [11]

In general, mobile applications are divided into three main types: native applications, web applications, and hybrid applications. Each type has its own advantages and limitations.

1.4.1 Native Applications

Native applications are developed specifically for a particular mobile operating system using platform-specific programming languages, tools, and frameworks. For example, Android native applications are developed using Java or Kotlin and rely on the Android SDK, while iOS native applications are developed using Swift or Objective-C. [10], [11]

Native applications are compiled and executed directly by the operating system, allowing them to fully leverage device hardware and system services. This tight integration enables native applications to access advanced features such as sensors, cameras, background services, notifications, and system-level optimizations.

Native applications are particularly suitable for performance-critical applications such as games, multimedia applications, and applications that require deep integration with device hardware. But they require considerable resources and expertise to develop, as they must be written specifically for each platform and must adhere to platform-specific development guidelines and standards. In addition, native applications may not be compatible with older versions of the operating system, which can limit their scope and market potential.

1.4.2 Web Applications

Web applications are applications that run inside a mobile web browser and are developed using standard web technologies such as HTML, CSS, and JavaScript. These applications are accessed through a URL and do not require installation from an application store. [10], [11]

Web applications rely on the browser as their execution environment, which limits their access to device hardware and system services. However, modern web technologies and frameworks have significantly improved the capabilities of web applications, allowing them to provide interactive and responsive user interfaces.

However, web applications can also present certain limitations in terms of performance and functionality compared to native applications. For example, web applications may not be able to take full advantage of the device's hardware features, such as the camera or accelerometer. In addition, web applications can be slower than native ones due to the need to load data from the network.

Modern web technologies have led to the emergence of Progressive Web Apps (PWAs), which extend traditional web applications by providing features such as offline data caching, background synchronization, and basic push notifications. PWAs can also access certain device capabilities, including geolocation and camera services, through standardized web APIs.

Despite these improvements, PWAs still lack the deep system integration and full hardware access offered by native applications. They represent an intermediate solution that combines ease of deployment with enhanced user experience.

1.4.3 Hybrid Applications

Hybrid applications combine elements of both native and web applications. They are developed using web technologies but are packaged within a native container that allows them to be installed and distributed like native applications through app stores. [10], [11]

Hybrid applications typically use frameworks that provide a bridge between web code and native device features. This allows developers to reuse a large portion of their code across platforms while still accessing selected native functionalities.

Classic hybrid applications are based on web technologies (HTML, CSS, and JavaScript) executed within a native container using a WebView component. In this model, the application interface is rendered as a web page inside a native application. While this approach simplifies

cross-platform development, it may result in limited performance and a user experience closer to that of a web application.

Cross-platform native frameworks, which are sometimes grouped under hybrid approaches, follow a different strategy. Frameworks such as React Native and Flutter allow developers to use a single programming language while rendering real native user interface components. This approach offers near-native performance and a user experience comparable to fully native applications, while still benefiting from code reuse across platforms. [10], [11]

Although these approaches differ internally, they share a common objective: simplifying multi-platform mobile application development.

Table 1: Types of Applications: Advantages and Disadvantages

Applications Type	***Advantages***	***Disadvantages***
Native Apps	• High performance & optimal OS integration • Full access to device features & system services. • UX aligned with platform guidelines. • High security via platform validation.	• Requires significant resources & specialized expertise. • Platform dependency limits portability. • High costs for multi-platform deployment & maintenance.
Web apps / PWAs	• Cross-platform via standard web tech. • Lower development/deployment costs. • Accessible from any connected device. • PWAs support offline caching & push notifications.	• Lower performance than native apps. • Limited access to advanced device features. • Network dependence (except partial PWA support). • Weaker security model compared to native platforms.
Hybrid apps	• Code reuse across platforms. • Reduced development time and cost • Access to device features through native bridges • Near-native UI (e.g., React Native, Flutter)	• WebView-based apps suffer performance lags • Increased architectural complexity. • Potential cross-platform compatibility issues. • Cannot fully match specific OS optimizations.

1.5 Mobile Application Development Process

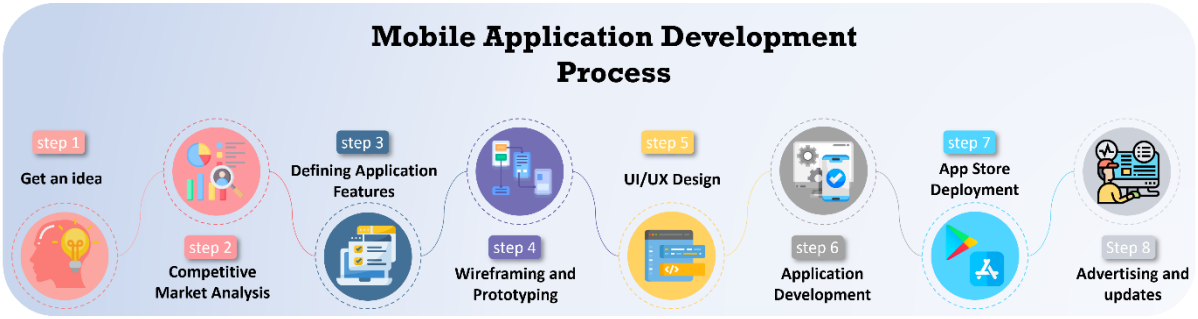

Figure 1: Mobile Application development process

Mobile applications are now an integral part of everyday life. From simple games to complex professional solutions, there is an application for almost every need. Creating a successful mobile application relies on a structured process composed of several key stages (Figure 1), ranging from idea generation to post-publication maintenance. [12]–[15]

This section presents the main steps of the mobile application creation process.

1.5.1 Idea Generation

Idea generation is the first step and one of the most important stages in the process. It consists of identifying a mobile application idea capable of solving a real problem or meeting a specific need for a given target audience. An application idea may relate to a specific or various domains such as healthcare, education, finance, gaming, entertainment, or services. [14], [15]

Sources of inspiration may include:

- Identifying a problem or unmet need in everyday life.
- Analyzing existing applications to improve or combine their features.
- Leveraging personal interests and passions while considering available skills and resources.
- Collecting feedback from friends, family, or potential users.

1.5.2 Competitive Market Analysis

Once the idea is defined, it is essential to conduct a competitive market study. This step involves analyzing existing similar applications and identifying market opportunities and gaps.

The market study helps to:

- Understand current trends.

- Identify popular and missing features.
- Evaluate monetization strategies.
- Analyze user reviews and ratings.

The main methods include:

- Exploring application stores (Google Play Store, Apple App Store)
- Direct analysis of competing applications.
- Conducting surveys or interviews with potential users.

1.5.3 Defining Application Features

Before starting development, it is necessary to clearly define the overall purpose and strategy of your app idea and design a prototype. This process can be called "business needs gathering". Take the time to clarify the details of your app idea and document them to avoid confusion later. [13]–[15]

This step includes:

- Dream: Take your app idea and imagine the perfect version, with no limits.
- Documentation: Write down the vision and objectives of your application idea to clarify your intentions.
- Gather requirements: Identify the overall objective and strategy of your application.
- MVP: Determine the essential features of a minimally viable product, which must nevertheless solve the general public's problem.
- User feedback: Launch the MVP to obtain real feedback from users and adapt your application according to this feedback.
- Repeat the cycle: Repeat the application update cycle, adding features and gathering user feedback until the product perfectly meets market needs.

The prototype makes it possible to visualize the application, validate usability, and gather early feedback before technical implementation begins.

1.5.4 Wireframing and Prototyping

Wireframing is a conceptual design activity that focuses on structure rather than appearance. It allows developers and designers to define how users will navigate through the application and how information will be organized on each screen. [12]–[15]

At this stage, visual simplicity is intentional. Wireframes are a communication tool that helps align technical teams and users with functionality and workflow before development begins. There are several tools that can help with wireframing, such as Sketch, Adobe XD or Figma.

The wireframing process typically includes:

- Defining the main screens and their roles within the application.
- Structuring content and interactions on each screen.
- Representing interfaces using basic shapes and placeholders.
- Annotating elements to clarify behavior and interactions.
- Refining page layouts gradually to ensure clarity and ease of use.

1.5.5 UI/UX Design

Graphic design transforms structural wireframes into a visually engaging product. This phase focuses on aesthetics, branding, and emotional appeal while maintaining usability and consistency.

A successful visual design enhances user experience by guiding attention, reinforcing brand identity, and improving readability. Design decisions should always support functionality rather than distract from it. [12], [14], [15]

The graphic design phase involves:

- Defining a consistent visual identity through colors (You can use websites like Coolors , MaterialPalette), typography, and layout rules.
- Selecting icons, images, and illustrations that align with the application's purpose. (You can use websites like FlatIcon, FreePik),
- Applying visual elements to wireframes to create complete screen designs.
- Testing visual choices with users to ensure clarity and accessibility.
- Refining the design to balance attractiveness, usability, and performance.

1.5.6 Application Development

This stage marks the transition from design to implementation. Application construction involves transforming concepts, wireframes, and designs into a working software product through coding and integration. [15]

Key activities during application construction include:

- Defining a technical architecture that supports application requirements.
- Preparing and configuring the development environment.
- Implementing functionalities according to specifications.
- Integrating components into a unified system.
- Conducting systematic testing to ensure reliability and correctness.

1.5.7 App Store Deployment

Applying to an app store is a formal process that ensures quality, security, and compliance with platform standards. The submission process generally involves: [16], [17]

- Registering as a developer on the chosen platform.
- Prepare the final application package and ensure that it meets all technical requirements.
- Provide the app store with all necessary information, including screenshots, descriptions and other metadata.
- Submitting the application for review.
- Addressing feedback or rejection if necessary.
- Publishing the application upon approval.

1.5.8 Marketing and Promotion Strategies

Marketing your app is a crucial step in ensuring its success and maximum exposure. A well-executed marketing plan can help you reach your target audience and attract more downloads, resulting in increased user engagement and revenue.

Effective marketing activities include:

- Identifying and understanding the target user profile.
- Promoting the application through social media channels.
- Collaborating with influencers or content creators.
- Optimizing store presence using keywords and visuals.
- Launching promotional campaigns to attract early users.
- Measuring performance and adjusting strategies accordingly.

1.5.9 User Feedback and Continuous Improvement

Application development does not end at release. Continuous improvement is essential to maintain relevance, satisfaction, and competitiveness.

This cycle includes:

- Collecting feedback from multiple sources.
- Identifying recurring issues and improvement opportunities.
- Prioritizing enhancements based on impact and feasibility.
- Implementing updates and corrections.
- Testing and validating changes before release.

> **Advice**
>
> In the wireframing phase, it's best to use Figma for its simplicity and functionality.
>
> For the choice of icons and images, it's best to use Freepik and Flaticon for their free license, so as not to infringe any copyright.
>
> For the graphic design and programming of the application, we recommend using Android Studio for Android applications, as this is the official IDE for Android development.
>
> Finally, the application will be the result of the previous steps, which means that the perfection of each step will result in an ideal application.

1.6 Essential Qualities of a Mobile Application

A successful mobile application is not defined solely by its functionality or the technologies used to build it. Beyond implementation, an application must satisfy a set of essential qualities that determine its acceptance, usability, and long-term success. [12], [15], [18]

1.6.1 Usability and User Experience (UX)

Usability refers to how easily users can learn, understand, and interact with an application, while user experience encompasses the overall feeling and satisfaction during its use. Mobile users typically interact with applications in short sessions and often in distracting environments, which makes simplicity and clarity essential.

To create a good user experience, it's important to consider the following:

- User interface (UI): The user interface (UI) is the part of the application that the user interacts with, such as buttons, menus and icons. It's important to design the UI to be intuitive, consistent and aesthetically appealing, so that users can easily navigate the application.
- Usability: which refers to the ease with which users can perform tasks within the application.
- Feedback: Feedback is important to ensure that users know what actions they have taken within the app and what the result has been.

1.6.2 Performance and Responsiveness

Performance is a critical factor in how users perceive application quality. Mobile applications are expected to respond immediately to user actions and to run smoothly, even on devices with limited hardware resources.

1.6.3 Reliability and Stability

Reliability refers to the ability of an application to function correctly under normal and unexpected conditions. A stable application behaves predictably, avoids crashes, and handles errors gracefully.

1.6.4 Security and Privacy

Mobile applications often access sensitive user data such as personal information, location, contacts, or camera input. Ensuring security and respecting user privacy are therefore essential qualities. [19]

To ensure sufficient security, it is important to consider the following elements:

- Data encryption: Encrypting data makes it unreadable for any user who does not possess the key to decode it.
- Authentication and authorization: verify the identity of users and ensure that they have the necessary authorization to access certain parts of the application.
- Vulnerability testing: testing the application for possible security vulnerabilities and correcting them before release.

1.6.5 Compatibility and Adaptability

Mobile ecosystems are highly diverse, with different screen sizes, hardware capabilities, and operating system versions. A quality mobile application must adapt to this diversity without compromising functionality or usability.

1.6.6 Maintainability and Evolvability

Mobile applications are rarely finished products; they evolve over time to include new features, fix bugs, and adapt to platform changes. Well-organized code, clear architecture, and good documentation help ensure that updates can be made efficiently and safely.

1.6.7 Energy Efficiency

Unlike desktop applications, mobile applications operate on battery-powered devices. Excessive energy consumption can quickly drain the battery and negatively impact user satisfaction.

1.6.8 Accessibility and Inclusiveness

Accessibility ensures that mobile applications can be used by people with different abilities and constraints. Designing accessible applications involves readable text, sufficient contrast, support for assistive technologies, and flexible interaction methods.

Together, these essential qualities define the overall quality of a mobile application. Addressing them early in the development process helps developers create applications that are not only functional but also robust, user-centered, and sustainable in real-world conditions.

1.6.9 Example: A Mobile Public Transportation Application

Consider a public transportation mobile application that allows users to check bus and tram schedules, plan routes, and receive real-time updates.

- From a usability and UX perspective, the application provides a simple home screen where users can quickly search for routes or nearby stations with minimal input.
- In terms of performance, schedules and maps load quickly, even on mid-range devices or slower networks.
- The application demonstrates reliability and stability by handling temporary network loss gracefully, displaying cached schedules when real-time data is unavailable.
- Regarding security and privacy, the app request’s location access only when necessary and clearly informs users why this permission is needed.

- Thanks to compatibility and adaptability, the interface adjusts to different screen sizes and remains usable across various Android versions and devices.
- The application is designed with maintainability and evolvability in mind, allowing developers to add new cities or transportation modes without rewriting the core system.
- It respects energy efficiency by limiting GPS usage and background updates when the application is not actively used.
- Finally, accessibility and inclusiveness are addressed through readable text, high-contrast colors, and support for assistive technologies.

Table 2: Key Quality Attributes in Application Design

Quality	Focus	Illustration in the Example
Usability & UX	Ease of use, intuitive interaction	Simple route search and clear navigation
Performance & Responsiveness	Speed, smooth interaction	Fast loading of schedules and maps
Reliability & Stability	Correct behavior, error handling	Offline access to cached schedules
Security & Privacy	Protection of user data	Controlled and justified location access
Compatibility & Adaptability	Device and OS diversity	Responsive layout across devices
Maintainability & Evolvability	Long-term evolution	Easy addition of new cities or services
Energy Efficiency	Battery preservation	Optimized GPS and background usage
Accessibility & Inclusiveness	Inclusive user experience	High contrast and assistive technology support

Chapter 2. Android Platform

Introduction

In the previous chapter, we explored the general concepts of mobile applications, their types, development process, and the essential qualities that define a successful mobile application. These concepts apply broadly across mobile platforms and technologies.

To move from general principles to practical implementation, it is now necessary to focus on a concrete and widely used mobile platform. This chapter introduces the Android platform and provides the architectural foundations required to understand how Android applications are structured and executed. Mastering these fundamentals is essential before addressing application behavior, life cycle management, and programming details in the following chapters.

Android is an open-source mobile operating system developed by Google [4] and based on the Linux kernel. It is designed primarily for touchscreen devices such as smartphones and tablets, but it is also used in other domains including smart TVs, wearable devices, and embedded systems. Due to its openness, flexibility, and large ecosystem, Android has become one of the most widely adopted mobile platforms worldwide.

From a developer's perspective, Android is more than just an operating system. It provides a complete application platform that includes development tools, system services, runtime environments, and application distribution mechanisms. These elements work together to enable the creation, execution, and maintenance of mobile applications.

One of the key characteristics of Android is its open ecosystem. Device manufacturers can customize the operating system, developers can freely build and publish applications, and users can choose from a vast range of devices and application sources. This openness contributes to Android's popularity but also introduces challenges such as device fragmentation and compatibility, which developers must take into account.

Android applications are typically distributed through application stores, most notably the Google Play Store, which provides a centralized platform for publishing, updating, and managing applications. The platform enforces certain technical and security requirements, helping to ensure application quality and user safety. Understanding Android as a platform involves recognizing its dual nature:

- as an operating system that manages hardware resources and system services,
- and as an application framework that defines how applications are built, interact with the system, and deliver functionality to users.

This foundational understanding is crucial for developers, as it influences design decisions, performance considerations, and the overall structure of Android applications. The next sections of this chapter will explore the internal architecture of the Android system and the core components that form an Android application.

Objectives

At the end of this chapter, students will be able to:

- Describe the layered architecture of the Android system, identifying the purpose of each layer and how they interact.
- Explain the relationship between hardware, the operating system, and Android applications, using the Android architecture as a reference model.
- Identify the core components of an Android application and describe the responsibility of each component within the application structure.
- Analyze how Android application components interact with the system framework to deliver functionality to the user.
- Develop a conceptual model of Android application execution, preparing for deeper discussions on life cycle management and application behavior in the following chapter.

2.1 Android System Architecture

The Android operating system is designed using a layered architecture, where each layer has a specific role and communicates with the layers above and below it. This architectural

organization improves modularity, security, and reusability, while allowing applications to run consistently across a wide range of devices.[3], [20]

Understanding this architecture is essential for mobile application developers, as it explains how applications interact with hardware resources, system services, and the runtime environment. Although developers do not usually interact directly with the lower layers, design and performance decisions at the application level are strongly influenced by the underlying system architecture. [3], [20]

The Android system architecture is commonly represented as a stack of five main layers are: the Linux kernel, the Hardware Abstraction Layer (HAL), the Native Libraries and Android runtime, the java API Framework and finally the Applications. Figure 2 illustrate the android platform architecture.

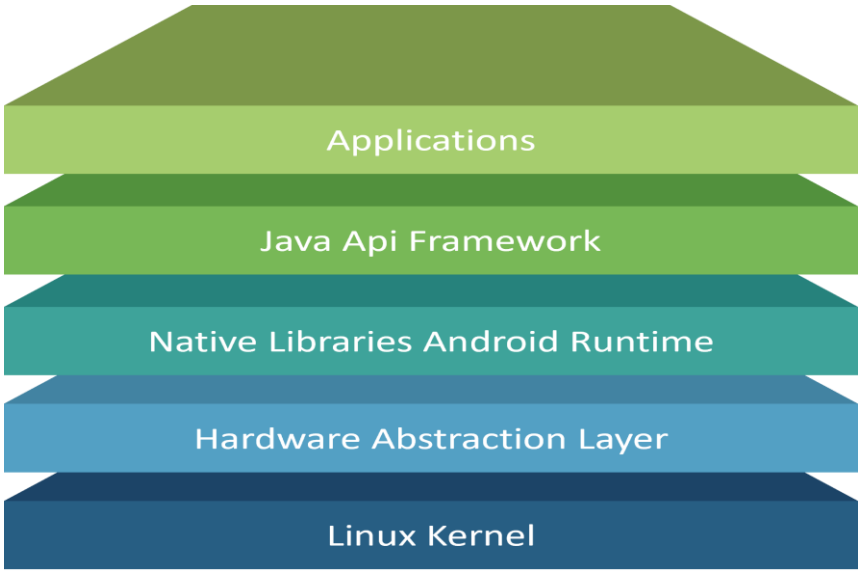

Figure 2: Android Platform Architecture

2.1.1 Linux Kernel

The Linux kernel is the core component of the Android operating system. It is responsible for managing fundamental system resources such as CPU, memory, and input/output devices. Acting as the interface between hardware and software, the kernel provides a stable and secure foundation on which the rest of the Android system operates.[3], [20]

Android uses a modified version of the Linux kernel that has been optimized for mobile devices. This kernel manages low-level hardware drivers and ensures efficient resource usage in environments with limited power and memory.

The Linux kernel in Android includes several key responsibilities:

- Process management: The kernel manages all running processes, allocates system resources, and ensures that applications execute smoothly without interfering with one another.
- Memory management: It controls memory allocation and deallocation, tracks memory usage per process, and prevents system instability due to memory exhaustion.
- Device drivers: The kernel provides drivers for hardware components such as the display, touchscreen, camera, sensors, and network interfaces.
- File system management: It ensures reliable storage and retrieval of data through file system management.
- Security: The Linux kernel enforces a strong security model, including process isolation and permission control, which protects the system from malicious or faulty applications.

2.1.2 Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) sits between the Linux kernel and higher-level software layers such as the Android Runtime and the Application Framework. Its purpose is to provide a standardized interface for accessing hardware components. [3], [20]

By abstracting hardware-specific details, the HAL allows device manufacturers to implement hardware support without modifying the Android framework itself. Each hardware category such as camera, audio, sensors, or Wi-Fi is represented by a dedicated HAL module.

When an application requests access to a hardware feature (for example, taking a photo or playing audio), the request is routed through the appropriate HAL module. The HAL handles communication with the physical hardware and returns results to higher-level layers.

This design enables developers to write hardware-agnostic applications that can run consistently across a wide variety of Android devices.

2.1.3 Native C/C++ Libraries

The Android system includes a set of native libraries written in C and C++, compiled as shared libraries (.so files). These libraries implement performance-critical functionalities and act as a bridge between the framework and the underlying hardware. [3], [20]

Native libraries are used for tasks that require low-level system access or high performance, such as:

- graphics rendering,
- audio and video processing,
- database management,
- encryption and security operations.

Examples of commonly used native libraries include:

- **OpenSSL:** for secure communications,
- **Zlib:** for data compression,
- **SQLite:** for lightweight database management,
- **SDL:** for multimedia and game development.

Android provides the Native Development Kit (NDK), which allows developers to write and integrate native C/C++ code into Android applications. To connect native code with Java-based application code, Android uses the Java Native Interface (JNI), which enables data exchange and function calls between the two environments.

While most applications rely primarily on Java or Kotlin, native libraries are essential for achieving optimal performance in specific use cases.

2.1.4 Android Runtime (ART)

The Android Runtime (ART) is responsible for executing application code and managing memory. Each Android application runs in its own runtime instance, ensuring isolation and security. [3], [20]

ART is the successor to the Dalvik virtual machine and was introduced to improve performance and efficiency. Unlike Dalvik, which relied mainly on Just-In-Time (JIT) compilation, ART uses Ahead-Of-Time (AOT) compilation, converting application bytecode into native machine code before execution. Modern versions of ART combine AOT and JIT techniques to further optimize performance. ART also provides:

- efficient garbage collection,

- improved memory management,
- advanced debugging and profiling tools.

In addition, ART supports the execution of native libraries, enabling applications to combine managed code with high-performance native components.

2.1.5 Java API Framework

The Java API Framework, often referred to as the Application Framework, is the layer most directly used by application developers [3], [20]. It provides a comprehensive set of Java classes, interfaces, and system services that simplify application development. This framework includes core components such as:

- Activities,
- Services,
- Broadcast Receivers,
- Content Providers.

It also offers APIs for common tasks including networking, file management, multimedia handling, graphics, and user interface design.

One of the key strengths of the Application Framework is its high level of abstraction. Developers do not need to manage low-level system details such as hardware access or memory handling. Instead, they interact with a consistent and unified API that remains stable across different Android versions and devices.

2.1.6 Application Layer

The Application Layer is the topmost layer of the Android architecture and represents what users directly interact with. It includes all applications installed on the device, whether they are pre-installed system applications or user-installed applications downloaded from app stores. [3], [20]

Each application runs in its own process and within a sandboxed environment. This isolation ensures that if one application crashes, it does not affect the stability of the entire system or other applications.

The application layer includes:

- core system applications (phone, messaging, camera),

- system-level applications (settings, system UI),
- third-party applications developed using the Android SDK.

Developers build applications using the Android Software Development Kit (SDK), primarily with Java (or Kotlin), and rely on the underlying architecture to access system services securely and efficiently.

Table 3: Android System Architecture

Layer	Main Role	Key Responsibilities	Developer Perspective
Linux Kernel	Core of the operating system	Process management, memory management, device drivers, file system, security	Ensures stability, isolation, and secure execution of applications
Hardware Abstraction Layer (HAL)	Bridge between hardware and software	Provides standardized interfaces to access device hardware (camera, audio, sensors, Wi-Fi)	Enables hardware-independent application development
Native C/C++ Libraries	High-performance system services	Graphics rendering, multimedia processing, databases, encryption	Used indirectly via framework APIs or directly via NDK for performance-critical tasks
Android Runtime (ART)	Application execution environment	Bytecode execution, memory management, garbage collection, code optimization	Influences application performance and responsiveness
Java API Framework	High-level development APIs	Activities, services, UI components, system services	Main interface used by developers to build Android applications
Application Layer	User-facing software	System apps and third-party applications	Where developers' applications run and interact with users

2.2 The core components of an Android app

An Android application is not a single executable program, but a collection of interacting components managed by the Android system. Each component plays a specific role and is designed to respond to system events or user actions. This component-based architecture allows Android applications to be flexible, reusable, and well-integrated with the operating system. [21]–[24]

Rather than controlling the entire execution flow, developers define application components and declare their capabilities. The Android system is then responsible for creating, managing, and coordinating these components as needed.

At the heart of this architecture are four core components.

2.2.1 Activities

In Android, an activity is a fundamental component of an application that represents a single screen with a user interface. Activities are used to build the interactive parts of an application that users can see and interact with. Each activity generally corresponds to a screen in the application, such as a login screen, a settings screen or a main screen. Activities are designed to be modular and reusable, so they can be combined in different ways to create different layouts and user experiences.[22]

An application may consist of one or multiple activities, each corresponding to a distinct task or screen. Activities can launch other activities, either within the same application or in another application, enabling seamless navigation and integration.

From an architectural perspective, activities serve as the entry points for user-driven interactions.

2.2.2 Services

In an Android application, a service is a component that can run in the background, even when the application is not being actively used. Services are used for tasks that need to run continuously in the background, such as playing music or downloading data. Services run on the application's main thread, but they can also run in a separate thread to avoid blocking the user interface.[25]

There are two types of service in Android: foreground services and background services. Foreground services have a visible notification that the user can see in the notification area. They are generally used for tasks that require the user's attention, such as playing music or recording audio. Background services have no visible notification and are used for tasks that do not require the user's attention, such as downloading data or processing data in the background.

Services can be started and stopped by other application components, such as activities or broadcast receivers. They can also be linked to other components to interact with them and share data. Services can be launched as long-term or one-time tasks, and can be stopped at any time by the system.

2.2.3 Broadcast Receivers

A Broadcast Receiver allows an application to respond to system-wide or application-specific events. These events, known as broadcasts, can originate from the system or from other applications.[26]

Examples of broadcasts include changes in network connectivity, battery level updates, or incoming messages. Broadcast receivers enable applications to react dynamically to changes in the system environment.

This mechanism promotes loose coupling between components and allows applications to integrate naturally with system events.

2.2.4 Content Providers

A Content Provider manages access to a structured set of data. It provides a standardized interface for storing and retrieving data, making it possible to share data between applications in a secure and controlled manner.[27]

Content providers are commonly used to access shared data such as contacts, media files, or application-specific databases. They enforce permissions and data access rules, ensuring data integrity and security.

From an architectural standpoint, content providers support data abstraction and interoperability between applications.

2.3 The Android SDK

The Android SDK (Software Development Kit) is a set of software tools and libraries provided by Google to enable developers to create and test Android applications. The Android SDK includes a complete set of development tools, such as a debugger, libraries, an emulator, documentation and code samples. [23]

The Android SDK provides the tools and resources needed to develop, debug and test Android applications. This includes a comprehensive set of development tools, such as Android Studio, which is the official integrated development environment (IDE) for Android application development. Android Studio offers features such as code completion, debugging and performance analysis tools, among others. [24]

In addition to development tools, the Android SDK includes a set of platform-specific libraries and APIs that enable developers to create a range of applications, from games and productivity tools to communication and social media apps. These libraries and APIs provide access to various Android functionalities, such as the camera, sensors, multimedia playback and network connectivity. [28]

The Android SDK also includes an emulator that enables developers to test their applications on virtual devices that emulate the behaviour of real devices. This enables

developers to test their applications on a range of devices and configurations, without the need for physical devices.

.

2.4 Tool Installation and Configuration

Installing and configuring the development tools is very simple: just install Android studio, which will automatically install all the tools (except java) (See the installation video in moodle).

Step 1: Download the latest version of Android Studio from the official website.

- Go to the official Android Studio website “https://developer.android.com/studio”.
- Click on the "Download" button to download the installer.
- Choose the Windows version if necessary.

Step 2: Run the downloaded installer.

- Open the downloaded installation file (e.g. android-studio-2023.1.1.0.exe).
- If you have any problems with the download, you can check the integrity of the file using its MD5 hash.

Step 3: If you're asked to choose the type of installation, select "Standard" and click "Next".

- This option will install Android Studio with the recommended components.

Step 4: Select the components you wish to install and click "Next".

- You can check/uncheck the components you want to install or not.

Step 5: Choose the installation location and click "Next".

- You can change the installation path if you wish.

Step 6: On the "SDK components configuration" screen, select "Recommended" and click "Next".

- This option will install the recommended SDK components.

Step 7: On the "Check settings" screen, click "Install".

Step 8: Android Studio will start installing the selected components. Wait for the installation to finish.

- This may take some time, depending on the speed of your computer, the speed of your connection and the number of components to be installed.
- Once installation is complete, click "Next", then "Finish".
- Android Studio is now installed on your computer.

You may encounter an error saying "No JVM installation found. Please install a 64-bit JDK. If you have already installed a JDK, set a JAVA_HOME variable".

To resolve this error, you need to install a JDK (Java Development Kit) and set the JAVA_HOME environment variable. To do this, follow these steps:

1. Go to the official Java SE download page (https://www.oracle.com/java/technologies/javase-downloads.html).
2. Download the JDK for Windows, taking care to select the appropriate version (32-bit or 64-bit) for your system.
3. Run the downloaded installer and follow the prompts to complete the installation.
4. Once installation is complete, open the Windows Control Panel, go to "System and Security" > "System" > "Advanced System Settings".
5. In the System Properties window, click on the "Environment Variables" button.
6. In the "System Variables" section, click on "New" to create a new environment variable.
7. Enter "JAVA_HOME" as the variable name and the path to the JDK installation directory as the variable value. For example, if you installed the JDK in the default location, the value should be "C:\Program Files\Java\jdk1.8.0_281".
8. In the same interface, search for the "Path" variable and add the following value: "%JAVA_HOME%\bin".
9. Click "OK" to save the environment variable.
10. Close all open windows and restart Android Studio.

2.5 Create an Android Emulator

Android Emulator offers the possibility of simulating different Android devices on a computer in order to test applications on various versions of Android without the need to own each physical device. It allows great flexibility in simulating various device configurations,

such as phones, tablets, connected watches and Android TVs. What's more, the emulator offers almost all the functionality of a real Android device, such as the ability to make calls and SMS, simulate location, network, rotation and other hardware sensors, and access the Google Play Store. Finally, testing your application on the emulator is faster and easier than on a physical device, not least thanks to the ability to transfer data to the emulator faster than to a USB-connected device [28]. Creating and running an emulator is straightforward, and involves the following steps:

Open Android Studio. If you haven't already installed Android Studio, follow the steps described in the previous section to download and install it.

1. Click on "AVD Manager" in the toolbar or go to "Tools" > "AVD Manager".
    - The AVD Manager (Android Virtual Device Manager) lets you create, configure and manage Android emulators.
2. Click on "Create Virtual Device".
    - This option lets you create a new emulator.
3. Select the type of Android phone or tablet you wish to emulate.
    - You can choose from several options, such as Nexus, Pixel or Android TV.
4. Click on "Next".
5. Select the Android version you wish to emulate.
    - You can choose from the most recent Android versions, as well as some older ones.
6. Configure your emulator settings.
    - You can modify parameters such as screen size, resolution, storage, RAM and operating system.
7. Click on "Finish".
    - You have now created a new Android emulator.
8. To launch the emulator, click on the "Play" button next to the emulator in the AVD Manager.
    - The Android emulator will start running. Please be patient, as this may take some time depending on your computer's performance.

Chapter 3. Activities and Resources

Introduction

To create an Android application, we use various components such as activities, views, fragments, services and content providers. These components are organized and managed by the Android operating system. This chapter presents these different components and their interactions within an Android application.

Objectives

The objective of this chapter is to provide a comprehensive overview of the **fundamental building blocks and best practices** related to Activities, resources, the R class, and the Android manifest in Android application development.

At the end of this chapter, students will be able to:

- **Explain the role, structure, and organization of Activities** within an Android application.
- **Describe the Activity lifecycle**, including creation, execution, interruption, and destruction phases, and explain why lifecycle management is essential for application performance and reliability.
- **Demonstrate how Activities are created, launched, and managed**, following best practices to ensure correct behavior and optimal performance.
- **Understand the purpose and structure of the Android Manifest file**, and identify the key elements it contains.
- **Explain how the Android system uses the manifest file** to identify, configure, and launch an application.

- **Apply good practices for managing permissions and application characteristics** through the manifest file.
- **Explain the role and importance of resources** in Android development and identify the different types of resources used in an application.
- **Organize and access application resources efficiently**, with an emphasis on maintainability, performance, and adaptability.
- **Explain the purpose of the R class**, its role in Android applications, and how it is automatically generated during the build process.
- **Demonstrate how to use the R class** to access different types of resources in a structured and efficient manner.

3.1 Introduction to Activities

In Android development, activities represent the core interaction points between the user and the application. An activity typically corresponds to a single screen with a user interface, such as a login screen, a list of items, or a settings page. Understanding activities is fundamental because they define how users navigate through an application and how the system manages application state. [22], [29]

From a system perspective, activities are managed by the Android operating system, not directly by the developer. This means that developers must design activities in a way that respects system constraints such as limited memory, multitasking, and configuration changes (e.g., screen rotation).

The primary purpose of an activity is to:

- Display a user interface to the user
- Handle user interactions (touch, input, navigation)
- Act as an entry point for user-driven application behavior

Each activity focuses on one well-defined task from the user's point of view. Complex applications are therefore composed of multiple activities, each responsible for a specific function. This separation improves clarity, maintainability, and reuse of application components.

3.1.1 Activity Life Cycle

The life cycle of an Android activity comprises several stages, corresponding to different events that occur during the execution of the activity. The activity life cycle begins when it is launched by the user and ends when the activity is closed or destroyed by the operating system.[24], [29], [30]. Figure 3 Present the activity life cycle and its methods.

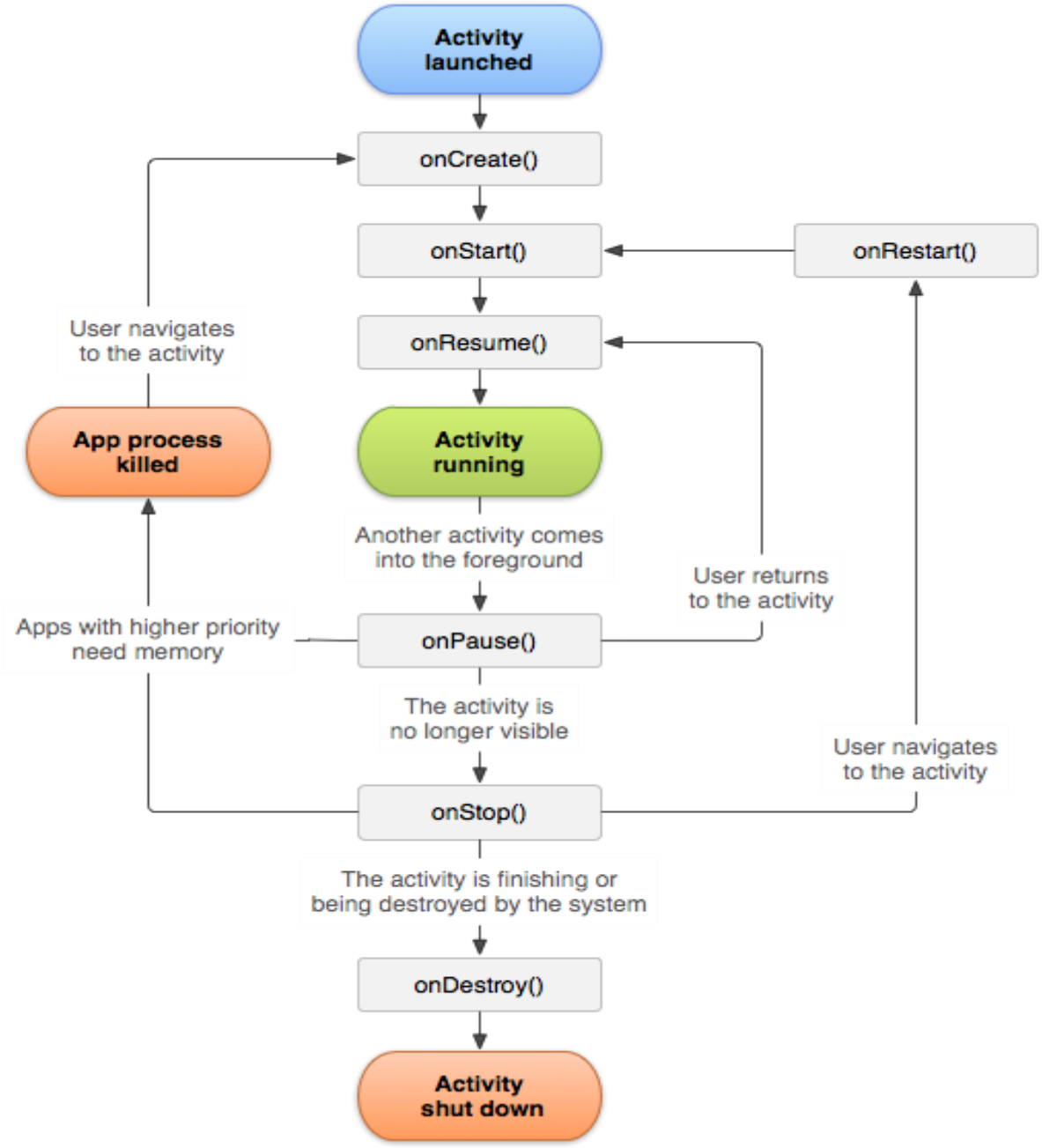

Figure 3: Activity Life Cycle

In Android, activities are managed using a stack, which means that the current application is first on the stack. An activity can be in one of four states:

- Active or running: The activity is in the foreground, i.e. at the top of the activity stack, interacting with the user.
- Paused: The activity is still visible to the user but is no longer in the foreground.
- Stopped: The activity is no longer visible to the user and is in the background.
- Destroyed: The activity has been removed from the activity stack and its resources released.

In general, switching between the life cycle states of an activity is handled by the following callback methods:

1. onCreate() - This method is called when the activity is created. It is generally used to perform initial activity configuration tasks, such as setting up the user interface, initializing variables or loading data from a database. This is also when you call setContentView() to define the activity's layout.

```
1. @Override
2. protected void onCreate(Bundle savedInstanceState) {
3.     super.onCreate(savedInstanceState);
4.     setContentView(R.layout.activity_main);
5.     // perform necessary setup tasks here
6. }
```

2. onStart() - This method is called when the activity becomes visible to the user.

```
1. @Override
2. protected void onStart() {
3.     super.onStart();
4.     // start background tasks or connect to services here
5. }
```

3. onResume() - This method is called when the activity is in the foreground and has focus.

```
1. @Override
2. protected void onResume() {
3.     super.onResume();
4.     // resume animations or update UI here
5. }
```

4. onPause() - This method is called when the activity loses focus and is no longer in the foreground.

```
1. @Override
2. protected void onPause() {
3.     super.onPause();
4.     // stop animations or release resources here
5. }
```

5. onStop() - This method is called when activity is no longer visible to the user.

```
1. @Override
2. protected void onStop() {
3.     super.onStop();
4.     // release resources or stop background tasks here
5. }
```

6. onRestart() - This method is called when the activity is restarted after being stopped.

```
1. @Override
2. protected void onRestart() {
3.     super.onRestart();
4.     // reset UI components or reload data here
5. }
```

7. onDestroy() - This method is called when the activity is about to be destroyed.

```
1. @Override
2. protected void onDestroy() {
3.     super.onDestroy();
4.     // release any remaining resources or unregister listeners here
5. }
```

Note

The onPause() and onResume() methods are very important for managing system resources, such as battery life and memory usage. You should always release resources you no longer need in the onPause() method to save system resources, and reacquire them in the onResume() method when activity resumes.

It is also important to consider the impact of life cycle calls on the user experience. Lengthy operations in onResume() can lead to a lack of responsiveness in the application and therefore a poor user experience.

Warning

Be aware of the risks involved in using long-running operations in onStart() or onStop(), as these methods may be called frequently during the lifetime of an activity.

Be aware of the risk of bugs or unexpected behavior when using complex or third-party libraries that may not be fully compatible with the Android life cycle.

2.2. Navigation Between Activities

In Android, an activity is a self-contained screen with a user interface. To create a functional application, it may be necessary to use several activities or a single activity with several fragments (we'll look at these in later chapters). When using multiple activities, you need to navigate between them to provide the required functionality. This navigation is generally done using intents. [22]–[24]

Intents are objects that allow you to specify an action to be performed, such as starting a new activity. There are two types of intention: explicit and implicit. Explicit intentions are used to specify the target activity class and start a new activity within the same application. Implicit intentions are used to perform an action that can be handled by an activity in a different application, such as opening a web page in the user's preferred browser. [30], [31]

Intents are objects that allow you to specify an action to be performed, such as starting a new activity. There are two types of intention: explicit and implicit.

1. **Start a new activity (using explicit intent)**

To start a new activity, you must first create an instance of the Intent class and pass the context of the current activity and the class of the target activity as parameters. Then, you can call the startActivity() method on the current activity and pass the intent as a parameter. Here's an example of code :

```
1. Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);
2. startActivity(intent);
```

2. **Start a new activity for a result (using explicit intent)**

If you need to obtain a result from an activity, you can use the startActivityForResult() method instead of startActivity(). This method allows you to pass an intent and request code to the target activity, and the target activity returns a result to the main activity by calling setResult() and finish(). Here's an example code:

Main activity code

```
1. Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);
2. int REQUEST_CODE= 1;
3. startActivityForResult(intent, REQUEST_CODE);
4.
5. // Override the onActivityResult() method to handle the result
6. @Override
7. protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
8.     super.onActivityResult(requestCode, resultCode, data);
9.
```

```
    if (requestCode == REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            // Handle the successful result
        } else if (resultCode == RESULT_CANCELED) {
            // Handle the canceled result
        }
    }
}
```

Target activity code

```
.....
// perform necessary logic here
.....
Intent resultIntent = new Intent();
resultIntent.putExtra("parameter name", parameter value);
setResult(Activity.RESULT_OK, resultIntent);
finish();
```

3. Start a new activity using implicit intents

To start a new activity using an implicit intent, you need to create an instance of the Intent class and specify the action to be performed as well as any additional data. For example, to open a web page in the browser, you can create an Intent with the ACTION_VIEW action and the URL as data. Then you can call the startActivity() method on the current activity and pass the intent as a parameter. Here's an example code:

```
Intent intent = new Intent(Intent.ACTION_VIEW,
Uri.parse("https://www.example.com"));
startActivity(intent);

```

Other examples of implicit intent actions:

- ACTION_EDIT: modifies the data identified by the intent.
- ACTION_SEND: Sends data to another application, such as a messaging or e-mail application.
- ACTION_PICK: Allows you to select an item from a list, such as a contact or photo.
- ACTION_SEARCH: Allows you to perform a search on the data identified by the intent.
- ACTION_WEB_SEARCH: Performs a web search using the given query.
- ACTION_DIAL: Allows you to dial a telephone number.
- ACTION_CALL: Initiates a phone call.
- ACTION_GET_CONTENT: Retrieves the contents of a file or document.
- ACTION_SENDTO: Sends a message to a specific phone number or e-mail address.
- ACTION_VIEW_DOWNLOADS: displays the list of downloads.
- ACTION_SYNC: performs a synchronization operation.
- ACTION_INSTALL_PACKAGE: Installs a package.
- ACTION_UNINSTALL_PACKAGE: to uninstall a package.

4. Back Button

When the user presses the Back button, the current activity is completed and removed from the activity stack, and the previous activity on the stack is resumed. This is the default behavior of the Back button, and you don't need to write any code to implement it. If you wish to customize this behavior, you can override the onBackPressed method in your

activity. It's important to note that when you start a new activity, the previous activity is paused but remains in the activity stack. You can use the "back" button to return to the previous activity or call the "finish()" method of the current activity to remove it from the stack.

Table 4: Activity Class Methods

Related to	*Method declaration*	*Description*
Life cycle	onCreate(Bundle)	Called when the activity is created.
	onStart()	Called when the activity is about to become visible.
	onResume()	Called when the activity has become visible (foreground).
	onPause()	Called when the activity is about to go into the background.
	onStop()	Called when the activity is no longer visible.
	onRestart()	Called after the activity has been stopped and is about to restart.
	onDestroy()	Called before the activity is destroyed.
	onSaveInstanceState (Bundle)	Called before the activity is stopped, saves the state of the activity in a bundle.
	onRestoreInstanceSt ate(Bundle)	Called when the activity is being restored, retrieves the state saved in the bundle.
Interface	setContentView(int)	Creates the user interface from an XML layout file specified by the resource identifier parameter.
	findViewById(int)	Retrieves a view element from the user interface. Returns a View object. The parameter is the view element ID (normally defined in the R class).
	showDialog(int)	Opens a dialog window. The parameter is the dialog ID.
	showDialog(int, Bundle)	Opens a dialog window. The first parameter is the dialog ID, the second allows parameters to be passed to the dialog.
	dismissDialog(int)	Close a dialog specified by its ID.
	onCreateDialog(int, Bundle)	Called when a dialog is opened. The first parameter is the dialog ID, the second is the data transmitted to it.
	onPrepareDialog(int , Bundle)	Called when a dialog that is already open is activated. The first parameter is the dialog ID, the second is the data passed to it.
Resources	getResources()	Returns the Resources object associated with the activity.
	getString(int)	Retrieves a string resource with the given ID.
Starting activities	startActivity(Intent)	Starts a new activity specified by an explicit intent.
	startActivityForResu lt(Intent, int)	Starts a new activity and waits for a result. The first parameter is the intention used to start the activity, the second is a code identifying the request.
	startActivityIfNeede d(Intent, int)	Starts a new activity only if it is not already running in the current task.
	startActivityFromCh ild(Activity, Intent, int)	Starts a new activity from a child activity. The first parameter is the child activity that starts the new activity.

3.2 The Android Manifest File

The AndroidManifest.xml file is a mandatory configuration file for every Android application. It acts as the official declaration of the application to the Android operating system and is examined by the system before the application is installed or launched. [23], [24], [30], [32]

Android relies on this file to understand the overall characteristics and capabilities of the application, including:

- The structure of the application
- The list of application components
- The permissions required to access system or hardware features
- The minimum and target SDK versions
- The hardware and software features required by the application

Without a correctly configured AndroidManifest.xml file, the application cannot be installed, launched, or executed properly.

Below is the general structure of the AndroidManifest.xml file

```
 1. <?xml version="1.0" encoding="utf-8"?>
 2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
 3.     package="com.example.myapp">
 4.
 5.     <uses-permission android:name="android.permission.INTERNET" />  <!--
Permission declaration -->
 6.
 7.     <application
 8.         android:allowBackup="true"
 9.         android:icon="@drawable/app_icon"
10.         android:label="@string/app_name"
11.         android:supportsRtl="true"
12.         android:theme="@style/AppTheme">
13.
14.         <!-- Activity declaration -->
15.         <activity
16.             android:name=".MainActivity"
17.             android:label="@string/app_name"
18.             android:theme="@style/AppTheme.NoActionBar">
19.
20.             <intent-filter> <!--Entry Point Definition-->
21.                 <action android:name="android.intent.action.MAIN" />
22.
23.                 <category android:name="android.intent.category.LAUNCHER"
/>
24.             </intent-filter>
25.         </activity>
26.
27.       <!-- Service declaration -->
28.         <service
```

```
            android:name=".MyService"
            android:exported="false">
            <intent-filter>
                <action android:name="com.example.myapp.MyService" />
            </intent-filter>
        </service>
      <!-- Broadcast declaration -->
        <receiver
            android:name=".MyReceiver"
            android:enabled="true"
            android:exported="false">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"
/>
            </intent-filter>
        </receiver>
      <!-- Content Provider declaration -->
        <provider
            android:name=".MyContentProvider"
            android:authorities="com.example.myapp.MyContentProvider"
            android:exported="false" />

    </application>

</manifest>
```

3.2.1 Intent Filter

An “intent filter” is a powerful Android feature that allows an application to declare the types of “intents” it can handle. This filter is defined in an application's manifest file and specifies the type of request the component wishes to receive. This filter can be seen as a description of a component's ability to handle specific types of commands.

When an implicit intent is triggered, the Android system checks the intent filter of each component to see if any of them can handle the request. If more than one component can handle the request, the user is presented with a list of options to choose from. This allows flexibility and customization within the Android ecosystem.

The following “intent filter” is used to specify that the main activity (MainActivity) is the application's main entry point

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3.2.2 Activity

The <activity> tag describes each activity existing in the Android application. The basic information about the main activity is generally defined as follows:

- android:name specifies the name of the Java class that implements the activity.
- android:label defines the displayed name of the application.
- <intent-filter> describes the types of intent that this activity can handle, for example, the intent to launch the application as the main activity.

```
1. <activity
2.     android:name=".MainActivity"
3.     android:label="@string/app_name">
4.     <intent-filter>
5.         <action android:name="android.intent.action.MAIN" />
6.         <category android:name="android.intent.category.LAUNCHER" />
7.     </intent-filter>
8. </activity>
```

3.2.3 Service

The <service> tag defines a service within the application. It contains information about the service, such as its name and intent filters. Here's an example:

```
1. <service
2.      android:name=".MyService"
3.      android:exported="false">
4.      <intent-filter>
5.          <action android:name="com.example.myapp.MY_ACTION" />
6.      </intent-filter>
7. </service>
```

3.2.4 Broadcast Receiver

A broadcast receiver is defined by the <receiver> tag in the manifest file. It contains information about the receiver, such as its name and intent filters. Here's an example:

```
1. <receiver
2.      android:name=".MyReceiver">
3.      <intent-filter>
4.          <action android:name="android.intent.action.BOOT_COMPLETED" />
5.      </intent-filter>
6. </receiver>
```

3.2.5 Content Provider

A content provider is defined by the <provider> tag in the manifest file. It contains information about the provider, such as its name and intent filters. Here's an example:

```
<provider
     android:name=".MyProvider"
     android:authorities="com.example.myapp.provider"
     android:exported="false">
</provider>
```

3.2.6 Permissions

In Android, permissions are used to protect sensitive information and system resources from unauthorized access. Permissions are declared in the AndroidManifest.xml file and must be requested by the application at runtime. The Android platform uses a permissions system to ensure that only trusted applications can access certain features or data on the device.

To declare a permission in the AndroidManifest.xml file, the <uses-permission> tag is used. The android:name attribute is used to specify the name of the permission. The most common permissions are INTERNET, CAMERA, READ_CONTACTS and WRITE_EXTERNAL_STORAGE.

For example, to request permission to use the device's camera, the following code can be added to the AndroidManifest.xml file:

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

Table 5 Android Permissions

Permission	***Description***
android.permission.CAMERA	Allows the application to access the camera to take photos or record video.
android.permission.ACCESS_FINE_LOCATION	Allows the application to access the device's precise location using GPS or other location sources.
android.permission.READ_CONTACTS	Allows the application to read contacts stored on the device.
android.permission.WRITE_EXTERNAL_STORAGE	Allows the application to access the device's external storage space to read, write and delete files.
android.permission.RECORD_AUDIO	Allows the application to record audio using the device's microphone.
android.permission.READ_CALENDAR	Allows the application to read calendar events stored on the device.
android.permission.SEND_SMS	Allows the application to send text messages.
android.permission.CALL_PHONE	Allows the application to make phone calls without user intervention.
android.permission.INTERNET	Allows the application to access the Internet.

android.permission.ACCESS_NETWORK_STATE	Allows the application to find out whether the device is connected to the Internet and to check the status of the network connection.

3.3 Resources and the R Class

Introduction

In Android development, resources refer to all non-code assets used by the application. Instead of hard coding values directly inside Java files, Android encourages developers to externalize UI elements, strings, images, layouts, and configuration values into dedicated resource files.[23], [24], [30], [33]

This separation between code and resources provides better organization, easier maintenance, improved localization, reusability, and automatic device adaptation.

Resource Directories

Resources in Android are divided into several directories:

- /res/layout: This directory contains the XML files that describe the application's user interface.
- /res/values: This directory contains the XML files that define the values of resources that are used throughout the application, such as strings, colors, dimensions, styles, themes, etc.
- /res/drawable: This directory contains the image and graphics files.
- /res/mipmap: This directory contains application icons.
- /res/raw: This directory contains raw resource files, such as audio, video, JSON.
- /res/xml: This directory contains XML files defining custom resources.
- /res/menu: This directory contains XML files defining the menus used in the application.

3.3.1 Value Resource

Value resources are used to define various types of data, such as strings, colors, dimensions, integers and arrays, which can be used throughout the application.

a. Strings

To define a static string resource in XML, you need to create a new entry in the strings.xml file located in the values directory of your Android project. By keeping text out of your Java code and layout files, you can update your content in one place or easily add support for multiple languages by creating additional strings.xml files for different regions.

Here's an example of how to define a string resource:

```
<!-- strings.xml -->
<resources>
    <string name="hello_world">Hello World!</string>
</resources>
```

To use a string resource in XML, simply reference the string resource using the syntax "@string" followed by the name of the string resource. Here's an example:

```
<!-- activity_main.xml -->
<TextView
    android:id="@+id/hello_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
```

To use a string resource in Java, you can access it using the generated R class. Here's an example:

```
// MainActivity.java
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Get the string resource
        String hello = getString(R.string.hello_world);

        // Use the string resource
        Toast.makeText(this, hello, Toast.LENGTH_SHORT).show();
    }
}
```

Static string arrays are also can be defined in the strings.xml file as follows:

```
 <resources>
    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>Earth</item>
        <item>Mars</item>
        <item>Jupiter</item>
        <item>Saturn</item>
        <item>Uranus</item>
        <item>Neptune</item>
    </string-array>
</resources>
```

String arrays are used in XML layout files as follows:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@array/planets_array" />
```

String arrays can be also used in java code as follows:

```
String[] planets = getResources().getStringArray(R.array.planets_array);
```

b. Styles

The styles are defined in a file called styles.xml which exists in the directory res/values. In this file, you can define any component style, from the TextView text size to the background color and elevation of a Material Card. This allows you to centralize your design attributes and apply them to multiple UI elements at once, ensuring a professional and uniform appearance. The code below illustrates a style that changes text size and color.

```
<style name="MyStyle">
    <item name="android:textSize">20sp</item>
    <item name="android:textColor">#FF0000</item>
</style>
```

To use this style in your XML layout file, you simply reference the name:

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/MyStyle"
    android:text="Hello World!" />
```

You can also use this style in your java code, you uses the resource name:

```
TextView textView = findViewById(R.id.textView1);
textView.setStyle(R.style.MyStyle);
```

c. Colors

The color resources are defined in a file called colors.xml located in the res/values directory. In this file, you can define any specific color used in your application, from the main toolbar primary color to the specific hex code for a button's ripple effect. By referencing these colors by name throughout your layouts, you ensure visual consistency and make it simple to implement themes like Dark Mode. You can define colors using Hexadecimal (HEX) values:

```
<resources>
    <color name="primary_blue">#00574B</color>
    <color name="accent_orange">#FF9800</color>

```

```
    <color name="error_red">#D32F2F</color>
    <color name="background_gray">#F5F5F5</color>
    <color name="text_secondary">#757575</color>
</resources>
```

Once defined, you can apply these colors to any UI component in your XML layout files:

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Submit"
    android:backgroundTint="@color/primary_blue"
    android:textColor="@color/background_gray" />
```

You can also retrieve these colors programmatically in your Activity:

```
int myColor = getResources().getColor(R.color.accent_orange, getTheme());
myTextView.setTextColor(myColor);
```

3.3.2 The R Class

The R class is an automatically generated class that provides references to all application resources. It acts as a bridge between Java code and XML resources. Every resource defined inside the res/ directory receives a unique identifier in the R class.[23], [24], [30], [33]

During the compilation process, the Android build system scans the res/ directory and assigns a unique integer ID to every resource found. These IDs are then used to generate the R.java file (or the compiled R class), which acts as a bridge allowing the application to access resources through a structured Java class. However, because this generation is dependent on the integrity of your resources, any error within an XML file can cause the R class to fail to generate. This prevents the project from compiling and explains the common “Cannot resolve symbol R” error, as the compiler cannot find the class that was supposed to be created.

To visualize how these IDs are organized, here is a simplified example of what the generated class looks like:

```
public final class R {
    public static final class anim {
        public static final int fade_in = 0x7f050000;
        public static final int fade_out = 0x7f050001;
    }
    public static final class drawable {
        public static final int ic_launcher = 0x7f020000;
        public static final int logo = 0x7f020001;
    }
    public static final class id {
        public static final int button_ok = 0x7f060000;
        public static final int button_cancel = 0x7f060001;
    }
```

```
14.     // more inner classes for other resource types
15. }
```

Chapter 4. Graphic Interfaces and Widgets

Introduction

Graphical user interfaces (GUIs) play a crucial role in Android application development, providing a user-friendly and intuitive experience. Users can interact with the application via buttons, text fields, drop-down lists, checkboxes and other widgets.

Views and widgets are essential for creating graphical interfaces for Android applications. Although the terms “views” and “widgets” are sometimes used interchangeably, views are generally considered to be the fundamental components of the user interface. Widgets, on the other hand, are primarily visual UI elements and are placed on the application screen.

This chapter explores the fundamental concepts of interfaces and widgets in Android development, including the View class hierarchy and the hierarchy of views in Android application interfaces. It covers different layouts and how to optimize them for performance, as well as fragments and their lifecycle methods. The chapter also explores the different types of widgets available, from basic UI elements such as buttons and TextViews to more advanced widgets such as RecyclerViews and ListViews. Tips and best practices for optimizing performance and enhancing the user experience are provided throughout the chapter.

Objectives

The objective of this chapter is to introduce the principles, structure, and practical implementation of graphical user interfaces (GUI) in Android applications. While previous chapters focused on architecture, application components, and life cycle management, this chapter shifts toward visual construction and user interaction, which are essential for creating usable and responsive mobile applications. Android applications are fundamentally interactive systems. Their success depends not only on functionality, but also on how effectively users can

perceive, understand, and manipulate the interface. For this reason, developers must understand both the technical structure and the design logic behind Android user interfaces.

- By the end of this chapter, students should be able to:
- Understand how Android represents graphical elements through the View system
- Explain the hierarchical structure of UI components
- Design user interfaces using XML layout mechanisms
- Apply different layout models to organize interface elements
- Use fragments to build modular and reusable interface sections
- Integrate widgets to support user interaction
- Connect interface elements with application logic through event handling

4.1 Introduction to User Interfaces

User interfaces are the visible and interactive layer of an Android application. They represent the point of communication between the user and the system, allowing information to be displayed and actions to be performed. A well-designed interface is not only visually appealing, but also structured, responsive, and easy to navigate.[12], [23], [24], [30]

In Android, user interfaces are built around a component-based graphical model. Every visual element that appears on the screen (e.g. text, buttons, images, or containers) is represented internally as an object derived from the View class. These objects are organized into hierarchical structures that determine how elements are displayed, positioned, and interacted with.

Understanding this structure is essential because Android does not treat the interface as a simple visual layer. Instead, the UI is tightly integrated with:

- The activity life cycle
- Event handling mechanisms
- Resource management
- System rendering processes

This integration allows Android to dynamically adapt interfaces to different screen sizes, orientations, and device configurations.

Another important characteristic of Android UI design is the separation between presentation and logic. Interface structure is typically described using XML layout files, while behavior is implemented in Java code. This separation improves readability, maintainability, and collaboration between designers and developers.

From a conceptual standpoint, Android interfaces are based on three foundational ideas:

- **Encapsulation**: each UI element is a reusable object
- **Hierarchy**: UI elements are arranged in parent–child structures
- **Event-driven interaction**: user actions trigger system responses

Together, these principles enable developers to construct interfaces that are flexible, scalable, and efficient.

The following sections explore these foundations in detail, beginning with the fundamental building block of the Android interface system: Views and widgets.

4.1.1 Views and Widgets

In Android, every graphical element displayed on the screen is derived from the View class. A View represents a rectangular area responsible for drawing content and responding to user interaction. Views serve two main purposes:

- Rendering visual content (text, images, shapes)
- Handling user input (touch, clicks, gestures)

A widget is a specialized type of View designed for user interaction. Examples include:

- TextView — displays text
- Button — triggers actions
- EditText — accepts user input
- ImageView — displays images

Widgets extend the base View functionality by adding predefined behaviors and properties, allowing developers to quickly construct interactive interfaces.

From a developer's perspective, Views and widgets are typically declared in XML layout files:

```
1. <TextView
2.     android:id="@+id/title"
3.     android:layout_width="wrap_content"
```

```
4.      android:layout_height="wrap_content"
5.      android:text="Welcome" />
```

The system interprets this XML structure and converts it into actual View objects during runtime.

> **Warning**
>
> The term "widget", in the context of Android development, refers specifically to user interface elements that can be added to an application's user interface, such as buttons, text fields and images.
>
> On the other hand, the term "widget" can also refer to small applications that offer specific functionality on the user's home screen or lock screen, such as weather widgets, clock widgets, music player widgets.

4.1.2 Android UI View Hierarchy

The View hierarchy (Figure 4) in an Android application interface represents the structure of all Views that are currently visible on an application's screen. Essentially, it's a tree structure where each node represents a View or View group, this structure defines:

- Parent–child relationships
- Screen layout organization
- Drawing order
- Event propagation flow

The root of the View hierarchy is always a View group, which may contain one or more child Views or View groups. The view group types most commonly used in Android layouts are LinearLayout, RelativeLayout and ConstraintLayout.

Each View in the hierarchy is assigned a unique identifier, which is used to identify the View and manipulate its properties programmatically. Views can also be assigned a tag, which is an optional object that can be used to store additional data about the view.

View hierarchies can be programmed or created using XML layout files. In both cases, the hierarchy is generated at runtime, and the result is used to render the user interface on screen.

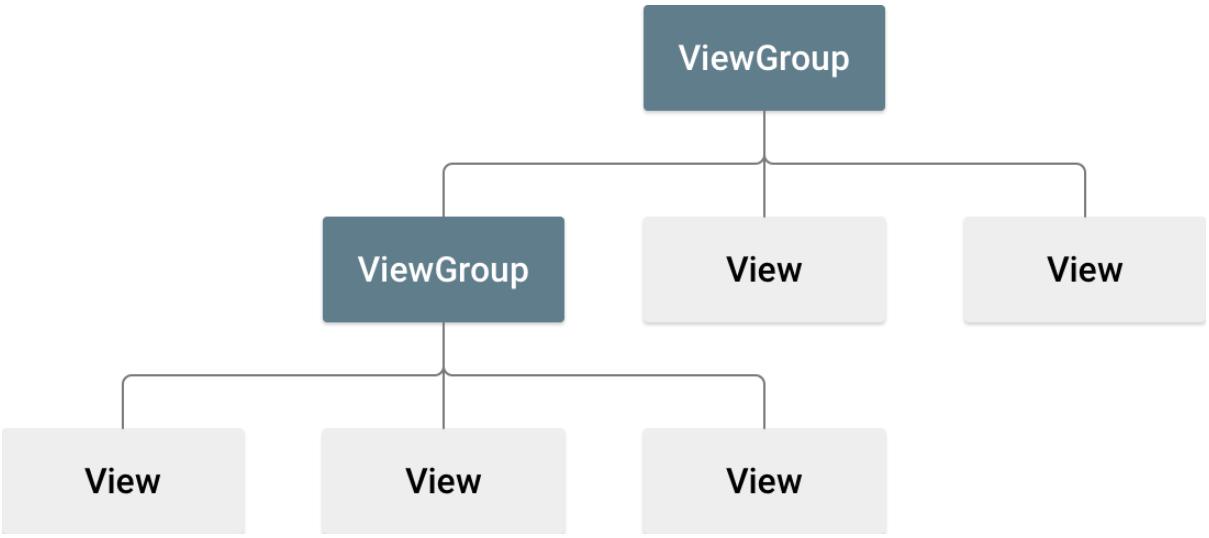

Figure 4 : Android UI View Hierarchy

Warning

It's important to note that the View hierarchy can have a significant impact on an application's performance and responsiveness. A deep or complex View hierarchy can result in slower layout and rendering times. For this reason, it is generally recommended to keep the View hierarchy as shallow and simple as possible.

4.2 Layouts

Layouts are container components responsible for organizing Views and widgets on the screen. While individual Views define *what* appears, layouts determine *where* and *how* those elements are positioned. [23], [24], [34]–[37]

In Android, layouts are implemented as subclasses of ViewGroup, meaning they can contain child Views or even other layouts. This nesting capability allows developers to construct complex, adaptive interfaces.

A good layout design is essential for:

- Visual clarity
- Responsiveness across devices
- Performance efficiency
- Maintainability

Android provides several layout types, each optimized for specific positioning strategies. Choosing the appropriate layout reduces complexity and improves rendering performance.

4.2.1 Linear Layout

LinearLayout is a type of layout in Android that allows developers to arrange child views vertically or horizontally, depending on their orientation. It's one of the most commonly used layouts in Android applications, because it's simple and easy to use. Layout orientation is defined using the android:orientation attribute in XML or the setOrientation() method in java code.[24], [35]

The LinearLayout class extends the ViewGroup class and can therefore contain any view as a child view. Child views are positioned one after the other in the order in which they are added to the layout. The default orientation is vertical, meaning that child views are arranged from top to bottom. If the orientation is horizontal, child views are arranged from left to right.

One of the advantages of using LinearLayout is that it's very easy to create responsive layouts that can adapt to different screen sizes and resolutions. For example, if the layout orientation is set to vertical, the height of each child view can be set to wrap_content, meaning that the view will be sized according to its content. This allows the layout to adapt automatically to different screen sizes and resolutions. [24], [35]

Another advantage of using LinearLayout is that it can be combined with other layouts to create more complex layouts. For example, one LinearLayout can be nested within another LinearLayout to create a two-dimensional layout. This can be useful for creating view grids or for positioning views in more complex ways.

Here's an example of a vertical LinearLayout in XML :

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!"/>

</LinearLayout>
```

Here's an example of horizontal LinearLayout:

```
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 1"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 2"/>

</LinearLayout>
```

and finally, you can create a LinearLayout programmatically in Java code as follows:

```
// Create a new LinearLayout
LinearLayout linearLayout = new LinearLayout(context);

// Set the orientation of the LinearLayout
linearLayout.setOrientation(LinearLayout.VERTICAL);

// Create a new TextView
TextView textView = new TextView(context);
textView.setText("Hello, World!");

// Add the TextView to the LinearLayout
linearLayout.addView(textView);

// Create a new Button
Button button = new Button(context);
button.setText("Click me!");

// Add the Button to the LinearLayout
linearLayout.addView(button);

// Set the LinearLayout as the content view of the Activity
setContentView(linearLayout);
```

Note

Please note that LinearLayout does not support the use of constraints to position views in relation to each other. For this, you'll need to use another layout, such as RelativeLayout or ConstraintLayout. However, LinearLayout remains a very useful layout for many scenarios and is often used in combination with other layouts to create more complex layouts.

4.2.2 Relative Layout

RelativeLayout is another type of layout in Android that allows developers to arrange user interface components in relation to each other. Unlike LinearLayout, RelativeLayout doesn't require child views to be arranged in a linear fashion. Instead, each child view is positioned relative to the others according to its relationship to the parent view or to other child views. RelativeLayout is therefore a flexible and powerful layout for creating complex user interfaces. [24], [36]

To use RelativeLayout, developers can define the position of each child view in relation to the parent view or other child views using attributes such as android:layout_above, android:layout_below, android:layout_toLeftOf and android:layout_toRightOf. These attributes enable developers to specify the relationship between Views in terms of relative position, for example above, below, to the left or right of another View. [24], [36]

One of the advantages of RelativeLayout is that it reduces the number of nested views required to achieve the desired layout. With LinearLayout, nested views are often required to achieve more complex layouts. However, with RelativeLayout, developers can position views in relation to each other without having to resort to nested views.

Here's an example of a simple RelativeLayout that positions two TextViews in relation to each other:

```
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_width="match_parent"
   android:layout_height="wrap_content">

   <TextView
       android:id="@+id/textview1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Hello" />

   <TextView
       android:id="@+id/textview2"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="World"
       android:layout_toRightOf="@id/textview1" />

</RelativeLayout>
```

the same example in java code

```
RelativeLayout relativeLayout = new RelativeLayout(context);
```

```
    2. RelativeLayout.LayoutParams layoutParams = new
RelativeLayout.LayoutParams(
    3.         RelativeLayout.LayoutParams.MATCH_PARENT,
RelativeLayout.LayoutParams.WRAP_CONTENT);
    4. relativeLayout.setLayoutParams(layoutParams);
    5.
    6. TextView textView1 = new TextView(context);
    7. textView1.setId(View.generateViewId());
    8. textView1.setText("Hello");
    9.
   10. RelativeLayout.LayoutParams textView1Params = new
RelativeLayout.LayoutParams(
   11.         RelativeLayout.LayoutParams.WRAP_CONTENT,
RelativeLayout.LayoutParams.WRAP_CONTENT);
   12. textView1.setLayoutParams(textView1Params);
   13.
   14. TextView textView2 = new TextView(context);
   15. textView2.setId(View.generateViewId());
   16. textView2.setText("World");
   17.
   18. RelativeLayout.LayoutParams textView2Params = new
RelativeLayout.LayoutParams(
   19.         RelativeLayout.LayoutParams.WRAP_CONTENT,
RelativeLayout.LayoutParams.WRAP_CONTENT);
   20. textView2Params.addRule(RelativeLayout.RIGHT_OF, textView1.getId());
   21. textView2.setLayoutParams(textView2Params);
   22.
   23. relativeLayout.addView(textView1);
   24. relativeLayout.addView(textView2);
   25.
   26. // Add relativeLayout to a parent view
   27. parentView.addView(relativeLayout);
```

When using Relative Layout in Android, you should be aware that it requires more manual coding than other layouts. This is because Relative Layout positions elements in relation to each other, often requiring you to specify the position of each element in relation to its neighbors. Furthermore, using RelativeLayout often requires the use of XML attributes and manual adjustments, rather than the simple drag-and-drop operation provided by other layout types such as ConstraintLayout. While this may seem more tedious, it allows more precise control over layout and view positioning, which can be particularly useful in complex UI designs.

4.2.3 Constraint Layout

ConstraintLayout is designed to adapt to different device sizes and orientations. It is a flexible layout manager that enables developers to create responsive user interfaces capable of adapting to different screen sizes and aspect ratios.[23], [24], [37]

One of ConstraintLayout's key features is its ability to create responsive designs using constraints. Constraints define the position and size of views in relation to other views or to the parent layout. This allows developers to create layouts that adapt to different screen sizes and orientations by adjusting constraints.

For example, you can create a layout with a TextView centered horizontally and vertically on the screen, and an ImageView below the TextView that fills the rest of the screen. To do this, you can define constraints between the views and the parent layout. When the screen size or orientation changes, the layout adjusts the constraints to maintain the same relative position of the views. [23], [24], [37]

In addition to constraints, Constraint Layout also supports guidelines, chains and barriers. Guidelines are invisible lines used to align views in a layout. Chains are used to group views and apply common constraints to the group. Barriers are used to create a virtual boundary between views.

Here's the xml code for the previous example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    <TextView
        android:id="@+id/textview1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ImageView
            android:id="@+id/imageview"
            android:layout_width="0dp"
            android:layout_height="0dp"
            android:src="@drawable/image"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/textview1" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

4.3 Fragments

Fragments are a fundamental part of the user interface of an Android application. They represent a modular part of a user interface and can be combined with each other to create a flexible and dynamic layout. Fragments were introduced in Android 3.0 (API level 11) to address the challenge of creating user interfaces that can adapt to different screen sizes and resolutions. By dividing the user interface into smaller pieces, developers can create more flexible and reusable layouts that can be easily adapted to different devices and form factors.[23], [24], [30], [38]

Essentially, a fragment is a self-contained user interface component with its own lifecycle that can be added to or removed from an activity's layout at runtime. Each fragment can have its own layout, behavior, and set of callbacks, and can communicate with other fragments or activities through a shared activity or through the use of interfaces. With fragments, developers have more control over the layout and behavior of their application's user interface, making it easier to create dynamic and responsive user interfaces.

Fragments in Android offer several advantages, including

- Reusability: Fragments can be reused across multiple activities, enabling modular and efficient development.
- Compatibility: Fragments are compatible with a wide range of devices and screen sizes, making it easier to develop applications that work across multiple devices.
- Simplified code: Fragments can simplify code by allowing developers to encapsulate code for specific user interface elements or functions in one place, making it easier to manage and maintain.
- Improved performance: Using fragments can improve the performance of your application by reducing the amount of memory and resources required to display complex user interface elements.

Overall, fragments are a powerful and flexible way to develop applications in Android, and using them can lead to more efficient, modular, and scalable code.

4.3.1 4.1. Life Cycle of a Fragment

Fragments in Android have a lifecycle that defines the different states and events they can go through from creation to destruction (Presented in Figure 5). The fragment lifecycle includes the following states: Active, Paused, Stopped, and Destroyed. [23], [24], [30], [38]

The lifecycle callback methods include:

- onAttach(): Called when the fragment is attached to its hosting activity.

- onCreate(): Called when the fragment is created.
- onCreateView(): Called when the fragment's user interface is created.
- onViewCreated(): Called after onCreateView() returns.
- onStart(): Called when the fragment is visible to the user.
- onResume(): Called when the fragment is visible and interactive.
- onPause(): Called when the fragment is still visible but not interactive.
- onStop(): Called when the fragment is no longer visible.
- onDestroyView(): Called when the fragment's view is being destroyed.
- onDestroy(): Called when the fragment is being destroyed.
- onDetach(): Called when the fragment is no longer associated with its hosting activity.

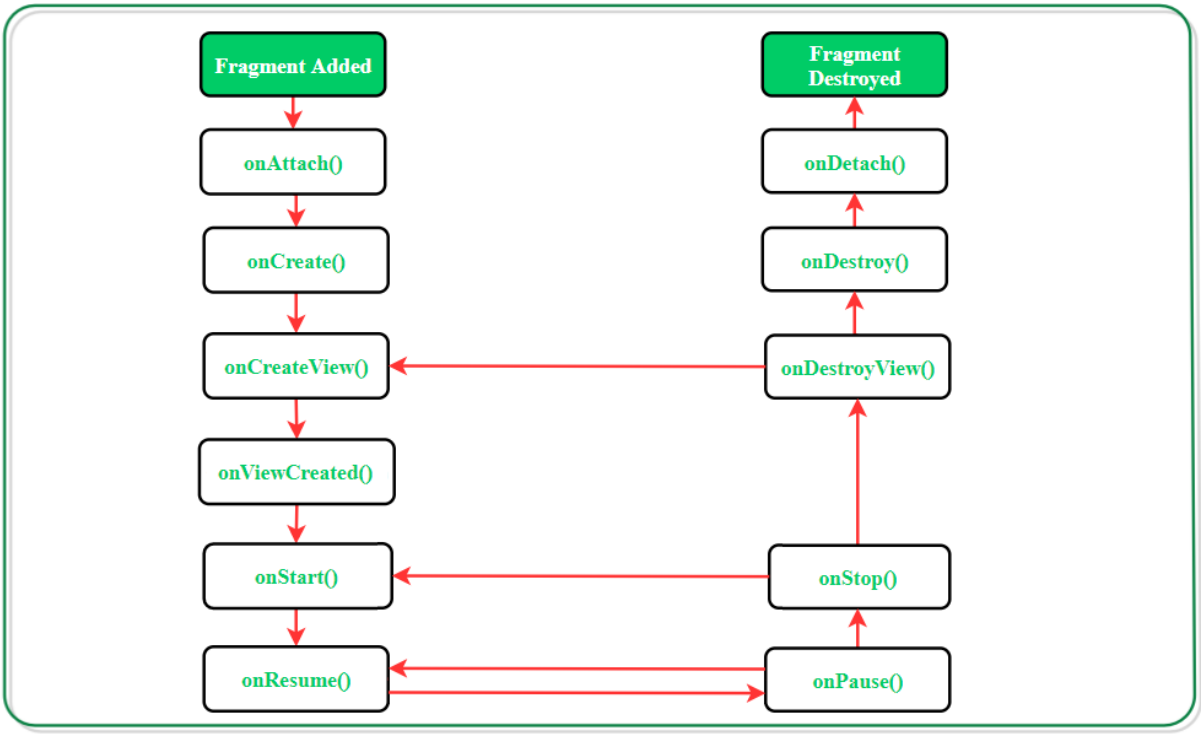

Figure 5: Fragment Lifecycle

4.3.2 How to Use the Fragment

In order to create a new fragment in your Android project, you need three things: the fragment's Java class, which represents the fragment's controller and implements the onCreateView() method to inflate the fragment's layout, the XML layout file for the fragment, and the activity to which you will add the fragment. The fragment's Java class extends the Fragment class, while the XML layout file defines the layout of the views in the fragment. Next, you need to add the fragment to the activity using a FragmentManager to start a

FragmentTransaction and calling the add() method to add the fragment to a layout container (FrameLayout) in the activity.

To create a new fragment in your Android application, you must follow these steps:

1. In Android Studio, open the project where you want to add the new fragment.
2. In the Project pane, right-click the folder where you want to add the new fragment.
3. In the context menu, select New > Fragment > Fragment (Blank).
4. In the dialog box that appears, enter a name for your new fragment, as well as any other desired options (for example, the location where you want to create the fragment).
5. Click Finish to create the new fragment and its associated layout.

This will automatically create a new fragment class and a layout file for the fragment, and add both to your project. You can then customize the layout and code of your fragment to suit your specific needs.

After creating a fragment in Android Studio, here are the steps to add it to an activity:

- In the Android Studio code editor, open the class of the activity where you want to add the fragment.
- Find the location in the activity layout where you want to add the fragment. Add a FrameLayout container that will serve as a container for your fragment.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

   <!— Other Views of the activity -->

    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@id/autre_vue"
        android:layout_alignParentBottom="true" />

</RelativeLayout>
```

- In the activity's onCreate() method, instantiate the fragment you created and add it to the container you just created in the previous step. You can do this using a FragmentManager and a FragmentTransaction.

Here is a sample code for adding the fragment:

```
// Instantiate the fragment
MyFragment fragment = new MyFragment();
```

```
    3.
    4. // Start the fragment transaction
    5. FragmentTransaction transaction =
getSupportFragmentManager().beginTransaction();
    6.
    7. // Add the fragment to the container
    8. transaction.add(R.id.container, fragment);
    9.
   10. // Commit the transaction
   11. transaction.commit();
```

- Compile and run your application to see your fragment displayed in the activity.

The code shown above is just a basic example, and there are many other features and options available for working with fragments in Android. Here are some additional options and features related to fragments in Android:

- Fragment transactions: In addition to adding, removing, and replacing fragments, you can also perform other operations on fragments using fragment transactions. These operations include attaching and detaching fragments and setting custom animations for fragment transitions.
- Fragment arguments: Fragment arguments allow you to pass data between fragments and between a fragment and its parent activity. You can pass arguments as a Bundle object using the setArguments() method and retrieve them in the fragment using the getArguments() method.
- Fragment communication: Fragments can communicate with their parent activity and with other fragments using interfaces. By defining an interface in a fragment and implementing it in the activity or in another fragment, you can send and receive data and events between fragments.
- Fragment layouts: Fragments can have their own layouts, which can be created using XML or programmatically. You can use a layout file to define the user interface elements and their properties in the fragment, and deploy the layout in the onCreateView() method.
- Restoring the fragment state: When a fragment is destroyed and recreated as a result of a configuration change, such as a screen rotation, you can save and restore its state using the onSaveInstanceState() and onViewStateRestored() methods. This allows you to preserve the fragment's data and UI state during configuration changes.
- Fragment transactions with the back stack: You can add fragments to the back stack and navigate between them using the “back” button. This provides users with a more intuitive navigation experience.

- Fragment customization: Fragments can be customized with different styles and themes to match the overall look and feel of the application. You can also customize the behavior of fragments by extending the Fragment base class and adding custom functionality.

4.4 5. Widgets

Widgets are ready-to-use interface components that allow users to view information and interact with an Android application. While layouts organize structure, widgets provide the interactive elements that make an interface functional.[23], [24], [30], [39]

Every widget ultimately derives from the View class, inheriting rendering and event-handling capabilities. Android supplies a rich collection of widgets that support text display, user input, navigation, feedback, and media presentation.

Widgets are typically declared in XML and manipulated in Java code, maintaining a clear separation between presentation and behavior.

4.4.1 Basic Widgets

Basic widgets are the most common user interface components in Android applications. They include views such as TextView, EditText, Button, ImageView, ProgressBar, CheckBox, and RadioButton, which allow users to interact with the application and display content.

a. TextView

The TextView widget is used to display text on the screen. It can be customized with properties such as font size, text color, and alignment. Here is an example of a TextView with bold, centered text:

```
<TextView
    android:id="@+id/textview_hello"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Bonjour le monde!"
    android:textStyle="bold"
    android:textAlignment="center"/>
```

b. EditText

The EditText widget allows the user to enter text. It can be used to enter text or numbers into the application. Here is an example of EditText with help text and a length limit of 10 characters:

```
<EditText
    android:id="@+id/edittext_input"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Entrez un texte ici"
    android:maxLength="10"/>
```

c. Button

The Button widget is used to trigger actions in the application when clicked. It can be customized with properties such as text, background color, and size. Here is an example of a Button with text in French:

```
<Button
    android:id="@+id/button_submit"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Envoyer"/>
```

d. ImageView

The ImageView widget is used to display images in the application. It can be customized with properties such as image source, size, and scale. Here is an example of ImageView with an image of a cat.

```
<ImageView
    android:id="@+id/imageview_cat"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/cat"/>
```

e. ProgressBar

The ProgressBar widget is used to indicate the progress status of a task in progress. It can be customized with properties such as the color of the progress bar and the style. Here is an example of a ProgressBar with a circle style

```
<ProgressBar
    android:id="@+id/progressbar_loading"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:indeterminate="true"
    style="?android:attr/progressBarStyleLarge"/>
```

f. CheckBox

The CheckBox widget is used to allow the user to select one or more options at a time. It can be customized with properties such as option text and selected state. Here is an example of a CheckBox with three options

```
<CheckBox
    android:id="@+id/checkbox_options"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Options"
    android:checked="true"/>

<CheckBox
    android:id="@+id/checkbox_option1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Option 1"/>

<CheckBox
    android:id="@+id/checkbox_option2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Option 2"/>
```

g. RadioButton

The RadioButton widget is used to allow the user to select one option at a time from a set of options. It can be customized with properties such as option text and selected state. Here is an example of a RadioButton with three options

```
<RadioGroup
    android:id="@+id/radiogroup_options"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <RadioButton
        android:id="@+id/radiobutton_option1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 1"/>

    <RadioButton
        android:id="@+id/radiobutton_option2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 2"/>

    <RadioButton
        android:id="@+id/radiobutton_option3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 3"/>
```

```
23. </RadioGroup>
```

4.4.2 Advanced Widgets

Advanced Android widgets offer more advanced and specialized features for creating more complex and interactive user interfaces. These widgets are more flexible and customizable than basic widgets, allowing developers to design more sophisticated and aesthetically pleasing applications.

a. RecyclerView

The RecyclerView widget is used to display a scrollable list of items, with great flexibility for customizing the display of individual items. It requires the use of an adapter to provide the data and define the view for each item. Here is an example of a RecyclerView with a list of names.

```
1. <androidx.recyclerview.widget.RecyclerView
2.     android:id="@+id/recyclerview_names"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"/>
5.
6. // Define the adapter in Java or Kotlin code
7. recyclerView.adapter = NamesAdapter(namesList)
```

b. DatePicker

The DatePicker widget allows users to select a date from a calendar. It can be customized with properties such as the initial date and selectable date limits. Here is an example of a DatePicker with an initial date.

```
1. <DatePicker
2.     android:id="@+id/datepicker"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:calendarViewShown="false"
6.     android:spinnersShown="true"
7.     android:minDate="2022-01-01"
8.     android:maxDate="2023-12-31"
9.     android:layout_margin="16dp"/>
```

c. WebView

The WebView widget displays a web content in a view embedded in the application. It allows you to display complete web pages, videos, images, and scripts using the same features as a standard web browser. Here is an example of a WebView that displays a web page.

```
<WebView
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

// Load the web page in Java or Kotlin code
webView.loadUrl("https://www.example.com")
```

d. SeekBar

The SeekBar widget allows the users to select a numeric value within a given range by dragging a slider. It can be customized with properties such as the value range and the initial value. Here is an example of a SeekBar with a value range from 0 to 100.

```
<SeekBar
    android:id="@+id/seekbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:max="100"
    android:progress="50"/>
```

e. TimePicker

The TimePicker widget enables users to select a time from a digital clock. It allows you to customize properties such as the initial time, time format, and range of values for the user. Here is an example of a TimePicker with an initial time of 12:00 and a 24-hour time format.

```
<TimePicker
    android:id="@+id/timepicker"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:timePickerMode="spinner"
    android:format24Hour="true"
    android:hour="12"
    android:minute="0"/>
```

f. RatingBar

The RatingBar widget lets users pick a rating from a rating bar. You can customize it with properties like the number of stars, the initial rating, and the range of values. Here's an example of a RatingBar with 5 stars and an initial rating of 3.

```
<RatingBar
    android:id="@+id/ratingbar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="5"
    android:rating="3"
    android:stepSize="1.0"/>
```

4.4.3 Event Handling

In Android or any other system, applications or programs need a way to interact with the user, which is where event handlers come in. Event handlers allow the system to listen to user actions and respond accordingly, enabling the user to interact with the application or program in a meaningful way.[23], [24], [30], [40], [41]

In Android specifically, event handlers are commonly used with widgets such as buttons, text views, and progress bars to provide functionality such as responding to clicks, long presses, state changes, and other user interactions. By using event handlers effectively, developers can create engaging and responsive user interfaces that provide a positive user experience.

In Android, event handlers are implemented as callback methods, which means they are called by the Android system in response to a specific event. To use an event handler in your Android app, you must define a method that matches the signature of the event handler you want to use, then register the method with the appropriate event source. [23], [24], [30], [40], [41]

For example, to handle a button click event, you can define a method with the following signature

```
1. public void onClick(View view) {
2.     // Handle button click event here
3. }
```

You can then register this method as an event handler for a button by calling the setOnClickListener() method on the button and passing the reference to the method.

```
1. Button button = findViewById(R.id.my_button);
2. button.setOnClickListener(this::onClick);
```

Here are some examples of the most commonly used event handlers in Android for widgets

- **OnClickListener**: used to handle the click event of a View (button, ImageView, etc.)

```
1. // OnClick listener for a button
2. Button myButton = findViewById(R.id.my_button);
3. myButton.setOnClickListener(new View.OnClickListener() {
4.     @Override
5.     public void onClick(View v) {
6.         // Handle button click event
7.     }
8. });
```

- **OnLongClickListener** : Handle the long-click event of a view

```
1. // OnLongClick listener for a button
2. Button myButton = findViewById(R.id.my_button);
```

```
3. myButton.setOnLongClickListener(new View.OnLongClickListener() {
4.     @Override
5.     public boolean onLongClick(View v) {
6.         // Handle long click event
7.         return true; // return true to consume the event
8.     }
9. });
```

- **TextChangedListener**: used to monitor text changes in an EditText

```
1. // TextWatcher for an EditText
2. EditText myEditText = findViewById(R.id.my_edit_text);
3. myEditText.addTextChangedListener(new TextWatcher() {
4.     @Override
5.     public void beforeTextChanged(CharSequence s, int start, int count,
int after) {
6.         // Called before text changed
7.     }
8.
9.     @Override
10.    public void onTextChanged(CharSequence s, int start, int before, int
count) {
11.        // Called during text changed
12.    }
13.
14.    @Override
15.    public void afterTextChanged(Editable s) {
16.        // Called after text changed
17.    }
18. });
```

- **OnItemSelectedListener**: handle the selection events of a Spinner

```
1. // OnItemSelected listener for a Spinner
2. Spinner mySpinner = findViewById(R.id.my_spinner);
3. mySpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {
4.     @Override
5.     public void onItemSelected(AdapterView<?> parent, View view, int
position, long id) {
6.         // Handle item selection event
7.     }
8.
9.     @Override
10.    public void onNothingSelected(AdapterView<?> parent) {
11.        // Handle no selection event
12.    }
13. });
```

- **OnCheckedChangeListener** : is a method that allows you to define a receiver for changes in the state of a check box or radio button in a View.

```
1. checkBox.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
2.     @Override
3.     public void onCheckedChanged(CompoundButton buttonView, boolean
isChecked) {
```

```
        // Code to handle checkbox state change
    }
});
```

The following table contains a list of common event handlers used in Android development, along with their descriptions and usage examples.

Table 6 : Android UI Event Listeners

Event handlers	*Description*	*Usage*
onTouch	Handles touch events for a view	view.setOnTouchListener(new View.OnTouchListener() {...});
onKey	Handles keyboard events for a view	view.setOnKeyListener(new View.OnKeyListener() {...});
onScrollChanged	Handles scroll events for a ScrollView or ListView view	view.setOnScrollChangeListener(new View.OnScrollChangeListener() {...});
onFocusChange	Handles focus change events for a view	view.setOnFocusChangeListener(new View.OnFocusChangeListener() {...});
onEditorAction	Handles IME action events for a view	view.setOnEditorActionListener(new TextView.OnEditorActionListener() {...});
onMenuClick	Handles click events for a menu item in a menu	menuItem.onMenuItemClick(new MenuItem.OnMenuItemClickListener() {...});
onItemClick	Handles click events for items in a ListView or GridView view	listView.setOnItemClickListener(new AdapterView.OnItemLongClickListener() {...});
onItemLongClick	Handles long click events for items in a ListView or GridView view	listView.setOnItemLongClickListener(new AdapterView.OnItemLongClickListener() {...});
onDateSet	Handles date selection events in a DatePickerDialog dialog box	datePickerDialog.setOnDateSetListener(new DatePickerDialog.OnDateSetListener() {...});
onTimeSet	Handles time selection events in a TimePickerDialog dialog box	timePickerDialog.setOnTimeSetListener(new TimePickerDialog.OnTimeSetListener() {...});
onProgressChanged	Handles progress change events for a SeekBar view	seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {...});
onRatingChanged	Handles rating change events for a RatingBar view	ratingBar.setOnRatingBarChangeListener(new RatingBar.OnRatingBarChangeListener() {...});
onSwipe	Handles swipe events for a view	view.setOnTouchListener(new OnSwipeTouchListener(context) {...});
onPinch	Handles pinch events for a view	view.setOnTouchListener(new OnPinchTouchListener(context) {...});

onDoubleTap	Handles double-tap events for a view	view.setOnTouchListener(new GestureDetector.OnDoubleTapListener() {...});
onGesturePerformed	Handles gesture events for a view	Overlay.addOnGesturePerformedListener(new GestureOverlayView.OnGesturePerformedListener() {...});

Chapter 5. Menus and Dialogs

Introduction

Menus and dialogs are fundamental components of the user interface in any mobile application. Menus provide users with organized, hierarchical access to the app's features, while dialogs are used to present important information or prompt the user for input. Efficient use of menus and dialogs can significantly enhance user experience and make an application more intuitive and user-friendly.

This chapter covers the fundamentals of menus and dialogs in Android, including how to create, customize, and integrate them effectively.

Menus in Android are generally classified into two types: options menus and context menus. Options menus offer a set of actions relevant to the current application context, while context menus are associated with a specific view or element and provide context-sensitive actions.[23], [24], [42]

Dialogs can be categorized into four types: alert dialogs, date pickers, time pickers, and custom dialogs. Alert dialogs are commonly used to display critical information or request user confirmation. Date and time pickers allow users to select a specific date or time from a predefined set of options. Custom dialogs are fully flexible and can be designed to create unique, context-specific user experiences.[23], [24], [43]

5.1 Menus

5.1.1 Options Menus

Options menus in Android provide a set of actions relevant to the current application context. They are typically accessed by pressing the Menu button on the device or tapping the overflow icon (three vertical dots) in the top-right corner of the screen. Options menus are hierarchical and can include sub-menus, allowing for better organization and more extensive options. [23], [24], [42]

Options menus are commonly used to give access to frequently used features and actions within an application. They can be defined programmatically in Java or through XML menu resource files. Is is important to note that options menus are not always visible, they appear only when the user requests them.

When displayed, the user can select a menu item to trigger an action or navigate to another screen. Options menus can also be customized with icons or other visual elements to enhance user experience.

In Android, the XML file defining an options menu is usually placed in the res/menu/ directory of the project. Each menu item is represented by an `<item>` tag with attributes such as id, title, and icon to specify the text and icon. Sub-menus can be nested within `<item>` tags to create hierarchical menus. [23], [24], [42]

Once defined, the XML menu can be linked to an activity or fragment by overriding two methods: `onCreateOptionsMenu()` and `onOptionsItemSelected()`. The first method inflates the XML file to create the menu, while the second handles user interaction with menu items.

To add an options menu to your application, you must first create the XML file that contains the menu options. The example below shows how to create an options menu with three options, one of which has two sub-options.

```
 1. <menu xmlns:android="http://schemas.android.com/apk/res/android">
 2.     <item
 3.         android:id="@+id/action_search"
 4.         android:icon="@drawable/ic_search"
 5.         android:title="Search"
 6.         android:showAsAction="ifRoom" />
 7.
 8.     <item
 9.         android:id="@+id/action_settings"
10.         android:title="Settings"
```

```
        android:showAsAction="never" />

    <item
        android:id="@+id/action_favorites"
        android:title="Favorites"
        android:icon="@drawable/ic_favorite"
        android:showAsAction="always">

        <menu>
            <item
                android:id="@+id/action_add_favorite"
                android:title="Add to favorites"
                android:icon="@drawable/ic_add_favorite"/>
            <item
                android:id="@+id/action_remove_favorite"
                android:title="Remove from favorites"
                android:icon="@drawable/ic_remove_favorite"/>
        </menu>
    </item>
</menu>
```

After creating the XML file, you can add it to the desired activity or fragment by overriding two methods, namely onCreateOptionsMenu and onOptionsItemSelected. The first method allows you to place the XML file in the activity or fragment, and the second manages actions related to selected menu items.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_item1:
            // Handle menu item 1 action
            return true;
        case R.id.menu_item2:
            // Handle menu item 2 action
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

5.1.2 Context menus

Context menus provide users with actions or additional information specific to a selected view or element. They appear when the user performs a long-press on a view, offering quick

access to frequently used actions or additional context specific options. Context menus are commonly used in list views, enabling users to act on individual list items.

the construction a contextual menu starts by defining the menu items either in an XML resource file located in the `res/menu` directory or directly in code. XML definition is generally preferred in instructional settings because it separates interface description from logic, thereby improving maintainability. For example, a menu resource file named `context_menu.xml` might contain the following items:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/copy"
        android:title="Copy" />
    <item android:id="@+id/cut"
        android:title="Cut" />
    <item android:id="@+id/paste"
        android:title="Paste" />
</menu>
```

Once the menu resource is prepared, the next step is to register the target view with the contextual menu system. This is accomplished by invoking the registerForContextMenu(View view) method within the onCreate function. Registration ensures that the system recognizes the view as capable of responding to a long-press gesture:

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

    TextView textView = findViewById(R.id.myTextView);
    registerForContextMenu(textView);  // Register the view for context
menu
}
```

When a registered view is long-pressed, the system automatically invokes the onCreateContextMenu callback. Inside this method, the menu resource that was previously defined is inflated into the contextual menu object, making its items available to the user. In cases where more than one view has been registered, the identity of the triggering view is examined (by checking its resource ID) and the appropriate menu resource is inflated for that specific element. This ensures that each view presents only the actions that are relevant to its context.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);

    if (v.getId() == R.id.myTextView) {
```

```
        getMenuInflater().inflate(R.menu.context_menu, menu);
    }
}
```

After the menu is displayed, the user may select one of the available actions. The system then invokes the onContextItemSelected method, where the logic for handling each menu item is provided. Each item is identified by its resource ID, and the corresponding action is executed. For example:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.copy:
            // handle copy action
            return true;
        case R.id.cut:
            // handle cut action
            return true;
        case R.id.paste:
            // handle paste action
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

That's it! Your context menu is now ready to use. When the user long presses on the view or saved item, the context menu appears and the corresponding action is performed when the user selects an item from the menu.

5.2 Dialogs

Dialogs in Android are user interface components that present a pop-up window with content to the user. They are typically employed to display messages, prompt for user input, or provide additional information relevant to the current application context. By temporarily interrupting the normal flow of interaction, dialogs allow applications to capture the user's attention and guide them toward a specific decision or action.[23], [24], [43]

Android supports several dialog types, each serving a distinct purpose. Alert dialogs are the most common and are used to display critical information, request user confirmation, or present a set of choices. Progress dialogs indicate the status of a time-consuming operation, such as downloading or processing data, thereby reassuring the user that the task is ongoing. Date pickers and time pickers provide structured interfaces for selecting a specific date or time from predefined options, ensuring accuracy and consistency in user input. Custom dialogs offer complete flexibility, allowing developers to design tailored user interfaces with widgets,

layouts, and behaviors that match the unique requirements of the application. Finally, bottom sheet dialogs slide up from the bottom of the screen to present additional options or contextual information. These can be configured as half-screen or full-screen components, triggered by button clicks or swipe gestures, and dismissed by swiping down or tapping outside the dialog.

5.2.1 Alert Dialog

Alert dialogs display messages to the user and request confirmation or input. They are commonly used to notify users of errors, confirm actions, or collect small pieces of information.

Steps to create an alert dialog:

1. Instantiate AlertDialog.Builder with the current context.
2. Set the dialog title, message, and buttons.
3. Call create() to build the dialog.
4. Call show() to display it.

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Title of the dialog");
builder.setMessage("Message to be displayed in the dialog");
builder.setPositiveButton("OK", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        // Do something when OK button is clicked
    }
});
builder.setNegativeButton("Cancel", new
DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        // Do something when Cancel button is clicked
    }
});
AlertDialog dialog = builder.create();
dialog.show();
```

5.2.2 Bottom Sheet Dialog

Bottom sheet dialogs slide up from the bottom of the screen to display content or actions. They are ideal for providing quick access to frequently used options or additional context-specific information.[44]

To create a Bottom Sheet Dialog, you can follow these steps:

- Define the layout XML for the dialog.
- Create a BottomSheetDialog instance in your activity.

- Inflate the layout and attach it using setContentView().
- Optionally configure properties such as peek height or behavior.
- Call show() to display the dialog.

Here is an example of code that shows how to create and display a bottom sheet dialog in an activity:

```
// Step 1: Create a BottomSheetDialog instance
BottomSheetDialog bottomSheetDialog = new BottomSheetDialog(this);

// Step 2: Set the layout for the Bottom Sheet Dialog
View bottomSheetView =
getLayoutInflater().inflate(R.layout.bottom_sheet_dialog, null);
bottomSheetDialog.setContentView(bottomSheetView);

// Step 3: Set optional properties for the Bottom Sheet Dialog
bottomSheetDialog.setBehavior(BottomSheetBehavior.STATE_EXPANDED);
bottomSheetDialog.setPeekHeight(300);

// Step 4: Display the Bottom Sheet Dialog
bottomSheetDialog.show();
```

5.2.3 Custom Dialogs

Custom dialogs provide fully flexible interfaces, allowing developers to include any Android view including text fields, buttons, drop-downs, etc. They are ideal for collecting user input or creating specialized interactions.[43]

Steps to create a custom dialog:

1. Define an XML layout containing the desired views (see below).
2. Instantiate AlertDialog.Builder with the current context.
3. Set dialog title and message (optional).
4. Add positive and negative buttons.
5. Inflate the custom layout and attach it using setView().
6. Create and show the dialog.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/dialog_title"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:text="Custom Dialog Title"
        android:padding="16dp"/>

    <EditText
        android:id="@+id/dialog_input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter input here..."
        android:padding="16dp"/>

</LinearLayout>
```

AlertDialog and BottomSheetDialog can both be customized to add more elements to the View and provide additional functionality.

In AlertDialog, you can customize the layout by adding a custom View using the setView() method of the AlertDialog.Builder class. Once the custom view is added, you can add additional widgets such as buttons, text fields, etc. You can also add custom logic to handle events generated by these widgets.

In BottomSheetDialog, you can customize the layout by adding a custom view using the setContentView() method. Once you have added the custom view, you can add additional widgets and handle their events if necessary. In addition, you can customize the behavior of BottomSheetDialog by providing custom callbacks such as onSlide() or onStateChanged().

Chapter 6. Data Management

Introduction

Any type of mobile or desktop application should behave as a persistent, state-aware system rather than a temporary runtime session. Users expect their data, preferences, progress, and media to remain available after each launch, device restart, and connection interruption. In the context of Android, persistence is not just a storage issue, it is a fundamental architectural layer that directly influences responsiveness, reliability, and user trust.

This section establishes the conceptual framework necessary to understand how Android organizes persistent data. Instead of viewing storage APIs as isolated tools, it is important to interpret them as coordinated components of a larger data ecosystem, shaped by system constraints, user experience goals, and evolving security models.

6.1 Data Persistence: Purpose and Principles

Data persistence is directly related to the quality of the user experience. A well-designed persistence strategy enables an application to maintain continuity despite life cycle interruptions, process shutdowns, or connectivity losses.[45], [46]

From a user experience perspective, persistence directly supports:

- State restoration: resuming workflows without data loss.
- Offline capability: functioning independently of network availability.
- Performance optimization: caching frequently accessed data.
- Reliability: long-term backup of user information.

This continuity is particularly critical in Android, where the operating system aggressively manages memory and background processes. Applications must assume that data in memory is temporary and explicitly design persistence boundaries. [45], [46]

Table 7: Data Classification

Data Category	***Description***
Temporary/Cache data	Temporary data used to accelerate access and optimize performance.
Preferences/Configuration	Lightweight application state and user settings.
Databases	Complex domain entities that require querying and strict relationships.
Shared Media/Documents	User-visible files, such as photos or downloads.
Distributed/Cloud data	Information that requires synchronization across multiple devices.

This classification highlights an important principle: *persistence is not a single problem; it is a spectrum of storage responsibilities***.** Each category implies different requirements regarding consistency, performance, privacy, and synchronization. Understanding this layered perspective prepares developers to map real-world application needs to appropriate storage strategies, rather than defaulting to a single, one-size-fits-all mechanism.

6.2 Databases in Android

Modern Android applications rarely operate without persistent data. Whether storing user profiles, caching remote content, managing application settings, or maintaining transactional records, applications require structured and reliable local storage mechanisms.

While simple key–value storage (e.g., `SharedPreferences` [47]) is suitable for lightweight configuration data, it becomes insufficient when the application must manage structured entities, relationships, constraints, or large datasets. In such cases, a database system is required.

Android offers native support for relational databases via SQLite [23], [24], [30], [48] and, more recently, a higher-level abstraction layer via the Room Persistence Library [24], [49]. It is essential to understand both approaches, as each has advantages and disadvantages.

This section introduces relational database management in Android, first through raw SQLite, then through Room, and finally through a structured comparison that enables informed technological choice.

6.2.1 SQLite in Android

SQLite is a lightweight, open-source relational database management system that is integrated into many mobile and web applications. It is designed for small to medium-sized databases and is popular due to its simplicity, ease of use, and low memory consumption. SQLite is used by many popular mobile applications such as Android, iOS, and Blackberry. [23], [24], [30], [48]

In this section, we will look at how to use SQLite in Android applications. We will cover the basics of SQLite, creating and managing databases, tables, and indexes, as well as performing various operations on data, such as inserting, updating, deleting, and querying data. We will also cover best practices for using SQLite in Android applications.

a. Introduction to SQLite

In Android, SQLite is used as the default database management system. It is used to store and manage data in local databases on the device. With SQLite, developers can create and manage databases, tables, indexes, and queries to perform various operations on the data. Some advantages of using SQLite in Android applications include:

- Easy integration: SQLite is built into the Android operating system, so it is easy to integrate into Android applications.
- No configuration required: There is no need to set up a separate server or database engine to use SQLite in Android applications. The database can be created and managed entirely within the application.
- High performance: SQLite is a fast database management system that can handle large amounts of data and execute queries quickly.
- Portable: The SQLite database file can be easily transferred between devices or backed up for safekeeping.
- Low memory usage: SQLite uses a small amount of memory and disk space compared to other database management systems, making it ideal for use in mobile applications.
- ACID compliance: SQLite is ACID compliant, which ensures that transactions are executed correctly and reliably, even in the event of a power outage, crash, or other system failures.
- SQL support: SQLite supports SQL, which is a widely used language for managing relational databases. This makes it easy to write complex queries and manipulate data in the database.

The table below compares SQLite with other database tools:

Table 8: Comparison between database tools

Database tool	*Advantage*	*Disadvantage*
SQLite	Lightweight, easy to integrate, open-source, suitable for small and medium-sized applications.	Not suitable for large-scale applications, does not support complex stored procedures.
MySQL	Powerful, scalable, widely used, suitable for complex data queries.	More complex to set up and configure, requires more resources than SQLite, not suitable for small applications.
PostgreSQL	Open-source, supports complex data types and queries.	More complex to set up and configure than SQLite, not suitable for small applications.
Oracle	Powerful, scalable, suitable for large-scale applications.	Expensive license, requires significant resources to run, not suitable for small applications.
MongoDB	Ideal for unstructured data, scalable, fast.	Less suited for deeply relational data. while it now supports transactions, it is generally less optimal for heavy transactional workloads than traditional SQL.

SQLite supports several different data types, including numeric data types (INTEGER, REAL), text data types (TEXT, CHAR, VARCHAR), binary data types (BLOB), and date and time data types (DATE, TIME, DATETIME). Here is a table of the 10 most commonly used data types in SQLite:

Table 9: SQLite Data Types

Data type	*Description*
INTEGER	64-bit signed integer
TEXT	Unicode text string
REAL	64-bit floating point number
BLOB	Raw binary data
NULL	Null value
NUMERIC	Generic numeric value
DATE	Date in 'YYYY-MM-DD' format
TIME	Time in 'HH:MM:SS' format
DATETIME	Date and time in 'YYYY-MM-DD HH:MM:SS' format
BOOLEAN	Boolean value (0 or 1)

b. SQLiteOpenHelper

SQLiteOpenHelper is a class in the Android framework that makes it easy to create and manage a SQLite database. It serves as a helper class for managing database creation and

version management. The class manages basic database operations such as opening a connection to the database, creating the database if it does not exist, and updating the database schema if necessary.[24], [48]

SQLiteOpenHelper consists of two main methods: onCreate() and onUpgrade(). onCreate() is called when the database is first created and is used to create the initial database schema. onUpgrade() is called when the database version is updated and is used to modify the existing schema to reflect changes made to the application.

To use SQLiteOpenHelper, you must create a subclass and implement these two methods. You must also define the database schema, including the names of the tables and their respective columns. The class takes care of creating and updating the database schema as needed, so you don't have to worry about manually managing the database.

```
1. public class MyDatabaseHelper extends SQLiteOpenHelper {
2.     private static final String DATABASE_NAME = "mydatabase.db";
3.     private static final int DATABASE_VERSION = 1;
4.
5.     public MyDatabaseHelper(Context context) {
6.         super(context, DATABASE_NAME, null, DATABASE_VERSION);
7.     }
8.
9.     @Override
10.    public void onCreate(SQLiteDatabase db) {
11.        // Create your database tables here
12.        db.execSQL("CREATE TABLE mytable (id INTEGER PRIMARY KEY, name
TEXT)");
13.    }
14.
15.    @Override
16.    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
17.        // Handle database upgrades here
18.        db.execSQL("DROP TABLE IF EXISTS mytable");
19.        onCreate(db);
20.    }
21. }
```

In addition to these two methods, the SQLiteOpenHelper class also provides the getReadableDatabase() and getWritableDatabase() methods, which return a SQLiteDatabase object that can be used to perform CRUD (Create, Read, Update, Delete) operations on the database. These methods ensure that only one SQLiteDatabase object is created per database and manage the locking and unlocking of the database. The main methods of the SQLiteDatabase object are:

- execSQL(String sql): Executes a single SQL statement that is not a SELECT or any other SQL statement that returns data.

```
db.execSQL("CREATE TABLE my_table (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)");
```

- rawQuery(String sql, String[] selectionArgs): Executes an SQL SELECT statement and returns a cursor object containing the results.

```
// Assuming a SQLiteDatabase object named "db" has been initialized

String query = "SELECT * FROM users WHERE age > ?";
String[] selectionArgs = new String[] { "18" };

Cursor cursor = db.rawQuery(query, selectionArgs);

if (cursor.moveToFirst()) {
    do {
        String name = cursor.getString(cursor.getColumnIndex("name"));
        int age = cursor.getInt(cursor.getColumnIndex("age"));
        // Do something with name and age
    } while (cursor.moveToNext());
}

cursor.close();
```

- insert(String table, String nullColumnHack, ContentValues values): Inserts a row into the specified table.

```
ContentValues values = new ContentValues();
values.put("name", "John");
values.put("age", 25);
long newRowId = db.insert("my_table", null, values);
```

- update(String table, ContentValues values, String whereClause, String[] whereArgs): Updates one or more rows in the specified table.

```
ContentValues values = new ContentValues();
values.put("age", 26);
String selection = "name = ?";
String[] selectionArgs = {"John"};
int count = db.update("my_table", values, selection, selectionArgs);
```

- delete(String table, String whereClause, String[] whereArgs): Deletes one or more rows from the specified table.

```
String selection = "age < ?";
String[] selectionArgs = {"20"};
int count = db.delete("my_table", selection, selectionArgs);
```

c. SQLite Usage - Complete Example

To use SQlite in android we start by adding the following dependencies to the project's build.gradle file:

```
dependencies {
    ....
    implementation 'androidx.sqlite:sqlite:2.1.0'
    ....
}
```

After that we create a new Java class called “DatabaseHelper” and extend the SQLiteOpenHelper class:

```
    // Called when the database needs to be upgraded
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        // Drop older table if existed
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);

        // Create tables again
        onCreate(db);
    }

    // Insert a new row into the table
    public long insertData(Student student) {
        SQLiteDatabase db = this.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(COLUMN_NAME, student.getName());
        values.put(COLUMN_AGE, student.getAge());
        values.put(COLUMN_ADDRESS, student.getAddress());
        values.put(COLUMN_LEVEL, student.getLevel());

        long id = db.insert(TABLE_NAME, null, values);

        db.close();

        return id;
    }
   // Get all students from the table
    public List<Student> getAllStudents() {
        List<Student> students = new ArrayList<>();

        SQLiteDatabase db = this.getReadableDatabase();

        String selectQuery = "SELECT * FROM " + TABLE_NAME;

        Cursor cursor = db.rawQuery(selectQuery, null);

        if (cursor.moveToFirst()) {
            do {
                Student student = new Student();

student.setId(cursor.getInt(cursor.getColumnIndex(COLUMN_ID)));

student.setName(cursor.getString(cursor.getColumnIndex(COLUMN_NAME)));
```

```
   42.
student.setAge(cursor.getInt(cursor.getColumnIndex(COLUMN_AGE)));
   43.
student.setAddress(cursor.getString(cursor.getColumnIndex(COLUMN_ADDRESS)));
   44.
student.setLevel(cursor.getInt(cursor.getColumnIndex(COLUMN_LEVEL)));
   45.
   46.                 students.add(student);
   47.             } while (cursor.moveToNext());
   48.         }
   49.
   50.         cursor.close();
   51.         db.close();
   52.
   53.         return students;
   54.     }
   55.      // Update a student in the table
   56.     public int updateStudent(Student student) {
   57.         SQLiteDatabase db = this.getWritableDatabase();
   58.
   59.         ContentValues values = new ContentValues();
   60.         values.put(COLUMN_NAME, student.getName());
   61.         values.put(COLUMN_AGE, student.getAge());
   62.         values.put(COLUMN_ADDRESS, student.getAddress());
   63.         values.put(COLUMN_LEVEL, student.getLevel());
   64.
   65.         return db.update(TABLE_NAME, values, COLUMN_ID + " = ?",
   66.                 new String[]{String.valueOf(student.getId())});
   67.     }
   68.     // Delete a student from the table
   69.     public void deleteStudent(Student student) {
   70.         SQLiteDatabase db = this.getWritableDatabase();
   71.         db.delete(TABLE_NAME, COLUMN_ID + " = ?",
   72.                 new String[]{String.valueOf(student.getId())});
   73.         db.close();
   74.     }
   75. }
```

The code above represents a Java class that extends SQLiteOpenHelper, a helper class for managing database creation and versioning. The class has methods for creating, inserting, updating, and deleting data from a table called “Students” with five columns: “id,” “name,” “age,” “address,” and “level.”

- The class constructor takes a Context object, which is used to create and access the database.
- The onCreate() method is called when the database is first created. It creates the “Students” table with the columns specified in the CREATE_TABLE query.
- The onUpgrade() method is called when the database needs to be updated, usually when the database schema is changed.
- The insertData() method inserts a new row into the “Students” table with the data from the given Student object. It returns the ID of the inserted row.

- The updateData() method updates a row in the “Students” table with the data from the given Student object. It returns the number of rows affected.
- The deleteData() method deletes a row from the “Students” table with the given ID. It returns the number of rows affected.
- The getAllStudents() method retrieves all rows from the “Students” table and returns them as a list of Student objects.
- The updateStudent() method updates a row in the “Students” array with the data from the given Student object. It returns the number of rows affected.
- The deleteStudent() method deletes a row from the “Students” array with the data from the given Student object.

6.2.2 Room Persistence Library

Despite SQLite's robust capabilities as a local database, interacting with it directly poses significant practical challenges in terms of code clarity and maintainability. The raw API is highly vulnerable to human error. For example, to retrieve a simple list of users, a developer must manually open connections, navigate through Cursor objects, map column data to domain models, and strictly manage resource closures. Multiplying this mechanical process across an entire project results in a fragile code base, weighed down by unnecessary repetition.[24], [49]

Room directly addresses these shortcomings. As a core persistence library within Google's Android Jetpack, Room provides an elaborate abstraction layer on top of SQLite. It abstracts away complex configuration while retaining all the functionality of the database. By providing compile-time validation of SQL queries, ensuring thread safety, and offering native integration with components such as LiveData and ViewModel, Room ensures that database code remains clean, secure, and consistent across all versions of Android.

This chapter presents Room in detail. We will examine its three main components (the entity, the DAO, and the database), then review a complete practical example, before finally exploring how Room handles schema changes using migrations.

d. Room's Core Components

Room is built around three closely related components, each with a specific responsibility. It is essential to understand the role of each component and their mutual relationships before writing any code.[49]

Entity

An entity is a simple Java class that represents a table in the database. Each instance of the class corresponds to a row in that table, and each field in the class corresponds to a column. Room uses the @Entity annotation to identify the class and determine how to map it to a SQLite table. By default, Room uses the class name as the table name and the field names as column names, but both can be customized using annotation parameters.

Each entity must define a primary key (which uniquely identifies each row). This is done using the @PrimaryKey annotation. In most cases, developers set autoGenerate = true so that SQLite automatically assigns an incremental integer to each new record, eliminating the need to manually manage identifiers. Additional annotations such as @ColumnInfo allow for finer control over column naming and behavior, while @Ignore tells Room to ignore a field completely and not persist it in the database. The table below describe more annotations for the entity class.[24], [49]

Table 10: Entity class annotations

Annotation	*Level*	*Primary Purpose*	*Key Parameters*
@Entity	Class	Marks the class as a database table.	tableName, primaryKeys, foreignKeys, indices
@PrimaryKey	Field	Sets the unique identifier for each row.	autoGenerate = true
@ColumnInfo	Field	Customizes the column configuration.	name, defaultValue, index, collate
@Ignore	Field	Tells Room not to persist this field to the DB.	N/A
@Embedded	Field	Flattens a nested object's fields into the table.	prefix
@ForeignKey	Class	Defines a relationship with another Entity.	entity, parentColumns, childColumns, onDelete
@Index	Class	Speeds up queries or enforces uniqueness	value, unique = true

The following example illustrates a data model consisting of a user entity linked to a Company via a foreign key relationship, while integrating an Address as an embedded object to show how Room flattens complex Java structures into a single optimized SQLite table.

```
1. package com.example.myapp.data.entity;
2.
3. import androidx.room.ColumnInfo;
4. import androidx.room.Embedded;
5. import androidx.room.Entity;
```

```
import androidx.room.ForeignKey;
import androidx.room.Ignore;
import androidx.room.Index;
import androidx.room.PrimaryKey;

/**
 * 1. Explicit Table Name: Decouples Java class name from Database table name.
 * 2. Indices: Unique constraint on email for data integrity and faster lookups.
 * 3. Foreign Key: Ensures a User cannot point to a non-existent Company.
 */
@Entity(
    tableName = "users",
    indices = {@Index(value = {"email"}, unique = true)},
    foreignKeys = @ForeignKey(
        entity = Company.class,
        parentColumns = "company_id",
        childColumns = "work_id",
        onDelete = ForeignKey.CASCADE
    )
)
public class User {

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "user_id")
    private int id;

    // Collate NOCASE: Good practice for names/emails to ignore capitalization in searches
    @ColumnInfo(name = "full_name", collate = ColumnInfo.NOCASE)
    private String name;

    private String email;

    @Embedded(prefix = "addr_")
    private Address address;

    @ColumnInfo(name = "work_id")
    private int companyId;

    // Field used only for UI logic, not persisted in SQLite
    @Ignore
    private boolean isSelected;

    // --- CONSTRUCTORS ---

    // Primary constructor used by Room
    public User(int id, String name, String email, Address address, int companyId) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.address = address;
        this.companyId = companyId;
    }

```

```
    // Secondary constructor for manual creation (where ID is 0/auto-generated)
    @Ignore
    public User(String name, String email, Address address, int companyId) {
        this.name = name;
        this.email = email;
        this.address = address;
        this.companyId = companyId;
    }

    // --- GETTERS & SETTERS (Standard Boilerplate) ---

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public Address getAddress() { return address; }
    public void setAddress(Address address) { this.address = address; }

    public int getCompanyId() { return companyId; }
    public void setCompanyId(int companyId) { this.companyId = companyId; }

    public boolean isSelected() { return isSelected; }
    public void setSelected(boolean selected) { isSelected = selected; }
}
```

```
public class Address {
    private String street;
    private String city;

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    // Room needs getters/setters for embedded fields too
    public String getStreet() { return street; }
    public String getCity() { return city; }
}
```

```
@Entity(tableName = "companies")
public class Company {
    @PrimaryKey
    @ColumnInfo(name = "company_id")
    private int id;

    private String companyName;
```

```
 8.
 9.      public Company(int id, String companyName) {
10.          this.id = id;
11.          this.companyName = companyName;
12.      }
13.
14.      public int getId() { return id; }
15.      public String getCompanyName() { return companyName; }
16. }
```

Notes :

- **Default Naming:** If you don't provide a tableName or @ColumnInfo(name = "..."), Room defaults to the **class name** and **field names**. In professional development, it is a "best practice" to define these explicitly to keep the database schema independent of your code's refactoring (like renaming a variable).
- **Composite Keys:** If your entity doesn't have one single ID but relies on two fields to be unique (like a firstName and lastName combination), you must define them inside the @Entity annotation:

@Entity(primaryKeys = ["firstName", "lastName"])

- **The @Embedded Advantage:** This is particularly useful for keeping your Kotlin/Java code clean. For example, if you have a Location class with latitude and longitude, using @Embedded puts those two coordinates directly into your User table without needing a second table or a complex join.

When you run a query like SELECT * FROM users, Room gets a row of raw data (IDs, Strings, Integers). It then looks at your User class and says: *"Okay, I have this data, but how do I turn it into a Java Object?"* To do this, it **must** call a constructor and Room follows a very specific set of rules to pick the right "tool" for the job:

1. **The Matching Rule:** Room looks for a constructor whose parameters match the fields it is loading from the database (by name and type).
2. **The "Only One" Rule:** If you have only one constructor, Room will use it. If that constructor is missing a field that isn't @Ignore-ed, Room will throw a compile-time error.
3. **The @Ignore Rule:** If you have multiple constructors, Room will get confused. You must use the @Ignore annotation on all constructors except the one you want Room to use.

Data Access Object (DAO)

The data access object, commonly referred to as DAO, is the component responsible for defining how the application interacts with the database. It is a Java interface annotated with @Dao. In this interface, developers declare methods that correspond to database operations: inserting, deleting, or retrieving records using a query. Room reads these method declarations at compile time and automatically generates the complete SQLite implementation.

Room provides three convenient annotations for the most common operations: @Insert, @Update, and @Delete. These annotations do not require any SQL: Room deduces the correct statement from the entity type passed to the method. For select operations and more complex queries, the @Query annotation accepts a raw SQL string. Room validates this SQL at compile time, which means that a typo or incorrect column name will produce a compile error rather than a crash on the user's device at runtime. This compile-time safety is one of the most valuable features Room offers over raw SQLite.[24], [49]

Since database operations can be slow and should never block the main thread (user interface), DAO methods that write to the database must be executed on a background thread. Room strictly enforces this rule: calling an insert or delete method on the main thread will throw an exception, unless the database is explicitly configured to allow it (which is not recommended). For read operations, it is preferable to return a LiveData object, as this avoids the developer having to manually manage threads for queries.

```
@Dao
public interface UserDao {
    // Query with parameters
    @Query("SELECT * FROM users ORDER BY first_name ASC")
    LiveData<List<User>> getAllUsers();
    //insert new user
    @Insert
    void insert(User user);
    //delete a user
    @Delete
    void delete(User user);
}
```

Database

The Database class is the parent component that connects all other elements. It is an abstract class that extends RoomDatabase and is annotated with @Database. This annotation takes two mandatory parameters: a list of Entity classes that define the tables in this database, and a version number that Room uses to track changes to the schema over time. The class must declare an abstract method for each DAO, which Room implements automatically.[24], [49]

Since creating a database instance is an expensive operation, it is common to implement the Database class as a singleton to ensure that only one instance exists during the lifetime of the application. The getInstance() method below uses the synchronized keyword to ensure thread safety, preventing two threads from simultaneously creating separate database instances.

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {

    public abstract UserDao userDao();

    // Volatile instance for thread visibility
    private static volatile AppDatabase INSTANCE;

    // Singleton pattern with double-checked locking
    public static synchronized AppDatabase getInstance(Context context)
{
        if (instance == null) {
            instance = Room.databaseBuilder(
                    context.getApplicationContext(),
                    AppDatabase.class,
                    "users_database"
            ).build();
        }
        return instance;
    }
}
```

Execution

Since Room prohibits database access on the Main Thread to prevent the UI from freezing, all operations must be offloaded to a Background Thread. Below is the implementation flow for inserting a new record using the components previously defined.

```
// Create the data object
Address userAddress = new Address("123 Android Lane", "Algiers");
User newUser = new User("John Doe", "john.doe@email.com", userAddress,
101);

// Get the Database Instance
AppDatabase db = AppDatabase.getInstance(this);

// Execute on a Background Thread
new Thread(() -> {
    try {
        // Perform the insertion
        db.userDao().insertUser(newUser);


        runOnUiThread(() -> {
            Toast.makeText(this, "User saved successfully!",
Toast.LENGTH_SHORT).show();
        });

```

```
    } catch (Exception e) {
        e.printStackTrace();
    }
}).start();
```

6.3 File Storage

Beyond databases, Android apps often need to read and write raw files: documents, images, audio files, cached responses, or configuration data that don't naturally fit into a relational schema. Android provides two distinct file storage areas for this purpose: internal storage, which is private to the app, and external storage, accessible to other apps and the user via a file manager. Understanding the difference between these two storage areas and choosing the right one for a given use case is essential for developing apps that are both functional and secure. [24], [46]

6.3.1 Internal Storage

Internal storage is a private area of the device's built-in memory, reserved exclusively for the app that created the files. No other app or user can directly access this area, making it the ideal choice for sensitive or app-specific data, such as configuration files, cached responses, or user preferences that should not be shared. Storage is always available, requires no special permissions, and its contents are automatically deleted when the app is uninstalled, ensuring that no orphaned data remains on the device[24], [46].

Android provides access to the internal storage directory via the Context class. The getFilesDir() method returns a File object pointing to the root of the app's private directory. From this root, files and subdirectories can be created, read, and deleted using standard Java I/O operations. To write a new file, the openFileOutput() method returns a FileOutputStream associated with a named file. To read an existing file, openFileInput() returns the corresponding FileInputStream. Subdirectories can be created and deleted using the mkdir() and delete() methods of the File class.

The following example shows how to create, read, and delete a text file in internal memory, as well as how to create and delete a subdirectory.

```
public class MainActivity extends AppCompatActivity {

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

```
    7.
    8.          // get the app's internal storage directory
    9.          File internalStorageDir = getFilesDir();
   10.
   11.          // create a new file in the internal storage directory
   12.          String fileName = "myFile.txt";
   13.          String fileContents = "Hello, World!";
   14.          try {
   15.              FileOutputStream fos = openFileOutput(fileName,
Context.MODE_PRIVATE);
   16.              fos.write(fileContents.getBytes());
   17.              fos.close();
   18.              Log.d("MainActivity", "File created: " + fileName);
   19.          } catch (IOException e) {
   20.              e.printStackTrace();
   21.          }
   22.
   23.          // read the contents of the file
   24.          try {
   25.              FileInputStream fis = openFileInput(fileName);
   26.              BufferedReader reader = new BufferedReader(new
InputStreamReader(fis));
   27.              StringBuilder sb = new StringBuilder();
   28.              String line;
   29.              while ((line = reader.readLine()) != null) {
   30.                  sb.append(line);
   31.              }
   32.              String fileContentsRead = sb.toString();
   33.              fis.close();
   34.              Log.d("MainActivity", "File contents: " + fileContentsRead);
   35.          } catch (IOException e) {
   36.              e.printStackTrace();
   37.          }
   38.
   39.          // delete the file
   40.          boolean deleted = deleteFile(fileName);
   41.          if (deleted) {
   42.              Log.d("MainActivity", "File deleted: " + fileName);
   43.          } else {
   44.              Log.d("MainActivity", "Failed to delete file: " + fileName);
   45.          }
   46.
   47.          // create a new directory in the internal storage directory
   48.          String dirName = "myDirectory";
   49.          File dir = new File(internalStorageDir, dirName);
   50.          boolean created = dir.mkdir();
   51.          if (created) {
   52.              Log.d("MainActivity", "Directory created: " + dirName);
   53.          } else {
   54.              Log.d("MainActivity", "Failed to create directory: " +
dirName);
   55.          }
   56.
   57.          // delete the directory
   58.          boolean deletedDir = dir.delete();
   59.          if (deletedDir) {
   60.              Log.d("MainActivity", "Directory deleted: " + dirName);
   61.          } else {
```

```
            Log.d("MainActivity", "Failed to delete directory: " +
dirName);
        }
    }
}
```

Internal storage in Android is suitable for storing app-specific configuration files or cached data when security, performance, and availability are important considerations. Here are a few examples:

- User preferences: When an app needs to store user preferences, such as font size, color scheme, or notification settings, internal storage can be a good option. Since this data is app-specific, it does not need to be shared with other apps or devices.
- Cached data: Apps can use internal storage to cache frequently accessed data, such as images or database queries, to improve performance. Cached data can be cleared when the app is closed or after a certain amount of time to free up space.
- Configuration files: Internal storage can be used to store configuration files necessary for the app to function properly, such as API keys or database connection information.
- App-specific data: Apps can use internal storage to store app-specific data, such as game scores or user profiles, which should not be accessible to other apps or users

6.3.2 External Storage

External storage refers to shared storage space that can be provided by a physical SD card or a dedicated partition on the device's built-in memory. Unlike internal storage, external storage is accessible to other apps and the user via the device's file manager, making it suitable for files intended to be shared, such as photos, downloaded documents, or audio recordings.[24], [46]

The main drawback of external storage is that its accessibility also represents its primary risk. Since any app with the appropriate permissions can read from this shared area, sensitive or private data should never be stored there. Furthermore, external storage is not always available (a removable SD card can be removed by the user at any time). Therefore, any code accessing external storage must first verify that it is mounted and accessible.

To check the availability of external storage, the Environment.getExternalStorageState() method can be used. If the external storage is mounted and accessible for reading and writing, the method returns Environment.MEDIA_MOUNTED. If the external storage is not available, the method returns Environment.MEDIA_UNMOUNTED or Environment.MEDIA_UNMOUNTED_READ_ONLY.

To access external storage, the app must request the necessary permissions from the user at runtime, as external storage permissions are considered dangerous. The user can grant or deny these permissions via a permission dialog.

To manage files on external storage, file I/O operations such as creating, reading, writing, and deleting files can be used, just as with internal storage. However, when accessing external storage, the app must specify the storage location using methods such as getExternalFilesDir(), getExternalCacheDir(), or getExternalStoragePublicDirectory().

External storage is suitable for storing user-generated files or multimedia content, such as photos, videos, and music. It is also suitable for files that need to be shared across multiple applications or devices. However, sensitive or confidential data should not be stored on external storage due to the security risks associated with shared storage.

Here is a code example that demonstrates how to use file I/O operations to manage files on external storage:

```
1. public class MainActivity extends AppCompatActivity {
2.
3.     private static final int REQUEST_EXTERNAL_STORAGE = 1;
4.     private static String[] PERMISSIONS_STORAGE = {
5.         Manifest.permission.READ_EXTERNAL_STORAGE,
6.         Manifest.permission.WRITE_EXTERNAL_STORAGE
7.     };
8.
9.     @Override
10.    protected void onCreate(Bundle savedInstanceState) {
11.        super.onCreate(savedInstanceState);
12.        setContentView(R.layout.activity_main);
13.
14.        // Check if the app has permission to access external storage
15.        if (ContextCompat.checkSelfPermission(this,
16.                Manifest.permission.WRITE_EXTERNAL_STORAGE) !=
PackageManager.PERMISSION_GRANTED) {
17.            // Request permission to access external storage
18.            ActivityCompat.requestPermissions(this, PERMISSIONS_STORAGE,
REQUEST_EXTERNAL_STORAGE);
19.        } else {
20.            // Access and manage files on external storage
21.            File directory = Environment.getExternalStorageDirectory();
22.            File file = new File(directory, "example.txt");
23.
24.            // Write data to file
25.            try {
26.                FileOutputStream outputStream = new
FileOutputStream(file);
27.                outputStream.write("Hello, World!".getBytes());
28.                outputStream.close();
29.            } catch (IOException e) {
30.                e.printStackTrace();
31.            }
32.
```

```
33.             // Read data from file
34.             try {
35.                 FileInputStream inputStream = new FileInputStream(file);
36.                 BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
37.                 String line = reader.readLine();
38.                 Log.d("MainActivity", line);
39.                 inputStream.close();
40.             } catch (IOException e) {
41.                 e.printStackTrace();
42.             }
43.
44.             // Delete file
45.             file.delete();
46.         }
47.     }
48.
49.     // Handle permission request result
50.     @Override
51.     public void onRequestPermissionsResult(int requestCode, String[]
permissions, int[] grantResults) {
52.         super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
53.
54.         if (requestCode == REQUEST_EXTERNAL_STORAGE) {
55.             if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
56.                 // Permission granted, access and manage files on
external storage
57.                 File directory =
Environment.getExternalStorageDirectory();
58.                 // ...
59.             } else {
60.                 // Permission denied
61.                 Toast.makeText(this, "Permission denied",
Toast.LENGTH_SHORT).show();
62.             }
63.         }
64.     }
65. }
```

6.3.3 Shared Preferences

Shared preferences are a lightweight mechanism for storing small amounts of data as key-value pairs within an app. They provide a simple and efficient way to store and retrieve app settings, user preferences, and other small amounts of data that need to be retained across app sessions. [23], [47]

Shared preferences are stored in an XML file in the app's internal storage and can be accessed from any activity or service within the app. To use shared preferences, you must first obtain a reference to the SharedPreferences object by calling the getSharedPreferences()

method. This method takes two arguments: a unique name to identify the preferences file and a mode that determines the file's visibility to other apps.

The benefits of using shared preferences are as follows

- Ease of use: Shared preferences provide a simple, easy-to-use mechanism for storing and retrieving small amounts of data.
- Efficient: Since SharedPreferences are stored in an XML file in the app's internal storage, they can be accessed quickly and efficiently.
- Automatic data management: SharedPreferences are automatically managed by the system, including the creation, writing, and reading of files.

The disadvantages of using shared preferences are as follows

- Limited storage capacity: Shared preferences are designed to store small amounts of data; therefore, they are not suitable for storing large amounts of data.
- Limited data types: Shared preferences support only a limited set of data types, including booleans, integers, floats, longs, and strings.
- No security: Shared preferences offer no security features. Therefore, sensitive data should not be stored using this mechanism.

Shared preferences are suitable for storing small amounts of data that need to be retained during application sessions, such as user preferences, settings, or small amounts of user-generated data. They are not suitable for storing large amounts of data or sensitive information.

Here is an example of using shared preferences in an Android application from the main activity

```
1. import android.content.Context;
2. import android.content.SharedPreferences;
3. import androidx.appcompat.app.AppCompatActivity;
4. import android.os.Bundle;
5. import android.widget.TextView;
6.
7. public class MainActivity extends AppCompatActivity {
8.
9.     private static final String MY_PREFS_NAME = "MyPrefsFile";
10.    private TextView textView;
11.
12.    @Override
13.    protected void onCreate(Bundle savedInstanceState) {
14.        super.onCreate(savedInstanceState);
15.        setContentView(R.layout.activity_main);
16.
17.        textView = findViewById(R.id.text_view);
```

```
18.
19.         // Get shared preferences
20.         SharedPreferences prefs = getSharedPreferences(MY_PREFS_NAME,
MODE_PRIVATE);
21.
22.         // Write to shared preferences
23.         SharedPreferences.Editor editor = prefs.edit();
24.         editor.putString("username", "John Doe");
25.         editor.putInt("age", 30);
26.         editor.apply();
27.
28.         // Read from shared preferences
29.         String username = prefs.getString("username", "");
30.         int age = prefs.getInt("age", 0);
31.
32.         // Display values
33.         textView.setText("Username: " + username + "\nAge: " + age);
34.     }
35. }
```

6.4 Remote Data Persistence with Firebase

All the storage mechanisms discussed so far (SQLite, Room, files, and shared preferences) store data locally on the user's device. Local storage is fast, works offline, and requires no network infrastructure, but it has a fundamental limitation: the data exists on a single device and cannot be shared between users or accessed from another device. When an app needs to sync data across multiple devices, share content between users, or back up data to a server, a remote storage solution is required.

Firebase is a platform developed by Google that provides a suite of back-end services for mobile and web apps. Among its numerous services, Firebase Real-time Database and Cloud Fire-store are the two most commonly used database services in Android apps. Both allow an app to store and retrieve data from Google's cloud infrastructure, synchronize data in real-time across all connected clients, and continue to function offline by caching data locally and synchronizing it when the connection is restored[50], [51].

This section focuses on Firebase Real-time Database, which is the original Firebase database service. Firebase stores all data as a single large JSON tree and instantly pushes updates to all connected clients, without the app needing to query the server to detect changes. It's a must-have for apps that require real-time collaborative data: chat apps, real-time whiteboards, shared to-do lists, and other similar use cases.

6.4.1 What is Firebase Real-time Database

The Firebase Real-time Database is a cloud-hosted NoSQL database that stores data in JSON format and synchronizes it in real-time to every connected client. Unlike a relational database such as SQLite, which organizes data into tables and rows, the Realtime Database organizes data as a tree of nested JSON objects. Each piece of data is stored at a path in the tree, and any client that holds a reference to that path can read or write to it, subject to the security rules configured in the Firebase console[50], [51].

The most important feature of the Real-time Database is its real-time synchronization. When data at a given path changes (whether modified by the current user, another user on a different device, or a server-side process) all clients that are listening to that path receive the update immediately, without any explicit polling or refresh request. This push-based model is what makes Firebase particularly well suited to collaborative and live-data applications.

Firebase also includes a robust offline capability. When a device loses network connectivity, the Firebase SDK continues to serve reads from a local cache and queues write operations. When connectivity is restored, the queued writes are automatically replayed against the server, and the local cache is updated with any changes that occurred while offline. From the application's perspective, the database simply works and the transition between online and offline states is handled entirely by the Firebase SDK.

6.4.2 Setting Up Firebase

Setting up Firebase requires connecting your local Android Studio project to the Google Cloud infrastructure. This process is divided into three main phases: Console Configuration, Configuration File integration, and Dependency management.

1. **Step 1: Console Configuration**

Before writing any code, you must register your app in the Firebase Console.

- **Create a Project:** In the console, create a new Firebase project.
- **Register the App:** Provide your Android **Package Name** (e.g., com.example.myapp). This must exactly match the applicationId in your app-level Gradle file.
- **Download Configuration:** Firebase generates a unique google-services.json file.

2. **Step 2: Adding the Configuration File**

The google-services.json file contains API keys and project identifiers used by the Firebase SDK. To place it in the correct path you must switch your Project view in Android Studio to "Project" mode and place the file in the app/ directory.

3. **Step 3: Gradle Configuration (Kotlin DSL)**

The example project uses Kotlin DSL (.kts files), we use the following syntax to apply the necessary plugins and libraries.

- Project-level (build.gradle.kts)

First, add the Google Services plugin to the build script. This allows the app to process the JSON file downloaded earlier.

```
// Root build.gradle.kts
plugins {
    id("com.google.gms.google-services") version "4.4.0" apply false
}
```

- App-level (app/build.gradle.kts): Apply the plugin and add the Firebase Real-time Database dependencies.

```
plugins {
    id("com.android.application")
    id("com.google.gms.google-services") // Must be applied last
}

dependencies {
    // Firebase Bill of Materials (BoM)
    implementation(platform("com.google.firebase:firebase-bom:32.7.0"))

    // Realtime Database (No version needed because of the BoM)
    implementation("com.google.firebase:firebase-database")
```

6.4.3 The Three Core Concepts

Working with the Firebase Realtime Database revolves around three concepts: the database reference, writing data, and reading data. A DatabaseReference is a pointer to a specific location in the JSON tree. All read and write operations are performed through a reference. The root reference is obtained from the Firebase Database singleton, and child references are obtained by calling child() with the path segment name.

```
// Get a reference to the root of the database
FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference rootRef = database.getReference();

// Get a reference to a specific path
DatabaseReference usersRef = rootRef.child("users");
```

```
DatabaseReference user1Ref = usersRef.child("user_001");

// Equivalent shorthand
DatabaseReference user1Ref2 = database.getReference("users/user_001");
```

Writing data to the database is done by calling setValue() on a reference. This method accepts any Java object, a Map, or a primitive value, and serializes it to JSON before sending it to the server. The write operation is asynchronous which means it completes in the background, and an optional completion listener can be attached to handle success or failure. To write to a specific child without overwriting siblings, updateChildren() is used with a Map of the fields to update

```
// Writing a Java object — Room uses @Entity, Firebase uses a Plain Old Java Object (POJO)
public class User {
    public String firstName;
    public String email;

    public User() {} // Required empty constructor for Firebase

    public User(String firstName, String email) {
        this.firstName = firstName;
        this.email     = email;
    }
}

// Write a user to the database
DatabaseReference usersRef = FirebaseDatabase.getInstance().getReference("users");
String userId = usersRef.push().getKey(); // Auto-generate a unique key
usersRef.child(userId).setValue(new User("Alice", "alice@example.com"))
    .addOnSuccessListener(aVoid -> Log.d("TAG", "User written"))
    .addOnFailureListener(e  -> Log.e("TAG", "Write failed", e));
```

Reading data is done by attaching a listener to a reference. There are two kinds of listeners: a one-time listener that reads the value once and then detaches, and a persistent listener that continues to receive updates whenever the data at that path changes. The persistent listener is what enables real-time synchronization which means that as soon as any client writes new data, all attached listeners are notified immediately.

```
DatabaseReference usersRef =
    FirebaseDatabase.getInstance().getReference("users");

// ONE-TIME READ — reads current value and detaches
usersRef.get().addOnSuccessListener(snapshot -> {
    for (DataSnapshot child : snapshot.getChildren()) {
```

```
        User user = child.getValue(User.class);
        Log.d("TAG", "User: " + user.firstName);
    }
});

// REAL-TIME LISTENER — fires every time data changes
usersRef.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot snapshot) {
        for (DataSnapshot child : snapshot.getChildren()) {
            User user = child.getValue(User.class);
            Log.d("TAG", "Updated user: " + user.firstName);
        }
    }

    @Override
    public void onCancelled(DatabaseError error) {
        Log.e("TAG", "Read failed: " + error.getMessage());
    }
});
```

6.4.4 A Complete Example of Users List with Firebase

The following example mirrors the SQLite-based Users application built earlier in the chapter but uses Firebase Real-time Database as the back end instead of local SQLite. The application displays a real-time list of users that updates automatically whenever any client adds or removes a user.

a. The User Model

```
// User.java — plain Java object (POJO)
// Firebase requires a public no-argument constructor
public class User {
    public String firstName;
    public String email;

    public User() {}

    public User(String firstName, String email) {
        this.firstName = firstName;
        this.email     = email;
    }
}
```

b. The Repository

```
// UserRepository.java
public class UserRepository {

    private DatabaseReference usersRef;

    public UserRepository() {
        usersRef = FirebaseDatabase.getInstance().getReference("users");
    }

```

```
    // Add a new user — Firebase auto-generates a unique key
    public void addUser(User user) {
        String key = usersRef.push().getKey();
        usersRef.child(key).setValue(user);
    }

    // Delete a user by their Firebase key
    public void deleteUser(String userId) {
        usersRef.child(userId).removeValue();
    }

    // Attach a real-time listener — caller receives updates
automatically
    public void listenForUsers(ValueEventListener listener) {
        usersRef.addValueEventListener(listener);
    }

    // Detach the listener when no longer needed
    public void stopListening(ValueEventListener listener) {
        usersRef.removeEventListener(listener);
    }
}

```

c. MainActivity

```
// MainActivity.java (key parts)
public class MainActivity extends AppCompatActivity {

    private UserRepository repository;
    private UserAdapter adapter;
    private ValueEventListener userListener;
    private EditText etFirstName, etEmail;

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

        repository = new UserRepository();
        etFirstName = findViewById(R.id.etFirstName);
        etEmail     = findViewById(R.id.etEmail);

        // Setup RecyclerView
        RecyclerView recyclerView = findViewById(R.id.recyclerView);
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
        adapter = new UserAdapter((userId, user) -> {
            repository.deleteUser(userId);
        });
        recyclerView.setAdapter(adapter);

        // Attach real-time listener
        userListener = new ValueEventListener() {
            @Override
            public void onDataChange(DataSnapshot snapshot) {
                List<Map.Entry<String, User>> users = new ArrayList<>();
                for (DataSnapshot child : snapshot.getChildren()) {
                    User user = child.getValue(User.class);
```

```
                    users.add(new
AbstractMap.SimpleEntry<>(child.getKey(), user));
                }
                adapter.setUsers(users); // UI updates automatically
            }
            @Override
            public void onCancelled(DatabaseError error) {
                Log.e("TAG", error.getMessage());
            }
        };
        repository.listenForUsers(userListener);

        // Add user on button click
        findViewById(R.id.btnAddUser).setOnClickListener(v -> {
            String firstName = etFirstName.getText().toString().trim();
            String email     = etEmail.getText().toString().trim();
            if (firstName.isEmpty() || email.isEmpty()) {
                Toast.makeText(this, "Fill all fields",
                               Toast.LENGTH_SHORT).show();
                return;
            }
            repository.addUser(new User(firstName, email));
            etFirstName.setText("");
            etEmail.setText("");
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Always detach listeners to prevent memory leaks
        repository.stopListening(userListener);
    }
}
```

Advice

Listeners must always be detached when the Activity is destroyed, typically in onDestroy(). Failing to remove a listener causes a memory leak, because Firebase holds a reference to the Activity through the listener, preventing it from being garbage collected.

Warning: Security Rules

By default, a newly created Firebase project allows anyone with the database URL to read and write all data, these settings intended only for rapid prototyping that must never be used in production. Firebase Security Rules are a declarative language written in the Firebase console that control who can read or write each path in the database. Rules are

evaluated server-side before any read or write is allowed, meaning they cannot be bypassed by client-side code.

A minimal but safer rule set requires that the user be authenticated before reading or writing any data. Firebase offers another service called Firebase Authentication which provides ready-made sign-in flows for email/password, Google, Facebook, and other providers. Once a user is signed in, their unique UID is available in the security rules as request.auth.uid, allowing rules to restrict each user to their own data.

```
// Firebase Security Rules (configured in the Firebase console)

// ❌ Default — anyone can read/write everything (development only!)
{
  "rules": {
    ".read": true,
    ".write": true
  }
}

// ✅ Safer — only authenticated users can read/write
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}

// ✅ Best — each user can only read/write their own data
{
  "rules": {
    "users": {
      "$uid": {
        ".read":  "auth != null && auth.uid === $uid",
        ".write": "auth != null && auth.uid === $uid"
      }
    }
  }
}
```

Conclusion

In summary, the development of mobile applications is a discipline that bridges the gap between abstract software engineering and the rigorous constraints of physical hardware. As explored throughout this document, mastering this field requires more than just learning a programming language; it demands a deep understanding of how various components from the UI layers to the underlying persistence engines interact within a volatile operating environment. By moving through the phases of analysis and synthesis, students have gained the necessary tools to build applications that are not only functional but also resilient to the lifecycle interruptions inherent to mobile systems.

The concepts presented in this course, particularly regarding frontend architecture and resource management, serve as the essential foundation for any aspiring developer.

As noted at the outset, the rapid evolution of this technology means that learning is a continuous process. This course is intended to be a living document, and its improvement depends heavily on the engagement of its readers. I remain committed to refining this work and encourage students and colleagues alike to share their observations and corrections, ensuring that our academic resources stay as dynamic as the field they describe.

References

[1] T. Farley, ‘Mobile telephone history’, *Priv. Com Httpwww Priv. Comwp-Contentuploads201601TelenorPage022-034 Pdf*, 2005, Accessed: Apr. 21, 2026. [Online]. Available: https://www.academia.edu/download/30805674/T05_3-4.pdf#page=24

[2] J. Agar, *Constant touch: A global history of the mobile phone*. Icon Books Ltd, 2013. Accessed: Apr. 21, 2026. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=sBNfZNqcOzoC&oi=fnd&pg=PT6&dq=mobile+telephone+history&ots=o04xAWNrdA&sig=URiOue58xiegxC5m6kClegfHm_k

[3] ‘Architecture overview’, Android Open Source Project. Accessed: Apr. 20, 2026. [Online]. Available: https://source.android.com/docs/core/architecture

[4] ‘Open Handset Alliance’. Accessed: Apr. 21, 2026. [Online]. Available: https://www.openhandsetalliance.com/

[5] ‘Mobile Operating System Market Share Worldwide’, StatCounter Global Stats. Accessed: Apr. 21, 2026. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[6] S. Morrissey, ‘History of Apple Mobile Devices’, in *iOS Forensic Analysis for iPhone, iPad, and iPod touch*, S. Morrissey, Ed., Berkeley, CA: Apress, 2010, pp. 1–23. doi: 10.1007/978-1-4302-3343-5_1.

[7] S. Morrissey, ‘iOS Operating and File System Analysis’, in *iOS Forensic Analysis for iPhone, iPad, and iPod touch*, S. Morrissey, Ed., Berkeley, CA: Apress, 2010, pp. 25–66. doi: 10.1007/978-1-4302-3343-5_2.

[8] ‘About the Service-Account Kit - HUAWEI Developers’. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.huawei.com/consumer/en/doc/HMSCore-Guides/introduction-0000001050048870

[9] Huawei Device Co., Ltd., ‘HarmonyOS 2 Security Technical White Paper V2.0’, Huawei Device Co., Ltd., White paper V2.0, 2021.

[10] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, ‘A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development’, *ACM Comput. Surv.*, vol. 51, pp. 1–34, Nov. 2018, doi: 10.1145/3241739.

[11] P. R.M.de Andrade, A. B.Albuquerque, O. F. Frota, R. V Silveira, and F. A. Da Silva, ‘Cross Platform App : A Comparative Study’, *Int. J. Comput. Sci. Inf. Technol.*, vol. 7, no. 1, pp. 33–40, Feb. 2015, doi: 10.5121/ijcsit.2015.7104.

[12] C. Banga and J. Weinhold, *Essential Mobile Interaction Design: Perfecting Interface Design in Mobile Apps*. Upper Saddle River, NJ: Addison-Wesley Professional, 2014.

[13] M. Hart, ‘The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses Eric Ries. New York: Crown Business, 2011. 320 pages. US$26.00.’, *J. Prod. Innov. Manag.*, vol. 29, May 2012, doi: 10.1111/j.1540-5885.2012.00920_2.x.

[14] A. Osterwalder, Y. Pigneur, G. Bernarda, A. Smith, and T. Papadakos, *Value Proposition Design: How to Create Products and Services Customers Want*. Hoboken: Wiley, 2014.
[15] I. Sommerville, 'Part 1 Introduction to Software Engineering', in *Software engineering*, Tenth edition., in Always learning. , Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London: Pearson, 2016, pp. 15–255.
[16] 'App review process and requirements for the Google Workspace Marketplace', Google for Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developers.google.com/workspace/marketplace/about-app-review
[17] A. Inc, 'App Review Guidelines', Apple Developer. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.apple.com/app-store/review/guidelines/
[18] *Systems and software engineering. Systems and software quality requirements and evaluation (SQuaRE). System and software quality models:* doi: 10.3403/30215101.
[19] I. Sommerville, 'Part 2 System Dependability and Security', in *Software engineering*, Tenth edition., in Always learning. , Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London: Pearson, 2016, pp. 238–208.
[20] 'Platform architecture', Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/platform
[21] 'Application fundamentals | App architecture', Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/components/fundamentals
[22] 'Introduction to activities | App architecture', Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities
[23] P. J. Deitel, Deitel & Associates, inc, H. M. Deitel, and A. Wald, *Android 6 for programmers: an app-driven approach*, Third edition. in Deitel® developer series. Boston Columbus Indianapolis: Prentice Hall, 2016.
[24] B. Sills, B. Gardner, K. Marsicano, and C. Stewart, *Android Programming: The Big Nerd Ranch Guide*. Addison Wesley Professional, 2022.
[25] 'Services overview | Background work | Android Developers'. Accessed: Apr. 30, 2026. [Online]. Available: https://developer.android.com/develop/background-work/services
[26] 'Broadcasts overview | Background work | Android Developers'. Accessed: Apr. 30, 2026. [Online]. Available: https://developer.android.com/develop/background-work/background-tasks/broadcasts
[27] 'Content provider basics | App data and files | Android Developers'. Accessed: Apr. 30, 2026. [Online]. Available: https://developer.android.com/guide/topics/providers/content-provider-basics
[28] 'Run apps on the Android Emulator | Android Studio | Android Developers'. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/studio/run/emulator
[29] 'The activity lifecycle | App architecture', Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle
[30] P. Deitel and H. Deitel, *Android How to Program*. Boston Munich: Pearson, 2017.
[31] 'Intents and intent filters | App architecture', Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/components/intents-filters
[32] 'App manifest overview | App architecture', Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro

[33] ‘App resources overview | App architecture’, Android Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/guide/topics/resources/providing-resources
[34] ‘Layouts in views | Views’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/layout/declaring-layout
[35] ‘Create a linear layout | Views’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/layout/linear
[36] ‘Relative Layout | Views | Android Developers’. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/layout/relative
[37] ‘Build a responsive UI with ConstraintLayout | Views | Android Developers’. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/layout/constraint-layout
[38] ‘Fragment lifecycle | App architecture | Android Developers’. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/guide/fragments/lifecycle
[39] ‘android.widget | API reference’, Android Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/reference/android/widget/package-summary
[40] ‘UI events | App architecture’, Android Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/topic/architecture/ui-layer/events
[41] ‘Input events overview | Views’, Android Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/touch-and-input/input-events
[42] ‘Add menus | Views’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/components/menus
[43] ‘Dialogs | Views’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/develop/ui/views/components/dialogs
[44] ‘BottomSheetDialog | API reference | Android Developers’. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/reference/com/google/android/material/bottomsheet/BottomSheetDialog
[45] ‘Save UI states | App architecture’, Android Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/topic/libraries/architecture/saving-states
[46] ‘Data and file storage overview | App data and files’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/training/data-storage
[47] ‘Save simple data with SharedPreferences | App data and files’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/training/data-storage/shared-preferences
[48] ‘Save data using SQLite | App data and files’, Android Developers. Accessed: Apr. 29, 2026. [Online]. Available: https://developer.android.com/training/data-storage/sqlite
[49] ‘Save data in a local database using Room | App data and files’, Android Developers. Accessed: Apr. 20, 2026. [Online]. Available: https://developer.android.com/training/data-storage/room
[50] ‘Firebase Realtime Database’, Firebase. Accessed: Apr. 20, 2026. [Online]. Available: https://firebase.google.com/docs/database
[51] ‘Read and Write Data on Android | Firebase Realtime Database’, Firebase. Accessed: Apr. 20, 2026. [Online]. Available: https://firebase.google.com/docs/database/android/read-and-write