Android App Development

Table of Contents

Android Development Environment

1. Android Studio Overview

Android Studio is the official Integrated Development Environment (IDE) for Android development, built on IntelliJ IDEA. It provides tools for building, testing, and debugging Android applications efficiently.

Key Features

- **Code Editor**: Advanced code editing with suggestions, refactoring, and linting tools.
- **Layout Editor**: Drag-and-drop interface for designing UI components.
- **Emulator**: Built-in emulator to test applications without a physical device.
- **Gradle Integration**: Build automation and dependency management.
- **Debugging Tools**: Profiler for memory, CPU, and network usage.
- **Version Control**: Supports Git, SVN, and more.

---

2. Installing and Configuring Android Studio

System Requirements

- **Windows**: Windows 10 or later, x64 processor.
- **Mac**: macOS 10.14 or later.
- **Linux**: GNOME or KDE desktop environment.
- Minimum 8 GB RAM (16 GB recommended).
- Minimum 4 GB disk space (SSD recommended).

Installation Steps

1. **Download Android Studio**:
    - Visit Android Studio Download Page.
    - Choose the appropriate version for your OS.
2. **Install**:
    - Run the installer and follow on-screen instructions.
    - Select additional components like the Android Virtual Device (AVD).
3. **Initial Setup**:
    - Launch Android Studio.
    - Configure SDK path and required SDK packages.
    - Complete the "Setup Wizard."

---

3. Android SDK and Tools

Overview

The Android Software Development Kit (SDK) is a set of tools necessary for developing Android applications. It includes libraries, tools, and APIs.

Key Components

- **SDK Manager**: Manages SDK versions and updates.
- **Android Debug Bridge (ADB)**: Command-line tool for debugging and device communication.
- **AVD Manager**: Creates and manages Android Virtual Devices (emulators).
- **Build Tools**: Includes compilers and utilities for building applications.
- **Platform Tools**: Tools specific to different Android versions.

Configuring SDK

1. Open Android Studio.
2. Go to **File > Settings > Appearance & Behavior > System Settings > Android SDK**.
3. Select required SDK Platforms and Tools.
4. Apply and download the necessary components.

---

4. File Structure in Android Studio

An Android project in Android Studio follows a structured hierarchy:

Key Directories

- **app/**: Contains source code and resources.
    - **java/**: Source code for your application.
    - **res/**: Resources like layouts, strings, drawables, and more.
    - **AndroidManifest.xml**: Application configuration file.
- **gradle/**: Build system files.
- **build/**: Generated build outputs.
- **.idea/**: IDE-specific configuration files.

Build Gradle Files

- **build.gradle (Project Level)**: Defines build script repositories and dependencies.
- **build.gradle (Module Level)**: Specifies app-level dependencies, compile SDK version, and more.

5. Android Versions

Overview

Android versions are released with specific codenames and API levels. These versions define new features, APIs, and behavioral changes.

List of Android Versions

Version	Codename	API Level
Android 1.0	No codename	1
Android 1.1	Petit Four	2
Android 1.5	Cupcake	3
Android 1.6	Donut	4
Android 2.0/2.1	Eclair	5-7
Android 2.2	Froyo	8
Android 2.3	Gingerbread	9-10
Android 3.x	Honeycomb	11-13
Android 4.0	Ice Cream Sandwich	14-15
Android 4.1-4.3	Jelly Bean	16-18
Android 4.4	KitKat	19-20
Android 5.x	Lollipop	21-22
Android 6.0	Marshmallow	23
Android 7.x	Nougat	24-25
Android 8.x	Oreo	26-27
Android 9	Pie	28
Android 10	Quince Tart	29
Android 11	Red Velvet Cake	30
Android 12	Snow Cone	31
Android 12L	Snow Cone v2	32

Android 13	Tiramisu	33
Android 14	**Upside Down Cake**	**34**
Android 15	**Vanilla Ice Cream**	**35**
Android 16	Watermelon Sorbet	36

API Levels

Each Android version corresponds to a unique API level. Apps must specify a **minimum SDK version** and **target SDK version** to ensure compatibility.

Design Tools

Overview of Canva

Canva is a user-friendly online design tool that enables users to create professional-quality designs for various purposes. It is widely used for creating graphics, presentations, social media posts, and more without requiring advanced design skills.

Key Features

1. **Drag-and-Drop Interface**:
   - Easy to use, allowing quick customization of templates.
2. **Templates**:
   - A vast library of pre-designed templates for presentations, posters, social media, and more.
3. **Elements**:
   - Access to millions of stock images, icons, shapes, and fonts.
4. **Collaborative Tools**:
   - Share designs with team members for real-time collaboration.
5. **Animations**:
   - Add motion effects to text and images for dynamic designs.
6. **Brand Kit**:
   - Store brand-specific colors, logos, and fonts for consistent branding.
7. **Export Options**:
   - Export designs in various formats such as PNG, JPG, PDF, and MP4.

Using Canva for Presentations (PPTs)

Canva provides dedicated tools for creating visually appealing presentations. Below are the steps to create a presentation using Canva:

Steps to Create a Presentation

1. **Login or Sign Up**:
   - Go to Canva and log in or create an account.
2. **Choose a Template**:
   - Select "Presentation" from the templates or search for a specific theme.
3. **Customize the Design**:
   - Add or modify slides using drag-and-drop functionality.
   - Include text, images, and videos as needed.
4. **Apply Animations**:
   - Use transition effects and animations to enhance slide visuals.
5. **Collaborate (Optional)**:
   - Share the design link with teammates for real-time collaboration.
6. **Export the Presentation**:
   - Download as a PowerPoint file (PPTX) or present directly from Canva.

Making Designs in Microsoft PowerPoint (MS Office)

Microsoft PowerPoint is another powerful tool for creating visually engaging presentations. While primarily used for slide decks, it can also be leveraged for custom graphic designs.

Steps to Create Designs in MS PowerPoint

1. **Open PowerPoint**:
   - Launch Microsoft PowerPoint and create a new blank presentation.
2. **Customize the Slide Size**:
   - Go to **Design > Slide Size > Custom Slide Size** to set dimensions based on your design needs (e.g., poster or banner size).
3. **Use Shapes and Icons**:
   - Access **Insert > Shapes** to add geometric designs.
   - Use **Icons** for ready-made vector graphics.
4. **Insert Images and Videos**:
   - Add visuals by navigating to **Insert > Pictures** or **Insert > Online Pictures**.
5. **Apply Design Themes**:
   - Use **Design > Themes** to apply a consistent visual style.
6. **Add Animations and Transitions**:
   - Use the **Animations** tab to bring motion effects to elements.

7. **Export the Design**:
   - Save your slide as an image (PNG/JPG) or export the entire presentation as a PDF.

Tips for Effective Designs in PowerPoint

- **Layering**: Arrange elements with the **Send to Back** or **Bring to Front** options.
- **Alignment**: Use the **Align** tools under the Arrange menu to ensure consistent spacing.
- **Custom Fonts**: Install and use unique fonts for a distinct style.

Overview of Figma

Figma is a collaborative design tool primarily used for UI/UX design. It operates entirely online, making it accessible from any device with an internet connection. It is widely preferred for creating prototypes, wireframes, and interactive designs.

Key Features

1. **Real-Time Collaboration**:
   - Multiple users can work on the same design simultaneously.
2. **Cross-Platform Accessibility**:
   - Works on web browsers, eliminating the need for installations.
3. **Design and Prototyping**:
   - Seamless transition from design to interactive prototypes.
4. **Version Control**:
   - Keeps track of changes with an autosave and version history feature.
5. **Plugins**:
   - Extensive plugin library for icons, stock images, and additional functionalities.
6. **Component System**:
   - Reusable components for maintaining design consistency.
7. **Integrations**:
   - Integrates with tools like Slack, Jira, and Zeplin for streamlined workflows.

Using Figma for Designs

Figma provides powerful tools to create designs collaboratively and efficiently. Here are the steps to use Figma:

Steps to Create a Design in Figma

1. **Sign Up or Login**:

   - Go to Figma and log in or create an account.
2. **Create a New File**:
   - Click on "New File" to start designing.
3. **Use Frames**:
   - Add frames (artboards) to define your workspace.
4. **Design with Tools**:
   - Use shapes, pen tools, and text tools to create layouts.
   - Drag and drop images or icons into your design.
5. **Prototyping**:
   - Link frames to create interactive prototypes.
6. **Collaborate**:
   - Share the file link with team members for feedback and edits.
7. **Export the Design**:
   - Export frames or elements as PNG, JPG, or SVG formats.

Tips for Using Figma Effectively

- **Leverage Plugins**: Use plugins for tasks like icon insertion, stock photo access, or wireframe creation.
- **Organize Layers**: Name layers and group related elements for easy navigation.
- **Master Components**: Create components for reusable elements like buttons and headers.

Advantages of Design Tools

Canva

1. **Ease of Use**: Intuitive interface suitable for beginners.
2. **Time-Saving**: Pre-designed templates speed up the design process.
3. **Wide Accessibility**: Works on browsers and mobile apps.

PowerPoint

1. **Offline Access**: Fully functional without an internet connection.
2. **Robust Customization**: Advanced features for creating unique designs.
3. **Familiar Interface**: Ideal for users accustomed to MS Office.

Figma

1. **Collaboration**: Real-time design edits with team members.
2. **Cross-Platform**: Works on any device with internet access.
3. **Prototyping**: Interactive mockups for UI/UX workflows.

Limitations of Design Tools

Canva

1. **Limited Customization**: May not satisfy advanced design needs.
2. **Premium Content**: Some features require a paid plan.

PowerPoint

1. **Design Constraints**: Less suited for modern design needs.
2. **Learning Curve**: Advanced features may require practice.

Figma

1. **Internet Dependency**: Requires a stable connection.
2. **Resource Intensive**: May lag on lower-spec devices.

History of Android

Android is an open-source operating system for mobile devices, developed by Google and based on the Linux kernel. Its development began in October 2003 by Android Inc., a company founded by Andy Rubin, Rich Miner, Nick Sears, and Chris White. Initially, Android was created to support the development of advanced mobile operating systems that could compete with Symbian and Windows Mobile.

In 2005, Google acquired Android Inc., marking the beginning of the platform’s association with Google. The first commercial version, Android 1.0, was released in September 2008, alongside the HTC Dream (also known as the T-Mobile G1). Over time, Android evolved through numerous versions, each introducing new features and improvements. The platform's popularity grew rapidly, and it became the world’s most widely used mobile operating system by the early 2010s.

What is Android?

Android is an open-source, Linux-based operating system primarily designed for touchscreen mobile devices such as smartphones and tablets. It provides a flexible and customizable platform that supports a wide variety of hardware configurations. Android features a user-friendly interface, support for multitasking, and an extensive library of applications available through the Google Play Store. It also supports wearables, TVs, automotive systems, and IoT devices.

Key features of Android include:

- A customizable user interface.
- Multitasking capabilities.
- Support for third-party applications and services.
- Integration with Google’s ecosystem, including Gmail, Google Maps, and Google Assistant.
- Regular updates to enhance security and functionality.

Why is there a need for Android?

The need for Android arose from the limitations of earlier mobile operating systems and the demand for a versatile, user-friendly platform capable of running on a wide range of devices. The primary reasons for the development of Android include:

1. **Open Ecosystem:** Android’s open-source nature allows device manufacturers and developers to customize and innovate freely, fostering diversity in hardware and software solutions.
2. **Affordability:** Android’s flexibility makes it possible for manufacturers to create devices at various price points, catering to a broad audience.
3. **Developer-Friendly:** Android’s support for Java and Kotlin programming languages, along with comprehensive development tools, encourages app creation and contributes to the vibrant app ecosystem.
4. **Consumer Demand:** The growing demand for smart devices required a platform that could adapt to various use cases, such as mobile phones, tablets, TVs, and more.
5. **Integration with Services:** Android’s seamless integration with Google services enhances user experience, productivity, and connectivity.

Origin of Standard Enterprise Platforms: ME and EE

1. **Java ME (Micro Edition):** Java ME is a subset of the Java programming language designed for resource-constrained devices, such as mobile phones and embedded systems. Introduced by Sun Microsystems (later acquired by Oracle) in 1999, Java ME provided a platform-independent environment for developing applications on small devices. It supported a wide range of applications, from simple games to complex mobile software.

   Java ME uses configurations like the Connected Limited Device Configuration (CLDC) and profiles such as the Mobile Information Device Profile (MIDP) to cater to the specific requirements of constrained devices. These frameworks were instrumental in bringing Java-based applications to early mobile devices, laying the groundwork for future mobile platforms, including Android.

   While Android does not directly rely on Java ME, its design philosophy and use of the Java programming language for app development were inspired by the success of Java ME. The transition from Java ME to Android represented a shift from constrained, resource-limited environments to more powerful and versatile devices capable of running full-fledged operating systems.

2. **Java EE (Enterprise Edition):** Java EE, now known as Jakarta EE, is a set of specifications and tools for building robust, large-scale, distributed, and enterprise-level applications. Introduced in 1998, it was designed to simplify application development by providing standardized APIs and services such as servlets, JSPs, and EJBs. It plays a significant role in enterprise computing, enabling businesses to build secure and scalable applications.

Together, these platforms represent the evolution of Java to address diverse application needs, from lightweight devices (ME) to enterprise-grade solutions (EE).

Why Android as a Software Stack?

Android operates as a comprehensive software stack, providing tools, frameworks, and services necessary for application development and deployment. This stack includes:

1. **Build Tools:**
   - **Maven, Gradle, and ANT:** These tools are essential for managing dependencies, automating builds, and packaging applications. Among these, Gradle is the most widely used for Android development due to its flexibility and powerful build system.

- **How Gradle Works as a Package Manager System:** Gradle is a build automation tool that simplifies dependency management and the build process. It uses a Groovy or Kotlin-based Domain-Specific Language (DSL) for configuring builds. Gradle integrates seamlessly with Android Studio and handles tasks such as:
  - Resolving dependencies from online repositories like Maven Central or JCenter.
  - Managing build variants (e.g., debug vs. release).
  - Packaging APKs (Android application packages).
  - Automating testing and deployment.

  Gradle's modular approach allows developers to define dependencies and tasks efficiently, making it easier to maintain and scale projects. Unlike Java ME, which lacked such sophisticated build tools, Android's reliance on Gradle enhances productivity and streamlines complex workflows.

Groovy in Android

- **Groovy**: A programming language used primarily in **Gradle scripts** for Android development.
- **Key Uses**:
  - Gradle build files (build.gradle) are written in Groovy by default.
  - Automates tasks such as dependency management and build configurations.
- **Features**:
  - Simplified syntax compared to Java.
  - Dynamic typing and closures for flexible logic.
  - Seamless integration with Java libraries.

Example:

```
android {
   compileSdkVersion 34
   defaultConfig {
      applicationId "com.example.myapp"
      minSdkVersion 21
      targetSdkVersion 34
      versionCode 1
      versionName "1.0"
   }
   dependencies {
      implementation 'androidx.appcompat:appcompat:1.6.1'
   }
}
```

Gradle

- **Gradle**: A flexible build automation tool and the default build system for Android projects.

- **Key Features**:
  1. **Build Automation**: Automates compilation, packaging, and deployment.
  2. **Dependency Management**: Fetches libraries from repositories like Maven Central.
  3. **Incremental Builds**: Optimizes builds by only re-executing necessary tasks.
  4. **Customizable**: Supports custom tasks and build configurations.
- **Configuration Files**:
  - **Project-Level build.gradle**: Configures global settings and repositories.
  - **Module-Level build.gradle**: Defines app/module-specific settings and dependencies.
- **Common Gradle Tasks**:
  - assembleDebug: Builds a debug APK.
  - assembleRelease: Builds a release APK.
  - clean: Cleans up build artifacts.

Example:

```
plugins {
  id 'com.android.application'
}
android {
  compileSdk 34
  defaultConfig {
    applicationId "com.example.myapp"
    minSdk 21
    targetSdk 34
  }
}
```

- **Comparison with Maven**:
  - Faster, supports incremental builds, and uses a flexible Groovy/Kotlin DSL.

Maven

- **Maven**: A build automation and dependency management tool, popular in Java-based projects.
- **Key Features**:
  1. **Dependency Management**: Automatically fetches and manages libraries from repositories.
  2. **Standardized Directory Structure**: Enforces a consistent project layout.
  3. **Build Lifecycle**: Organizes the build process into phases (e.g., validate, compile, test, package).
- **Core Components**:
  - **pom.xml**: Configuration file for project dependencies, plugins, and build settings.
  - **Repositories**: Fetches dependencies from **Maven Central** or custom repositories.

Example **pom.xml**:

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.3.20</version>
    </dependency>
  </dependencies>
</project>
```

- **Comparison with Gradle**:
  - Maven uses XML for configuration, lacks built-in incremental builds, and is less flexible than Gradle.

Ant

- **Apache Ant**: A task-based build automation tool, used before Gradle became standard for Android.
- **Key Features**:
  1. **Task-Based Approach**: Builds are defined as tasks in an XML file (build.xml).
  2. **Extensibility**: Custom tasks can be written in Java.
  3. **No Dependency Management**: Requires manual handling of dependencies.

Example **build.xml**:

```
<project name="MyApp" default="debug" basedir=".">
  <property name="sdk.dir" location="/path/to/android/sdk" />

  <target name="compile">
    <mkdir dir="bin/classes" />
    <javac srcdir="src" destdir="bin/classes" />
  </target>

  <target name="debug" depends="compile">
    <echo message="Building debug APK..." />
  </target>
</project>
```

- **Limitations**:
  - No built-in dependency management.
  - Verbose XML configuration.
  - Poor support for build variants compared to Gradle.
- **Comparison with Gradle**:

- Ant is outdated and less efficient; Gradle is more modern and robust.

Summary Table

Feature	Groovy	Gradle	Maven	Ant
Configuration Style	Groovy scripts	Groovy/Kotlin DSL	XML (pom.xml)	XML (build.xml)
Dependency Management	Via Gradle	Built-in, with transitive support	Built-in with repositories	Manual
Build Variants	Strong (via Gradle)	Strong support	Limited	Limited
Performance	Fast (via Gradle)	Fast (incremental builds)	Slower	Slower
Usage in Android	Core for Gradle scripts	Default build system	Indirect (library hosting)	Rarely used

---

2. **Kotlin Introduction and Purpose:** Kotlin, introduced by JetBrains and officially supported by Google for Android development in 2017, addresses some of the limitations of Java. Kotlin offers:
   - Null safety to reduce runtime crashes.
   - Concise syntax, reducing boilerplate code.
   - Enhanced interoperability with existing Java code.
   - Extension functions for better code modularity.

   Kotlin's modern features improve developer productivity and application quality, making it a preferred language for Android development.

3. **Configuration Using XML and Annotations:**
   - **XML Configuration:** XML files are used extensively in Android for defining layouts, resources, and application configurations (e.g., AndroidManifest.xml). This declarative approach separates the presentation layer from the logic, enhancing modularity and maintainability.
   - **Annotations:** Android leverages annotations to simplify code and reduce runtime overhead. For instance:
     - @Override for method overriding.
     - @NonNull and @Nullable for nullability contracts.
     - @Inject for dependency injection in frameworks like Dagger or Hilt.

By combining XML configuration with annotations, Android provides a balanced approach to application design and development, ensuring flexibility and performance.

Android Design and Architecture (MVC)

Android applications often follow the Model-View-Controller (MVC) architectural pattern to separate concerns and enhance code maintainability:

1. **Views (Design):**
   - The View layer is responsible for the user interface (UI) of the application. This includes XML layout files and widgets like TextView, Button, and RecyclerView.
   - Views handle user input and display data received from the Controller.
   - XML files define the structure and appearance of the UI, while Java/Kotlin code manages event listeners and UI logic.
2. **Controllers (Backend Code):**
   - Controllers serve as the intermediary between Views and Models. They process user input, interact with the database or services, and update the Views.
   - In Android, activities and fragments often act as Controllers. For example:
     - An Activity might handle user navigation and interactions.
     - A Fragment could manage a specific part of the UI and its logic.
3. **Models (DB and Connectivity Services):**
   - The Model layer represents the data and business logic of the application. It includes:
     - **Entity:** Classes that define the structure of data, such as User or Product.
     - **Entity Service:** Logic for retrieving, updating, and storing data in a database or over a network. For example, Room is a popular persistence library in Android that simplifies database operations.
     - Database connectivity services using libraries like Room, SQLite, or Firebase for real-time data.

Android Architecture Components:

1. **LiveData and ViewModel:**
   - LiveData is an observable data holder class that ensures the UI is updated whenever data changes.
   - ViewModel holds and manages UI-related data in a lifecycle-conscious way, ensuring data survives configuration changes (e.g., screen rotations).
2. **Repository Pattern:**
   - Repositories act as a single source of truth for data, abstracting access to multiple data sources (e.g., database, network).
3. **Navigation Component:**
   - Simplifies in-app navigation and ensures a consistent back stack.

By leveraging these components, Android applications achieve better separation of concerns, testability, and scalability.

Android Architecture

Android Architecture Overview

Android architecture consists of the following layers:

1. **Linux Kernel**: Core foundation for hardware drivers, memory management, and security.
2. **Android Runtime (ART)**: Includes the Dalvik virtual machine for running Java applications.
3. **Libraries**: Includes core C/C++ libraries like OpenGL, WebKit, SQLite.
4. **Application Framework**: APIs for building apps (e.g., Activity Manager, Window Manager).
5. **Applications**: The user-facing apps like Messages, Contacts, etc.

Project Structure (Using Gradle)

Default Files and Significance:

1. **build.gradle**: Defines dependencies and build configurations.
2. **AndroidManifest.xml**: Declares app components and permissions.
3. **MainActivity.java/.kt**: Entry point for the application logic.
4. **res/layout/**: XML files defining UI layouts.
5. **res/values/**: XML files for constants like strings, colors, dimensions.
6. **src/main/**: Contains Java/Kotlin code for app functionality.
7. **res/drawable/**: Images and vector graphics.

Gradle Significance:

- Automates build processes (compilation, packaging).
- Manages dependencies and project configurations.

Output in Android

1. **Text Output**: Using TextView or logs (Log.d, Log.e) for debugging.
2. **File Output**: Storing data in local files or SharedPreferences.
3. **GUI Output**: Interactive output through activities and fragments.
4. **CLI Output**: Accessible through logs (adb logcat) or shell commands.

User Interface in Android

Command-Line Interface (CLI):

- Used primarily for debugging (adb shell, adb logcat).
- No graphical interaction; focus on text-based operations.

Graphical User Interface (GUI):

- Built using XML layouts and integrated with activities.
- Components include:
    - **Widgets**: Buttons, TextViews, EditTexts, etc.
    - **Containers**: LinearLayout, RelativeLayout, ConstraintLayout.
    - **Navigation**: Fragments, NavGraph.

Form Integration and GUI Components

- **Form Components**:
    - **Input Fields**: EditText, Spinner, RadioGroup.
    - **Validation**: Using TextWatcher or programmatic checks.
- **Event Handling**:
    - Buttons triggering actions via onClickListeners.
    - Real-time form validation using listeners.

Event-Driven Architecture in Java (Spring)

Overview:

Event-driven architecture in Java Spring involves **events**, **publishers**, and **listeners**.

1. **Event**: Represents an occurrence in the system.
2. **Publisher**: Triggers the event using ApplicationEventPublisher.
3. **Listener**: Listens for and reacts to the event using @EventListener.

Adapters:

- **Role**: Enable compatibility between interfaces, making system components loosely coupled.

Event-Driven Architecture in Android

1. **Listeners**:
    - **Click Listeners**: View.OnClickListener for UI interactions.
    - **Lifecycle Callbacks**: onResume(), onPause() for activity states.
2. **Adapters**:

- Custom adapters for connecting data sources to UI components (e.g., RecyclerView.Adapter).

Android Java: Start from A to Z

1. **Environment Setup**: Install Android Studio, configure SDK.
2. **Create a Project**: Choose template (Empty Activity).
3. **Understand Project Structure**: Files like Manifest, MainActivity.
4. **Design Layout**: Use XML to design UI.
5. **Implement Logic**: Write Java/Kotlin code for interactions.
6. **Run and Debug**: Use emulator or device for testing.
7. **Advanced Concepts**: Explore Fragments, Services, Content Providers.
8. **Deployment**: Generate APK and upload to Play Store.

1. Android Project Structure and Build Tool (Gradle)

Default Project Files Explained

1. **AndroidManifest.xml**:
   - Declares essential information about the app (permissions, activities, services, etc.).

   Example snippet:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.myapp">
  <application
    android:allowBackup="true"
    android:label="@string/app_name"
    android:theme="@style/Theme.MyApp">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

2. **build.gradle (Module level)**:
   - Contains app-level configurations.

   Example:

```
android {
  compileSdkVersion 33
  defaultConfig {
      applicationId "com.example.myapp"
      minSdkVersion 21
      targetSdkVersion 33
  }
  buildTypes {
      release {
          minifyEnabled false
      }
  }
}
dependencies {
  implementation "androidx.core:core-ktx:1.10.1"
  implementation "androidx.appcompat:appcompat:1.6.1"
}
```

3. **res/ Folder**:

layout/: UI design in XML.
Example: activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!" />
</LinearLayout>
```

- **values/**: Contains reusable constants (strings, colors, dimensions).

---

2. Output in Android

1. **Log Output**:
   - Useful for debugging.

   Example:
   ```
   Log.d("MainActivity", "App started successfully");
   ```

2. **File Output**:
   - Writing data to internal storage:

   ```
   String filename = "myfile.txt";
   String content = "Hello File!";
   FileOutputStream fos = openFileOutput(filename, Context.MODE_PRIVATE);
   fos.write(content.getBytes());
   fos.close();
   ```

3. **UI Output**:

Example using Toast:

```
Toast.makeText(this, "Welcome to Android!", Toast.LENGTH_SHORT).show();
```

---

3. User Interface (GUI) in Android

Building a GUI

1. **Layouts**:
   - LinearLayout, RelativeLayout, ConstraintLayout, FrameLayout.

   Example: A form using ConstraintLayout.

```
<EditText
   android:id="@+id/username"
   android:hint="Username"
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintEnd_toEndOf="parent" />
<Button
   android:id="@+id/submitButton"
   android:text="Submit"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   app:layout_constraintTop_toBottomOf="@id/username" />
```

2. **Event Handling**:

   Adding onClickListener for buttons:

```
Button button = findViewById(R.id.submitButton);
button.setOnClickListener(view -> {
   Toast.makeText(this, "Button Clicked!", Toast.LENGTH_SHORT).show();
});
```

---

4. Event-Driven Architecture in Java Spring

1. **Publisher Example**:

```
@Component
public class MyEventPublisher {
   @Autowired
   private ApplicationEventPublisher publisher;
   public void publishEvent(String message) {
      publisher.publishEvent(new MyCustomEvent(this, message));
   }
}
```

2. **Listener Example**:

```
@Component
public class MyEventListener {
    @EventListener
    public void handleEvent(MyCustomEvent event) {
        System.out.println("Received event: " + event.getMessage());
    }
}
```

---

5. Event-Driven Architecture in Android

1. **UI Listeners**:

   Example: Listening to text changes in EditText.

```
EditText editText = findViewById(R.id.editText);
editText.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after) { }
    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) { }
    @Override
    public void afterTextChanged(Editable s) {
        Log.d("EditText", "Text changed to: " + s.toString());
    }
});
```

2. **Broadcast Receivers**:
   - Listening for system-wide events like network changes.

```
BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("BroadcastReceiver", "Network status changed");
    }
};
IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
registerReceiver(receiver, filter);
```

---

6. Android Java: End-to-End Steps

Step-by-Step Guide:

1. **Set Up Android Studio**: Install and configure Android SDK.
2. **Create a New Project**: Choose a project template like "Empty Activity".
3. **Understand Components**:
   - **Activities**: Entry points for user interaction.
   - **Fragments**: Reusable UI parts.
   - **Services**: Background tasks.
4. **Design the UI**:
   - Use XML for layouts, or programmatically define views.
5. **Write Business Logic**:

    - Handle user interactions and perform tasks like network requests.
6. **Debug and Test**:
    - Use Logcat, debug tools, and emulator.
7. **Publish the App**:
    - Generate APK or AAB, and upload to Google Play.

User Interface Design in Android

---

1. Views and ViewGroups

Views

- **Definition**: A View is the basic building block of Android's UI, representing a rectangular area on the screen.
- **Examples**:
    - TextView (displays text)
    - Button (clickable button)
    - ImageView (displays images)
    - EditText (input field for text)
- **Key Properties**:
    - id: Unique identifier for referencing the view.
    - layout_width and layout_height: Define the size (e.g., match_parent, wrap_content).
    - visibility: Can be visible, invisible, or gone.
    - padding and margin: For spacing inside or around the view.

ViewGroups

- **Definition**: A ViewGroup is a container that holds and arranges multiple View objects (and possibly other ViewGroup objects).
- **Examples**:
    - LinearLayout
    - RelativeLayout
    - ConstraintLayout
    - FrameLayout
- **Purpose**:
    - To structure and organize views.
    - To define layout behavior (e.g., stacking, aligning, or constraining views).
- **Common Methods**:
    - addView(View view): Adds a child view programmatically.
    - removeView(View view): Removes a child view.

Layouts

Android layouts define the structure and appearance of the user interface for an application. Here are the key points to understand about Android layouts:

---

1. Layout Types

Android provides several types of layouts to organize UI components:

- **LinearLayout**:
  - Aligns children in a single row (horizontal) or column (vertical).
  - Commonly uses android:orientation to set alignment.
  - Can include weights (layout_weight) for proportional spacing.
- **RelativeLayout**:
  - Positions child elements relative to each other or the parent container.
  - Uses attributes like layout_alignParentStart, layout_below, etc.
- **ConstraintLayout**:
  - Flexible and powerful layout allowing UI positioning with constraints.
  - Reduces nesting and improves performance.
  - Uses constraints for alignment (layout_constraintStart_toStartOf, etc.).
- **FrameLayout**:
  - Simple container for holding a single child view.
  - Useful for stacking views.
- **TableLayout**:
  - Organizes child views into rows and columns.
  - Uses TableRow to define each row.
- **GridLayout**:
  - Organizes content into a grid.
  - Uses rows and columns with flexible control over cell size.

---

2. Attributes

- **Width and Height**:
  - match_parent: The view fills the parent container.
  - wrap_content: The view adjusts to its content.
  - Fixed dimensions: Specify exact size (e.g., 200dp).
- **Padding and Margin**:
  - Padding: Space inside the view, between its border and content.
  - Margin: Space outside the view, between it and other views.
- **Gravity and Layout Gravity**:
  - gravity: Aligns content within a view.
  - layout_gravity: Aligns the view within its parent.

---

3. Nested Layouts

- Avoid excessive nesting as it affects performance.
- Use ConstraintLayout or flatten hierarchies for optimization.

---

4. XML Layout Files

- Define layouts in XML files stored in the res/layout folder.
- Example:

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  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>
```

---

5. ViewGroups

- Layouts are subclasses of ViewGroup, which manage how child views are displayed.
- Common ViewGroups:
  - ScrollView: Enables scrolling.
  - RecyclerView: Efficiently displays scrollable lists.

Intents

---

1. Intents in Android

Intents are a messaging mechanism in Android that facilitates communication between components like activities, services, and broadcast receivers.

Types of Intents

1. **Explicit Intent**:

  - Used to start a specific component (e.g., starting an activity or service).

    Example:

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

2. **Implicit Intent**:
   - Used when the target component is not explicitly specified.
   - The system matches the intent with suitable components using intent filters.

   Example:

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

Key Components of Intents

- **Action**: Specifies the action to perform (e.g., Intent.ACTION_VIEW).
- **Data**: Specifies the data to operate on (e.g., URI, MIME type).
- **Extras**: Passes additional data using a key-value pair.

  Example:

  ```
  intent.putExtra("key", "value");
  ```

- **Categories**: Provides additional information about the component.

Common Use Cases

- Start a new activity: startActivity(intent)
- Start a service: startService(intent)
- Send broadcast messages: sendBroadcast(intent)

Implementing Splash Screen in Android

Below are two methods to create a splash screen in Android: using a **ConstraintLayout** and using **MotionLayout**.

---

1. Splash Screen Using ConstraintLayout

This approach creates a static splash screen with a background and logo.

Steps

1. **Create a Drawable for the Background (Optional)**:
   - Add a drawable or color for the background in res/drawable or res/values/colors.xml.

   Example:

   ```
   <!-- res/values/colors.xml -->
   <color name="splash_background">#FFFFFF</color>
   ```

2. **Create the Layout File**:
   - Use ConstraintLayout for the splash screen.

   Example: res/layout/activity_splash.xml

   ```
   <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"
     android:background="@color/splash_background">

     <ImageView
       android:id="@+id/logo"
       android:layout_width="200dp"
       android:layout_height="200dp"
       android:src="@drawable/logo"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintTop_toTopOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintEnd_toEndOf="parent" />

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

3. **Set Up the Splash Activity**:
   - Create an activity to display the splash screen and navigate to the main screen.

   Example: SplashActivity.java

   ```
   public class SplashActivity extends AppCompatActivity {
     @Override
     protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_splash);

       new Handler().postDelayed(() -> {
         Intent intent = new Intent(SplashActivity.this, MainActivity.class);
         startActivity(intent);
         finish();
       }, 2000); // 2 seconds delay
     }
   }
   ```

4. **Set Splash Activity in Manifest**:

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

---

2. Splash Screen Using MotionLayout

MotionLayout allows animating the logo or other components during the splash screen.

Steps

1. **Add the Motion Scene File**:
   - Create a motion scene XML file in res/xml (e.g., res/xml/splash_scene.xml).

   Example:

```
<MotionScene xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:motion="http://schemas.android.com/apk/res-auto">

  <Transition
    motion:constraintSetStart="@id/start"
    motion:constraintSetEnd="@id/end"
    motion:duration="2000">

    <OnSwipe
      motion:dragDirection="dragDown"
      motion:touchAnchorId="@id/logo"
      motion:touchAnchorSide="top" />

  </Transition>

  <ConstraintSet android:id="@+id/start">
    <Constraint
      android:id="@id/logo"
      android:layout_width="200dp"
      android:layout_height="200dp"
      app:layout_constraintTop_toTopOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintEnd_toEndOf="parent" />
  </ConstraintSet>

  <ConstraintSet android:id="@+id/end">
    <Constraint
      android:id="@id/logo"
      android:layout_width="100dp"
      android:layout_height="100dp"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintEnd_toEndOf="parent" />
  </ConstraintSet>
</MotionScene>
```

2. **Create the Layout File**:
   - Use MotionLayout for the splash screen layout.

   Example: res/layout/activity_splash.xml

```
<androidx.constraintlayout.motion.widget.MotionLayout
  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"
  app:layoutDescription="@xml/splash_scene"
  android:background="@color/splash_background">

  <ImageView
    android:id="@+id/logo"
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:src="@drawable/logo" />

</androidx.constraintlayout.motion.widget.MotionLayout>
```

3. **Set Up the Splash Activity**:
   - Similar to the ConstraintLayout method, delay navigation to the main activity after the animation.

   Example:

```
public class SplashActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_splash);

    new Handler().postDelayed(() -> {
      Intent intent = new Intent(SplashActivity.this, MainActivity.class);
      startActivity(intent);
      finish();
    }, 2500); // Match the animation duration
  }
}
```

4. **Set Splash Activity in Manifest**:
   - Same as in the ConstraintLayout method.

Key Differences

ConstraintLayout	MotionLayout
Static layout	Supports animations and transitions
Easier to implement	Slightly more complex setup
No motion effects	Provides smooth animations

Logs in Android:

1. Verbose (Log.v)

- **Purpose:** Used for detailed and frequent logging during development.
- **Usage:** Logs that provide detailed information that is generally not needed in a production environment.
- **Example:**
- Log.v("TAG", "Verbose log message");

2. Debug (Log.d)

- **Purpose:** Used for debugging purposes to log diagnostic information.
- **Usage:** Logs that help developers understand the application's state while debugging.
- **Example:**
- Log.d("TAG", "Debug log message");

3. Info (Log.i)

- **Purpose:** Used for general informational messages that highlight the progress of the application.
- **Usage:** Logs that are less detailed than debug logs but useful for understanding app flow.
- **Example:**
- Log.i("TAG", "Info log message");

4. Warning (Log.w)

- **Purpose:** Used for logging potentially harmful situations.
- **Usage:** Logs that indicate non-critical issues that need attention but do not stop the app.
- **Example:**
- Log.w("TAG", "Warning log message");

5. Error (Log.e)

- **Purpose:** Used to log errors and critical issues that may cause application failure.
- **Usage:** Logs for exceptions or situations where the app cannot continue as expected.
- **Example:**
- Log.e("TAG", "Error log message");

6. Assert (Log.wtf)

- **Purpose:** Logs messages for assertions that should never happen. "WTF" stands for "What a Terrible Failure."
- **Usage:** Logs for conditions that are not expected to occur under any circumstances.
- **Example:**
- Log.wtf("TAG", "Assert log message");

Additional Information

- **Log Tags:** A tag is a short string (usually 1–23 characters) used to identify the source of a log message. It helps in filtering logs during debugging.
  - Example:
  - Log.d("MainActivity", "This is a debug log");
- **Logcat:** Android's system tool for viewing and filtering logs in real-time. It can be accessed in Android Studio using the Logcat panel.

Usage Guidelines

- Use appropriate log levels based on the message's importance.
- Avoid logging sensitive information (e.g., user data, passwords) in production.
- Use BuildConfig.DEBUG to ensure logs are included only in debug builds:
- if (BuildConfig.DEBUG) {
-     Log.d("TAG", "This log is visible only in debug builds");
- }

Android Lifecycle

Lifecycle Overview

Android Activity Lifecycle consists of several states that define the behavior and interaction of an activity with the user and system resources.

States in Lifecycle

- onCreate(): Called when the activity is created. Initialize components here.
- onStart(): Called when the activity becomes visible to the user.
- onResume(): Called when the activity starts interacting with the user.
- onPause(): Called when the activity is partially visible but not interactive.
- onStop(): Called when the activity is no longer visible.
- onDestroy(): Called before the activity is destroyed.
- onRestart(): Called when the activity is restarting after being stopped.

Example Program

This program demonstrates the Android Activity Lifecycle by logging messages at each lifecycle state. These logs help understand the sequence of state transitions.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d("Lifecycle", "onCreate called");
    }

    @Override
    protected void onStart() {
        super.onStart();
        Log.d("Lifecycle", "onStart called");
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.d("Lifecycle", "onResume called");
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.d("Lifecycle", "onPause called");
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.d("Lifecycle", "onStop called");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d("Lifecycle", "onDestroy called");
    }

    @Override
    protected void onRestart() {
        super.onRestart();
        Log.d("Lifecycle", "onRestart called");
    }
}
```

Explanation

1. onCreate(): Initializes the activity and sets up the initial configuration. It is called only once during the lifetime of the activity.

2. onStart(): The activity becomes visible to the user but is not yet interactive.

3. onResume(): The activity moves to the foreground and starts interacting with the user.

4. onPause(): Called when the activity is partially obscured, such as when a dialog appears. It is a good place to pause animations or release resources temporarily.

5. onStop(): Called when the activity is no longer visible. Use this to release resources or save data, as it indicates the user has navigated away.

6. onDestroy(): Called before the activity is destroyed. Use this to clean up resources and perform any necessary finalization.

7. onRestart(): Called when the activity transitions from the stopped state back to active. This typically occurs when the user navigates back to the activity.

Android Event Handling Programs

1. Program 1: Handling Button Clicks using XML Attributes

- Define the "onClick" attribute in the XML layout.
- Implement the corresponding method in the activity with a View parameter.

Example:

// XML Layout File: res/layout/activity_main.xml

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click Me"
    android:onClick="handleButtonClick" />
```

```
// Java Activity File: MainActivity.java
public void handleButtonClick(View view) {
    Toast.makeText(this, "Button Clicked!", Toast.LENGTH_SHORT).show();
}
```

2. Program 2: Handling Button Clicks using Anonymous Inner Classes

- Attach an OnClickListener programmatically in the activity.
- Override the onClick() method to handle the click event.

Example:

```
Button button = findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(MainActivity.this, "Button Clicked!", Toast.LENGTH_SHORT).show();
    }
});
```

3. Program 3: Handling Button Clicks using Lambda Expressions

- Simplify the OnClickListener using lambda expressions (Java 8+).
- Ensure the Android Studio supports Java 8 or higher for compatibility.

Example:

```
Button button = findViewById(R.id.button);
button.setOnClickListener(v -> Toast.makeText(MainActivity.this, "Button Clicked!",
Toast.LENGTH_SHORT).show());
```

4. Program 4: Handling Gestures using GestureDetector

- Use GestureDetector to detect gestures like swipe or fling.
- Attach an OnTouchListener to a View and pass touch events to GestureDetector.

Example:

```
  GestureDetector gestureDetector = new GestureDetector(this, new
GestureDetector.SimpleOnGestureListener() {

      @Override

      public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {

          Toast.makeText(MainActivity.this, "Swipe detected!", Toast.LENGTH_SHORT).show();

          return true;

      }

  });


  View view = findViewById(R.id.textView);

  view.setOnTouchListener((v, event)-> gestureDetector.onTouchEvent(event));
```

5. Program 5: Handling Key Events (Back Button and Volume Up)

- Override onBackPressed() to handle the back button.
- Override onKeyDown() to handle specific key events like volume buttons.

Example:

```
@Override
  public void onBackPressed() {
     Toast.makeText(this, "Back Button Pressed!", Toast.LENGTH_SHORT).show();
     super.onBackPressed();
  }

  @Override
  public boolean onKeyDown(int keyCode, KeyEvent event) {
     if (keyCode == KeyEvent.KEYCODE_VOLUME_UP) {
        Toast.makeText(this, "Volume Up Pressed!", Toast.LENGTH_SHORT).show();
        return true;
     }
     return super.onKeyDown(keyCode, event);
  }
```

6. Program 6: Handling Touch Events

- Override onTouchEvent() to detect touch interactions (ACTION_DOWN, ACTION_UP, ACTION_MOVE).

- Provide feedback using Toast messages or other UI updates.

Example:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            Toast.makeText(this, "Touch Down Detected!", Toast.LENGTH_SHORT).show();
            break;
        case MotionEvent.ACTION_UP:
            Toast.makeText(this, "Touch Up Detected!", Toast.LENGTH_SHORT).show();
            break;
        case MotionEvent.ACTION_MOVE:
            Toast.makeText(this, "Touch Move Detected!", Toast.ENGTH_SHORT).show();
            break;
    }
    return super.onTouchEvent(event);
}
```

7. Program 7: Handling Sensor Events

- Use SensorManager to register and listen to specific sensor events (e.g., accelerometer).

Example:

```
SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
  Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

  SensorEventListener sensorEventListener = new SensorEventListener() {
     @Override
     public void onSensorChanged(SensorEvent event) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];
        Toast.makeText(MainActivity.this, "Accelerometer: x=" + x + " y=" + y + " z=" + z,
Toast.LENGTH_SHORT).show();
     }

     @Override
     public void onAccuracyChanged(Sensor sensor, int accuracy) {
     }
  };
  sensorManager.registerListener(sensorEventListener, accelerometer,
SensorManager.SENSOR_DELAY_NORMAL);
```

8. Program 8: Handling Broadcast Receivers

- Define a BroadcastReceiver to listen for system or custom broadcasts.

Example:

```
BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        Toast.makeText(context, "Broadcast Received: " + action, Toast.LENGTH_SHORT).show();
    }
};

IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_LOW);
registerReceiver(receiver, filter);
```

9. Program 9: Handling Custom Events

- Create and trigger custom events using interfaces.

Example:

```
interface CustomEventListener {
    void onEventOccurred(String data);
}
class EventTrigger {
    private CustomEventListener listener;
    void setCustomEventListener(CustomEventListener listener) {
        this.listener = listener;
    }
    void triggerEvent() {
        if (listener != null) {
            listener.onEventOccurred("Custom Event Triggered!");
        }
    }
}
EventTrigger trigger = new EventTrigger();
trigger.setCustomEventListener(data -> Toast.makeText(this, data, Toast.LENGTH_SHORT).show());
trigger.triggerEvent();
```

Adapter (Default and Custom) in ListView, RecyclerView, and CardView

1. Introduction to Adapters in Android

Adapters in Android act as a bridge between UI components and data sources, converting data into UI elements. They are primarily used in **ListView, RecyclerView, and CardView**.

Types of Adapters in Android

- **Default Adapters:** Built-in adapters like ArrayAdapter, SimpleAdapter, and CursorAdapter.
- **Custom Adapters:** User-defined adapters extending BaseAdapter, RecyclerView.Adapter, or ArrayAdapter for complex layouts and data handling.

2. Adapter in ListView (Default & Custom)

Default Adapter (ArrayAdapter) for ListView

The ArrayAdapter is used to bind an array of data to a ListView.

Example of Default Adapter in ListView:

```
ListView listView;
String[] fruits = {"Apple", "Banana", "Mango", "Orange"};

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

    listView = findViewById(R.id.listView);
    ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple_list_item_1, fruits);
    listView.setAdapter(adapter);
}
```

Custom Adapter for ListView

For complex layouts, extend BaseAdapter and override necessary methods.

Example of Custom Adapter in ListView:

```
public class CustomAdapter extends BaseAdapter {
    private Context context;
    private String[] fruits;
    private LayoutInflater inflater;

    public CustomAdapter(Context context, String[] fruits) {
        this.context = context;
        this.fruits = fruits;
        this.inflater = LayoutInflater.from(context);
    }

    @Override
    public int getCount() { return fruits.length; }

    @Override
    public Object getItem(int position) { return fruits[position]; }

    @Override
    public long getItemId(int position) { return position; }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        if (convertView == null) {
            convertView = inflater.inflate(R.layout.list_item, parent, false);
        }
        TextView textView = convertView.findViewById(R.id.textView);
        textView.setText(fruits[position]);
        return convertView;
    }
}
```

Usage in Activity:

```
CustomAdapter adapter = new CustomAdapter(this, fruits);
listView.setAdapter(adapter);
```

---

3. Adapter in RecyclerView (Default & Custom)

RecyclerView is a more efficient and flexible alternative to ListView. It requires a **RecyclerView.Adapter** and a **ViewHolder**.

Default Adapter (Simple Example)

1. Add dependency in build.gradle (Module: app)

```
dependencies {
   implementation 'androidx.recyclerview:recyclerview:1.3.2'
}
```

2. XML for RecyclerView (activity_main.xml)

```
<androidx.recyclerview.widget.RecyclerView
   android:id="@+id/recyclerView"
   android:layout_width="match_parent"
   android:layout_height="match_parent"/>
```

3. **Adapter Class**

```
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {
   private Context context;
   private String[] fruits;

   public MyAdapter(Context context, String[] fruits) {
      this.context = context;
      this.fruits = fruits;
   }

   public static class ViewHolder extends RecyclerView.ViewHolder {
      TextView textView;

      public ViewHolder(View itemView) {
         super(itemView);
         textView = itemView.findViewById(R.id.textView);
      }
   }

   @Override
   public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
      View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.recycler_item, parent, false);
```

```
        return new ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        holder.textView.setText(fruits[position]);
    }

    @Override
    public int getItemCount() {
        return fruits.length;
    }
}
```

4. **Usage in Activity**

```
RecyclerView recyclerView = findViewById(R.id.recyclerView);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
recyclerView.setAdapter(new MyAdapter(this, fruits));
```

---

4. Adapter in CardView (Custom Adapter in RecyclerView)

CardView is a UI container that provides a card-like layout with rounded corners.

Steps to Implement CardView with RecyclerView Adapter

1. **Add CardView Dependency in build.gradle**

```
dependencies {
    implementation 'androidx.cardview:cardview:1.0.0'
}
```

2. **Card Layout (card_item.xml)**

```
<androidx.cardview.widget.CardView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardCornerRadius="8dp">
    <TextView
        android:id="@+id/cardText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:padding="16dp"/>
</androidx.cardview.widget.CardView>
```

3. **Custom Adapter for CardView**

```
public class CardAdapter extends RecyclerView.Adapter<CardAdapter.ViewHolder> {
    private Context context;
    private String[] items;

    public CardAdapter(Context context, String[] items) {
        this.context = context;
        this.items = items;
    }

    public static class ViewHolder extends RecyclerView.ViewHolder {
        TextView textView;

        public ViewHolder(View itemView) {
            super(itemView);
            textView = itemView.findViewById(R.id.cardText);
        }
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.card_item, parent, false);
        return new ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        holder.textView.setText(items[position]);
    }
    @Override
    public int getItemCount() {
        return items.length;
    }
}
```

4. **Usage in Activity**

```
RecyclerView recyclerView = findViewById(R.id.recyclerView);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
recyclerView.setAdapter(new CardAdapter(this, fruits));
```

---

5. Key Differences Between ListView, RecyclerView, and CardView

Feature	ListView	RecyclerView	CardView
Scrolling Efficiency	Less Efficient	Highly Efficient	Requires RecyclerView
ViewHolder Pattern	Optional	Mandatory	Used Inside RecyclerView
Performance	Slower (Rebinds Views)	Faster (Reuses Views)	UI Enhancer
Supports Card-like UI	No	No	Yes
Flexibility	Limited	Highly Customizable	Enhances RecyclerView
Animation Support	Minimal	Advanced Animations	Design-Oriented

---