Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO ANDROID DEVELOPMENT & PROGRAMMING BASICS (WEEKS 1-3)

INTRODUCTION TO ANDROID OS & MARKET TRENDS

CHAPTER 1: UNDERSTANDING ANDROID OS

1.1 What is Android?

- **Android** is an **open-source mobile operating system (OS)** developed by **Google**, primarily used for **smartphones, tablets, smart TVs, wearables, and IoT devices**.
- It is based on the **Linux kernel** and allows developers to create applications using **Java, Kotlin, and other programming languages**.
- As of today, Android powers **over 70% of the world's mobile devices**, making it the most widely used mobile OS.

1.2 Key Features of Android OS

✅ **Open-Source & Customizable** – Device manufacturers can modify the OS.

✅ **App Ecosystem** – Over **3 million apps** available on Google Play

Store.

✅ **Multi-Tasking** – Run multiple apps simultaneously.

✅ **Security & Updates** – Monthly security patches and feature updates.

✅ **Integration with Google Services** – Google Drive, Gmail, Google Maps, and more.

📌 **Example:**

- **Samsung modifies the core Android OS to create One UI, while Xiaomi customizes it as MIUI.**

CHAPTER 2: EVOLUTION OF ANDROID OS

2.1 History of Android

🔹 Android was founded by **Andy Rubin** in **2003** and later acquired by **Google in 2005**.

🔹 The first Android device, **HTC Dream (T-Mobile G1)**, was released in **2008**.

2.2 Android Versions & Key Features

Android Version	Release Year	Key Features
Android 1.0	2008	First version with basic Google apps.
Android 2.3 (Gingerbread)	2010	Improved UI and NFC support.
Android 4.0 (Ice Cream Sandwich)	2011	Redesigned UI and facial recognition unlock.
Android 5.0 (Lollipop)	2014	Material Design introduced.

Android 7.0 (Nougat)	2016	Split-screen multitasking.
Android 10	2019	System-wide Dark Mode and gesture navigation.
Android 12	2021	Material You design and enhanced privacy controls.
Android 14	2023	Battery optimizations, enhanced AI capabilities.

📌 **Example:**

- **Android 12 introduced the "Material You" design, allowing users to personalize the UI based on wallpaper colors.**

Chapter 3: Android Market Trends & Growth

3.1 Market Share of Android vs iOS

📊 **As of 2024, Android holds approximately 70% of the global mobile OS market share**, while iOS holds around 28%.

✅ **Why Android Dominates the Market?**

✓ **Affordability** – Available on a wide range of devices from budget to premium.

✓ **Diverse Manufacturer Support** – Samsung, Xiaomi, OnePlus, etc.

✓ **Customization** – Users can modify UI and use third-party app stores.

✓ **Variety of App Ecosystem** – Millions of free and paid apps.

📌 **Example:**

- **Developers prefer Android for global reach, while iOS is dominant in premium markets like the U.S. and Japan.**

3.2 Growth of the Android App Industry

🔹 **Google Play Store hosts over 3 million apps**, with thousands of new apps published daily.

🔹 **Key Industries Using Android Apps:**

Industry	Example Apps
E-Commerce	Amazon, Flipkart
Social Media	Instagram, TikTok
Health & Fitness	Fitbit, Google Fit
Finance & Banking	PayPal, Google Pay
Gaming	PUBG Mobile, Call of Duty

📌 **Example:**

- **The mobile gaming industry is projected to reach $200 billion by 2025, with Android leading the space.**

CHAPTER 4: FUTURE TRENDS IN ANDROID DEVELOPMENT

4.1 Emerging Technologies in Android

✅ **5G Integration** – Faster network speeds for seamless app experiences.

✅ **AI & Machine Learning** – Google Assistant, AI-driven personalization.

✅ **Foldable & Wearable Devices** – Apps optimized for new form factors.

✅ **Augmented Reality (AR) & Virtual Reality (VR)** – ARCore-powered apps.

✅ **Blockchain & Crypto Integration** – Secure transactions via Android apps.

📌 **Example:**

- **Google's ARCore allows developers to build AR apps like Google Lens and Pokémon Go.**

Exercise: Test Your Understanding

🔹 **What are the key features that make Android the most used mobile OS?**

🔹 **Explain the major differences between Android and iOS.**

🔹 **List five industries that are leveraging Android applications.**

🔹 **What are the latest trends shaping the future of Android development?**

🔹 **Why is Android's open-source nature beneficial for developers?**

Conclusion

✅ **Android OS is the most popular mobile operating system, with over 70% market share.**

✅ **It offers a customizable, open-source environment for developers and manufacturers.**

✅ **The Android market is constantly evolving, with trends like AI, AR/VR, and 5G shaping the future.**

✅ **Understanding Android's ecosystem is essential for mobile developers, businesses, and tech enthusiasts.**

SETTING UP ANDROID STUDIO & EMULATOR

CHAPTER 1: INTRODUCTION TO ANDROID STUDIO & EMULATOR

1.1 What Is Android Studio?

🔹 **Android Studio** is the **official integrated development environment (IDE) for Android app development**, created by Google. It provides a **powerful code editor, debugging tools, and an emulator** to test applications without requiring a physical device.

🔹 **Why Use Android Studio?**

✅ **Official Google Support** – Regular updates and seamless integration with Android SDK.

✅ **Code Assistance & Debugging** – Smart suggestions, refactoring, and built-in debugging tools.

✅ **Integrated Emulator** – Simulates different Android devices for testing.

✅ **Pre-built UI Templates** – Speeds up app development.

✅ **Supports Multiple Programming Languages** – Java, Kotlin, and C++.

🔹 **What Is an Android Emulator?**

✅ **A virtual Android device that runs on a computer, simulating real smartphones and tablets.**

✅ **Allows developers to test apps across different screen sizes, operating system versions, and hardware configurations.**

📌 **Example:**

- **A developer tests their Android app on an emulator running Android 12 without needing a physical device.**

CHAPTER 2: INSTALLING ANDROID STUDIO

2.1 System Requirements

✅ Windows

✓ OS: Windows 10 (64-bit) or later
✓ RAM: 8GB minimum (16GB recommended)
✓ Storage: 8GB of free disk space
✓ Processor: Intel i5 or AMD Ryzen (4 cores recommended)

✅ Mac

✓ OS: macOS 11+ (Big Sur or later)
✓ RAM: 8GB minimum
✓ Storage: 8GB of free disk space

✅ Linux

✓ OS: Ubuntu 18.04 or later
✓ RAM: 8GB minimum
✓ Storage: 8GB of free disk space

2.2 Downloading & Installing Android Studio

1️⃣ Download Android Studio

- Visit the official Android Studio website and download the latest version.

2️⃣ Run the Installer

- Windows: Open the .exe file and follow the setup wizard.
- Mac: Open the .dmg file and drag Android Studio into the Applications folder.
- Linux: Extract the .tar.gz file and run studio.sh.

3️⃣ Select Installation Type

- Choose **Standard** for recommended settings.
- Choose **Custom** to manually configure SDK, JDK, and UI themes.

4️⃣ Download Required SDK Components

- Ensure the **Android SDK, Emulator, and SDK Tools** are installed.

📌 **Example:**

- **A beginner developer selects the "Standard" setup option to install Android Studio with all essential tools pre-configured.**

CHAPTER 3: CONFIGURING ANDROID EMULATOR

3.1 Setting Up an Emulator in Android Studio

1️⃣ Open Android Studio

- Click **More Actions → AVD Manager** (Android Virtual Device Manager).

2️⃣ Create a New Virtual Device

- Click **Create Virtual Device** → Select a hardware profile (e.g., Pixel 6, Nexus 5X).

3️⃣ Choose a System Image

- Select an Android version (e.g., Android 13 – API Level 33).
- Click **Download** if the system image isn't installed.

4️⃣ Configure Emulator Settings

- Set RAM, resolution, and orientation based on the target device.

5️⃣ Launch the Emulator

- Click **Finish** → Select the emulator → Click **Run** to start.

📌 **Example:**

- **A developer sets up a Pixel 6 emulator running Android 12 to test their app's UI responsiveness.**

CHAPTER 4: RUNNING & DEBUGGING AN APP ON THE EMULATOR

4.1 Running an App on an Emulator

- Open an Android project in Android Studio.
- Click the **Run** button (▶) and select the emulator as the target device.
- The app will install and launch inside the emulator.

4.2 Debugging & Testing on the Emulator

✅ **Live Testing** – Interact with the app like on a real device.
✅ **Logcat Console** – View system logs and error messages.
✅ **Performance Testing** – Simulate low battery, slow network, and different CPU speeds.
✅ **Screen Rotation & Multi-Touch Gestures** – Test how the app adapts to different screen orientations.

📌 **Example:**

- **A developer uses Logcat to debug a crash when a user clicks a button in the app.**

CHAPTER 5: ADVANCED EMULATOR FEATURES

5.1 Simulating Real-World Scenarios

🔹 **GPS Simulation** – Test location-based features by setting mock GPS locations.

🔹 **Camera Simulation** – Test camera-based applications using a virtual webcam.

🔹 **Network Speed Simulation** – Simulate **3G, 4G, or slow Wi-Fi** for performance testing.

🔹 **Battery Performance** – Test how the app behaves on **low battery levels**.

📌 **Example:**

- **A ride-hailing app developer simulates different GPS locations to test real-time tracking.**

Exercise: Test Your Understanding

🔹 **What are the benefits of using Android Studio for app development?**

🔹 **How do you create an Android Emulator for testing an app?**

🔹 **What tools are available for debugging an app inside the emulator?**

🔹 **How can you simulate a slow network connection using the Android Emulator?**

🔹 **Why is testing on multiple emulator devices important?**

Conclusion

✅ **Android Studio provides a complete development environment for building Android applications efficiently.**

✅ **The Android Emulator allows developers to test apps on different devices without needing physical hardware.**

✅ **Debugging tools like Logcat, performance simulation, and**

network testing ensure a smooth development process.

✅ **Mastering emulator setup and testing is essential for professional Android developers.**

UNDERSTANDING ANDROID PROJECT STRUCTURE

CHAPTER 1: INTRODUCTION TO ANDROID PROJECT STRUCTURE

1.1 What Is an Android Project?

🔹 **An Android project is a structured collection of files that contain all the code, resources, and configurations needed to build an Android application.**

🔹 **The project structure follows a standardized hierarchy to organize source code, UI components, libraries, and assets efficiently.**

🔹 **Why Is Understanding the Android Project Structure Important?**

✅ **Efficient Code Organization – Helps developers find files easily.**

✅ **Faster Debugging – Knowing where configurations and dependencies are defined.**

✅ **Better Team Collaboration – Standardized structure allows teams to work seamlessly.**

✅ **Optimized Build Process – Helps in proper configuration of Gradle files for efficient compilation.**

📌 **Example:**

- **An Android developer working on a social media app needs to navigate the project structure to modify UI layouts and backend APIs efficiently.**

CHAPTER 2: OVERVIEW OF ANDROID PROJECT STRUCTURE

2.1 Default Android Project Hierarchy

When creating a new Android project in Android Studio, the default structure is divided into key directories and files:

📂 Project Root Directory
📁 app/ – Contains all the app-related code and resources.
📁 gradle/ – Contains Gradle wrapper files for project build management.
📂 Gradle Files – Define dependencies, build settings, and SDK versions.

🔹 Key Sections of the app/ Directory:
✔ manifests/ – Contains the AndroidManifest.xml file (application metadata).
✔ java/ – Contains Java/Kotlin source code files.
✔ res/ – Stores UI layouts, images, colors, and other resources.

📌 Example:

- **A banking app's res/layout/ directory contains XML files defining different activity UI layouts.**

CHAPTER 3: UNDERSTANDING KEY DIRECTORIES & FILES

3.1 The manifests/ Directory

📂 Location: app/src/main/AndroidManifest.xml

🔹 The AndroidManifest.xml file defines essential application metadata, including:
✅ App package name – Unique identifier of the app.
✅ Permissions – Requests necessary user permissions (e.g., internet access, camera usage).

✅ **Activities & Services – Registers app components (e.g., activities, services, broadcast receivers).**

📌 **Example: Sample AndroidManifest.xml File**

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.myapp">

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

  <application
    android:allowBackup="true"
    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>
```

📌 **Example Use Case:**

- **A weather app requests android.permission.ACCESS_FINE_LOCATION to fetch the user's location for weather updates.**

3.2 The java/ Directory

📁 **Location: app/src/main/java/com/example/myapp/**

🔹 **The java/ directory contains the main application logic written in Java or Kotlin.**

🔹 **It typically follows the MVC (Model-View-Controller) or MVVM (Model-View-ViewModel) design pattern.**

✅ **MainActivity.java / MainActivity.kt – The main entry point of the application.**

✅ **Adapters/ – Contains adapters for handling RecyclerView lists and dynamic UI elements.**

✅ **Models/ – Defines data models used in the app (e.g., User, Product).**

✅ **ViewModels/ – Manages UI-related data in MVVM architecture.**

📌 **Example:**

- **A messaging app defines a User.java model in the models/ package to store user details like name and profilePictureUrl.**

3.3 The res/ Directory

📁 **Location: app/src/main/res/**

🔹 The res/ directory contains all non-code resources, including XML files for layouts, images, colors, and themes.

✅ layout/ – Defines UI screens using XML files.
✅ drawable/ – Stores app images, icons, and vector assets.
✅ values/ – Stores reusable resources like colors, dimensions, and strings.
✅ mipmap/ – Contains different sizes of the app launcher icon.

📌 Example: Sample res/layout/activity_main.xml File

```
<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, Android!" />

</LinearLayout>
```

📌 Example Use Case:

- A news app stores category names in res/values/strings.xml for localization support.

CHAPTER 4: GRADLE BUILD SYSTEM IN ANDROID

4.1 What Is Gradle?

🔹 **Gradle is the build automation tool for Android projects, responsible for compiling, packaging, and managing dependencies.**

✅ **Handles Dependency Management – Fetches required libraries automatically.**
✅ **Builds & Signs APKs – Configures build settings for different app versions.**
✅ **Supports Multi-Module Projects – Helps in modular Android development.**

📌 **Example:**

- **An e-commerce app uses Gradle to include Retrofit for API calls and Glide for image loading.**

4.2 Key Gradle Files in an Android Project

📁 **Project-Level Gradle File (build.gradle - Project)**

- **Defines the Gradle version, repositories, and global configurations.**

📁 **Module-Level Gradle File (build.gradle - App)**

- **Specifies dependencies, SDK versions, and build flavors for the app.**

📌 **Example: Sample build.gradle (Module: app)**

```
android {
```

```
    compileSdkVersion 33

    defaultConfig {
        applicationId "com.example.myapp"
        minSdkVersion 21
        targetSdkVersion 33
        versionCode 1
        versionName "1.0"
    }

    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation 'androidx.appcompat:appcompat:1.4.0'
    implementation 'com.google.android.material:material:1.6.0'
}
```

📌 **Example Use Case:**

- **A fintech app enables ProGuard minification in Gradle to reduce APK size and obfuscate code.**

Chapter 5: Best Practices for Managing Android Project Structure

5.1 Organizing Code Efficiently

✅ **Follow the MVVM Architecture – Separate UI logic, business logic, and data management.**
✅ **Use Proper Package Naming Conventions – Group similar functionalities in the same package.**
✅ **Keep Resource Files Well-Structured – Store drawable images in drawable/, layout XMLs in layout/, and strings in values/.**

5.2 Optimizing Gradle Build Speed

✅ **Use Dependency Caching – Enable Gradle offline mode to speed up builds.**
✅ **Enable Parallel Builds – Run multiple build tasks simultaneously.**
✅ **Use ProGuard for Code Shrinking – Remove unused code to reduce APK size.**

📌 **Example:**

- **An enterprise app enables Gradle caching to reduce build time from 30 seconds to 10 seconds.**

Exercise: Test Your Understanding

- **What is the role of the AndroidManifest.xml file in an Android project?**
- **Where are UI layouts stored in an Android project?**
- **How does Gradle help in managing dependencies?**
- **What are the best practices for organizing Android code?**

Conclusion

✅ **Understanding the Android project structure is crucial for efficient development, debugging, and scaling of applications.**

✅ **Directories like java/, res/, and manifests/ organize different components of the app.**

✅ **Gradle plays a key role in managing dependencies and building APKs.**

✅ **Following best practices ensures better maintainability and performance of Android projects.**

VARIABLES, DATA TYPES, AND OPERATORS IN PROGRAMMING

CHAPTER 1: UNDERSTANDING VARIABLES IN PROGRAMMING

1.1 What is a Variable?

A **variable** is a **named storage location in memory** used to hold a value that can change during the execution of a program. It acts as a container for data.

🔹 **Why Are Variables Important?**

✅ **Store values dynamically** instead of hardcoding them.

✅ **Enhance code reusability** by allowing modifications.

✅ **Help perform calculations and data manipulations.**

🔹 **Variable Naming Rules:**

✓ Must begin with a **letter or underscore (_) but not a number**.

✓ Can only contain **letters, numbers, and underscores (a-z, A-Z, 0-9, _)**.

✓ **Case-sensitive** (e.g., age and Age are different).

✓ Should be **descriptive** (user_age instead of x).

📌 **Example (Python):**

```
name = "Alice"  # Assigning a string value to a variable
age = 25        # Assigning an integer value
price = 9.99    # Assigning a floating-point number
```

📌 **Example (JavaScript):**

```
let city = "New York"; // Using 'let' to declare a variable
const PI = 3.14;       // Using 'const' for a constant value
```

```
var isStudent = true;  // Using 'var' (older declaration method)
```

CHAPTER 2: DATA TYPES IN PROGRAMMING

2.1 What Are Data Types?

A **data type** defines the **type of value** a variable can store, such as numbers, text, or boolean values.

- **Common Data Types in Programming:**

Data Type	Description	Example
Integer (int)	Whole numbers (positive/negative)	age = 25
Floating-Point (float, double)	Decimal numbers	price = 99.99
String (str, char)	Sequence of characters	name = "John"
Boolean (bool)	True/False values	is_active = True
List/Array	Collection of values	colors = ["red", "blue", "green"]
Dictionary/Object	Key-value pairs	person = {"name": "Alice", "age": 25}

📌 **Example (Python):**

```
num = 100        # Integer
price = 45.75    # Float
name = "John Doe"  # String
is_logged_in = True  # Boolean
```

```
fruits = ["apple", "banana", "cherry"]  # List
```

📌 **Example (JavaScript):**

```
let age = 30;         // Integer
let temperature = 98.6; // Float
let user = "Charlie";   // String
let isActive = false;   // Boolean
let numbers = [1, 2, 3, 4]; // Array
```

2.2 Type Conversion (Casting)

- **Implicit Conversion (Automatic)** – The system converts one data type to another automatically.
- **Explicit Conversion (Casting)** – The programmer manually converts a data type.

📌 **Example (Python):**

```
# Implicit Conversion
x = 10    # Integer
y = 5.5   # Float
result = x + y  # Integer is automatically converted to Float
print(result)  # Output: 15.5

# Explicit Conversion
num = "100"      # String
converted_num = int(num)  # Convert string to integer
```

```
print(converted_num + 50)  # Output: 150
```

📌 **Example (JavaScript):**

```
// Implicit Conversion
let x = "5" + 2;  // JavaScript converts 2 to a string: "52"

// Explicit Conversion
let num = "200";
let convertedNum = Number(num); // Convert string to number
console.log(convertedNum + 50); // Output: 250
```

CHAPTER 3: OPERATORS IN PROGRAMMING

3.1 What Are Operators?

Operators are **symbols** that perform operations on variables and values, such as addition, subtraction, comparison, or logical operations.

- **Types of Operators:**

1. **Arithmetic Operators** – Perform mathematical operations.
2. **Comparison Operators** – Compare two values.
3. **Logical Operators** – Combine multiple conditions.
4. **Assignment Operators** – Assign values to variables.
5. **Bitwise Operators** – Operate on bits (advanced).

3.2 Arithmetic Operators

These operators perform mathematical operations like **addition, subtraction, multiplication, and division**.

Operator	Description	Example (Python)	Example (JavaScript)
+	Addition	x = 5 + 2 ⟶ 7	x = 5 + 2; ⟶ 7
-	Subtraction	x = 10 - 4 ⟶ 6	x = 10 - 4; ⟶ 6
*	Multiplication	x = 3 * 4 ⟶ 12	x = 3 * 4; ⟶ 12
/	Division	x = 15 / 3 ⟶ 5.0	x = 15 / 3; ⟶ 5
%	Modulus (Remainder)	x = 10 % 3 ⟶ 1	x = 10 % 3; ⟶ 1
**	Exponentiation (Power)	x = 2 ** 3 ⟶ 8	x = 2 ** 3; ⟶ 8

📌 **Example:**

```
a = 10

b = 3

print(a + b)  # Output: 13

print(a % b)  # Output: 1 (Remainder)

let a = 10;

let b = 3;

console.log(a * b);  // Output: 30

console.log(a ** b); // Output: 1000 (Exponentiation)
```

3.3 Comparison Operators

These operators compare two values and return **True or False**.

Operator	Description	Example (Python & JavaScript)
==	Equal to	5 == 5 → True
!=	Not equal to	10 != 5 → True
>	Greater than	7 > 3 → True
<	Less than	4 < 8 → True
>=	Greater than or equal to	10 >= 10 → True
<=	Less than or equal to	6 <= 9 → True

3.4 Logical Operators

These operators evaluate multiple conditions.

Operator	Description	Example
and / &&	Returns True if both conditions are true	(x > 5 and x < 10)
or / `		`
not / !	Reverses the condition result	not(x > 5)

📌 **Example:**

```
x = 7
print(x > 5 and x < 10)  # Output: True
print(not(x > 5))        # Output: False
let x = 7;
console.log(x > 5 && x < 10);  // Output: true
```

```
console.log(!(x > 5));       // Output: false
```

Exercise: Test Your Understanding

- **What is the difference between == and = in programming?**
- **How do you declare a variable in Python and JavaScript?**
- **What is the output of 10 % 4 in Python?**
- **Explain the difference between and, or, and not operators.**
- **Convert "500" (string) into an integer in Python.**

Conclusion

✅ **Variables store values that can change during program execution.**

✅ **Data types define the kind of values a variable can hold.**

✅ **Operators perform mathematical, comparison, and logical operations.**

Control Flow in Android Development (Loops, If-Else, Switch)

Chapter 1: Introduction to Control Flow in Programming

1.1 What is Control Flow?

🔹 **Control flow** refers to the **order in which individual statements, instructions, or function calls are executed or evaluated** in a program.

🔹 It **determines decision-making and repetition** in code execution, ensuring logical flow.

🔹 In **Android development**, control flow helps manage **user inputs, UI behaviors, and app logic dynamically**.

🔹 **Key Types of Control Flow:**

✅ **Decision-Making Statements** – if-else, switch-case (for conditional execution).

✅ **Loops** – for, while, do-while (for repetitive tasks).

📌 **Example:**

- **An Android app decides whether to show a dark or light theme based on user preferences using an if-else condition.**

Chapter 2: Decision-Making Statements (If-Else, Switch)

2.1 If-Else Statements

🔹 The if-else statement is used for **executing code conditionally**.

🔹 It allows the program to check whether a condition is **true or false** and execute the corresponding block of code.

Syntax of If-Else in Kotlin (Android Development)

```
val isDarkMode = true


if (isDarkMode) {
   println("Dark Mode Activated")
} else {
   println("Light Mode Activated")
}
```

📌 **Example:**

- **An Android app checks if a user is logged in before showing the home screen.**

```
val isLoggedIn = true


if (isLoggedIn) {
   println("Welcome, User!")
} else {
   println("Please log in.")
}
```

2.2 Nested If-Else Statements

🔹 **Nested if-else** is used when **multiple conditions** need to be checked.

```
val score = 85
```

```
if (score >= 90) {
   println("Grade: A")
} else if (score >= 75) {
   println("Grade: B")
} else {
   println("Grade: C")
}
```

📌 **Example:**

- **A food delivery app applies different discount percentages based on order value.**

```
val orderAmount = 500

if (orderAmount > 1000) {
   println("Discount Applied: 20%")
} else if (orderAmount > 500) {
   println("Discount Applied: 10%")
} else {
   println("No Discount Available")
}
```

2.3 Switch-Case in Kotlin (When Statement in Android)

- In **Kotlin,** the when statement is used instead of switch-case.
- It is **more powerful** and allows checking multiple conditions in a concise way.

Syntax of When Statement in Kotlin

```
val day = 3


when (day) {
   1 -> println("Monday")
   2 -> println("Tuesday")
   3 -> println("Wednesday")
   4 -> println("Thursday")
   5 -> println("Friday")
   6 -> println("Saturday")
   7 -> println("Sunday")
   else -> println("Invalid Day")
}
```

📌 **Example:**

- **A mobile app shows a different greeting message based on the day of the week.**

```
val day = "Sunday"


when (day) {
```

```
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" ->
println("Have a productive day!")
    "Saturday", "Sunday" -> println("Enjoy your weekend!")
    else -> println("Invalid input")
}
```

CHAPTER 3: LOOPS (FOR, WHILE, DO-WHILE) IN ANDROID DEVELOPMENT

3.1 For Loop

- A for loop is used when the **number of iterations is known**.
- It **executes a block of code multiple times** based on a defined range or list.

Syntax of For Loop in Kotlin

```
for (i in 1..5) {
    println("Iteration: $i")
}
```

📌 **Example:**

- **An Android e-commerce app displays a list of five product names using a loop.**

```
val products = listOf("Laptop", "Phone", "Headphones", "Tablet",
"Camera")

for (product in products) {
    println("Product: $product")
```

```
}
```

3.2 While Loop

- A while loop is used when the **number of iterations is unknown** and the loop continues until a condition is false.

Syntax of While Loop in Kotlin

```
var count = 5


while (count > 0) {
   println("Countdown: $count")
   count--
}
```

📌 **Example:**

- **A mobile banking app asks for the correct PIN, allowing up to 3 attempts.**

```
var attempts = 3
var correctPin = 1234
var enteredPin = 0


while (attempts > 0 && enteredPin != correctPin) {
   println("Enter PIN:")
   enteredPin = readLine()?.toIntOrNull() ?: 0
   attempts--
```

```
}


if (enteredPin == correctPin) {
   println("Access Granted")
} else {
   println("Account Locked")
}
```

3.3 Do-While Loop

- A do-while loop **executes at least once**, even if the condition is false.

Syntax of Do-While Loop in Kotlin

```
var number = 1


do {
   println("Number: $number")
   number++
} while (number <= 5)
```

📌 **Example:**

- **An Android fitness app prompts the user to input weight every day until they decide to stop.**

```
var input: String
```

```
do {
   println("Enter your weight (or type 'exit' to stop): ")
   input = readLine() ?: "exit"
} while (input.lowercase() != "exit")
```

Chapter 4: Best Practices for Using Control Flow in Android Apps

✅ **Use If-Else Only When Necessary** – Avoid complex nested conditions, use when instead.
✅ **Optimize Loops** – Prevent infinite loops that can crash an app.
✅ **Ensure User Input Validation** – Use loops to handle repeated input validation.
✅ **Use Coroutines for Asynchronous Operations** – Avoid blocking the main thread in loops.
✅ **Keep Code Readable & Modular** – Separate logic into functions for maintainability.

📌 **Example:**

- **A ridesharing app calculates the estimated fare based on distance using when.**

```
val distance = 10


val fare = when {
   distance <= 5 -> 50
   distance <= 10 -> 100
   distance <= 20 -> 200
```

```
    else -> 300
}

println("Total Fare: $$fare")
```

Exercise: Test Your Understanding

- **Explain the difference between if-else and when statements in Kotlin.**
- **Write a for loop that prints numbers from 1 to 10.**
- **Create a while loop that keeps asking for user input until "exit" is entered.**
- **Write an Android-based example where a switch-case (when) statement selects a UI theme (dark, light, system default).**
- **How does a do-while loop differ from a while loop?**

Conclusion

✅ **Control flow statements (If-Else, Switch, Loops) determine decision-making and repetitions in Android apps.**
✅ **If-Else and When are used for conditional execution, while Loops handle repeated tasks.**
✅ **Efficient control flow improves app performance, security, and user experience.**
✅ **Mastering control flow helps developers build responsive and intelligent Android applications.**

FUNCTIONS AND METHODS IN PROGRAMMING

CHAPTER 1: INTRODUCTION TO FUNCTIONS AND METHODS

1.1 What Are Functions and Methods?

🔹 **Functions and methods** are reusable blocks of code that perform specific tasks. They help **modularize code, reduce redundancy, and improve code readability**.

🔹 Functions exist **independently**, while methods are functions **associated with objects** (in object-oriented programming).

🔹 **Why Use Functions & Methods?**

✅ **Code Reusability** – Write once, use multiple times.

✅ **Improves Readability** – Organizes code into logical sections.

✅ **Reduces Errors** – Debugging is easier in smaller code blocks.

✅ **Enhances Maintainability** – Modular design simplifies updates.

📌 **Example:**

- **A function calculates the area of a rectangle by taking length and width as input.**

```
def area_of_rectangle(length, width):
    return length * width

print(area_of_rectangle(5, 3))  # Output: 15
```

CHAPTER 2: FUNCTIONS IN PROGRAMMING

2.1 Defining and Calling Functions

🔹 A **function** is a **named block of reusable code** that executes when called.

🔹 A function **takes input (parameters), processes data, and returns a result**.

Syntax for Defining a Function

```
def function_name(parameters):
    # Function body
    return result  # Optional
```

📌 **Example:**

```
def greet(name):
    return f"Hello, {name}!"
print(greet("Alice"))  # Output: Hello, Alice!
```

2.2 Types of Functions

✅ **Built-in Functions** – Predefined in programming languages (e.g., print(), len(), max()).

✅ **User-Defined Functions** – Created by developers for specific tasks.

✅ **Recursive Functions** – A function that calls itself.

✅ **Lambda Functions** – Anonymous functions (useful for short operations).

📌 **Example of a Lambda Function:**

```
square = lambda x: x * x
print(square(4))  # Output: 16
```

CHAPTER 3: FUNCTION PARAMETERS AND RETURN VALUES

3.1 Function Parameters

🔹 Functions can accept **arguments** (values passed during function calls).

✅ **Positional Arguments** – Passed in order.
✅ **Default Arguments** – Uses a default value if not provided.
✅ **Keyword Arguments** – Passed using parameter names.
✅ **Variable-Length Arguments (*args, **kwargs)** – Accepts multiple values.

📌 **Example of Default & Keyword Arguments:**

```
def introduce(name, age=25):
    return f"My name is {name} and I am {age} years old."

print(introduce("John"))  # Uses default age

print(introduce(age=30, name="Emma"))  # Uses keyword arguments
```

3.2 Returning Values from Functions

🔹 Functions can **return values** using the return statement.

📌 **Example:**

```
def add(a, b):
    return a + b

result = add(5, 10)

print(result)  # Output: 15
```

CHAPTER 4: METHODS IN OBJECT-ORIENTED PROGRAMMING (OOP)

4.1 What Are Methods?

- A **method** is a **function that belongs to an object or a class**.
- Methods in **object-oriented programming (OOP)** act on **class instances** and can modify object properties.

Syntax for Defining a Method

```
class ClassName:
    def method_name(self, parameters):
        # Method body
```

📌 **Example of a Class Method:**

```
class Person:
    def __init__(self, name):
        self.name = name  # Instance attribute

    def greet(self):
        return f"Hello, my name is {self.name}."

person1 = Person("Alice")
print(person1.greet())  # Output: Hello, my name is Alice.
```

4.2 Types of Methods in OOP

✅ **Instance Methods** – Operate on instance attributes (use self).
✅ **Class Methods (@classmethod)** – Operate on class variables (use cls).

✅ **Static Methods (@staticmethod)** – Independent of class attributes.

📌 **Example of a Class & Static Method:**

```
class MathOperations:
    @classmethod
    def class_method(cls):
        return "This is a class method."

    @staticmethod
    def static_method():
        return "This is a static method."

print(MathOperations.class_method())  # Output: This is a class method.
print(MathOperations.static_method())  # Output: This is a static method.
```

CHAPTER 5: SCOPE & LIFETIME OF VARIABLES IN FUNCTIONS

5.1 Local vs. Global Variables

- **Local Variables** – Defined inside a function, only accessible within that function.
- **Global Variables** – Defined outside functions, accessible throughout the program.

📌 **Example:**

```
x = 10  # Global variable


def modify_variable():
    x = 5  # Local variable
    return x


print(modify_variable())  # Output: 5
print(x)  # Output: 10 (Global x is unchanged)
```

CHAPTER 6: RECURSION IN FUNCTIONS

6.1 What Is Recursion?

🔹 **Recursion** is when a function **calls itself** until a base condition is met.

📌 **Example: Factorial Using Recursion**

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)


print(factorial(5))  # Output: 120
```

🔹 **Recursion vs. Iteration**

✅ Recursion simplifies complex problems (e.g., tree traversal).
✅ Iteration is **memory-efficient** but requires explicit looping.

CHAPTER 7: FUNCTIONS VS. METHODS - KEY DIFFERENCES

Feature	Functions	Methods
Belongs To	Independent (standalone)	Part of a class
Access To Data	Uses local/global variables	Accesses instance/class attributes
Call Syntax	function_name()	object.method_name()
Use Case	General-purpose computations	Object-specific behaviors

📌 **Example:**

- **A function calculates a square, while a method modifies an object's attribute.**

```
def square(x):  # Function
    return x * x


class Number:
    def __init__(self, value):
        self.value = value

    def square(self):  # Method
        return self.value * self.value

num = Number(4)
```

```
print(square(4))  # Using function → Output: 16
print(num.square())  # Using method → Output: 16
```

Exercise: Test Your Understanding

- **What are the advantages of using functions in programming?**
- **How do you define and call a function in Python?**
- **What is the difference between a function and a method?**
- **What is the purpose of self in class methods?**
- **How does recursion work, and when should it be used?**

Conclusion

✅ **Functions are reusable blocks of code that improve modularity and readability.**
✅ **Methods belong to classes in object-oriented programming and operate on objects.**
✅ **Functions can have parameters, return values, and different scopes.**
✅ **Using recursion, lambda functions, and class methods enhances programming efficiency.**

CLASSES, OBJECTS, AND CONSTRUCTORS IN JAVA (ANDROID DEVELOPMENT)

CHAPTER 1: INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP) IN JAVA

1.1 What Is Object-Oriented Programming?

🔹 **Object-Oriented Programming (OOP)** is a programming paradigm that organizes code using **objects** and **classes** to promote modularity, reusability, and scalability.

🔹 Java, the primary language for Android development, follows **OOP principles** to structure applications efficiently.

🔹 **Why Use OOP in Android Development?**

✅ **Encapsulation** – Protects data using private variables and public methods.

✅ **Reusability** – Code can be reused across different parts of the application.

✅ **Scalability** – Large apps can be built in a structured manner.

✅ **Modularity** – Classes can be separated into independent components.

🔹 **Key OOP Concepts:**

✓ **Classes** – Blueprints for creating objects.

✓ **Objects** – Instances of a class with specific properties and behaviors.

✓ **Constructors** – Special methods to initialize objects.

📌 **Example:**

- **A Car class defines properties like color and speed, while different car objects (e.g., car1, car2) have unique values.**

CHAPTER 2: UNDERSTANDING CLASSES IN JAVA

2.1 What Is a Class?

🔹 A **class** is a blueprint or template used to create objects. It defines **fields (variables)** and **methods (functions)**.

🔹 In Java, classes are created using the class keyword.

📌 Example: Creating a Simple Class

```
public class Car {
  // Fields (Variables)
  String color;
  int speed;

  // Methods (Functions)
  void drive() {
    System.out.println("The car is driving.");
  }
}
```

✅ Key Features of a Class:

✓ Defines properties (e.g., color, speed).
✓ Contains methods (e.g., drive()).
✓ Can be instantiated to create multiple objects.

📌 Example Use Case:

- **A User class in an Android app stores user details like name, email, and phoneNumber.**

CHAPTER 3: UNDERSTANDING OBJECTS IN JAVA

3.1 What Is an Object?

🔹 An **object** is an instance of a class. It holds **specific values** for the class properties.

📌 Example: Creating an Object from a Class

```
public class Main {
  public static void main(String[] args) {
    Car myCar = new Car(); // Creating an object
    myCar.color = "Red";
    myCar.speed = 120;

    System.out.println("Car color: " + myCar.color);
    System.out.println("Car speed: " + myCar.speed);
    myCar.drive();
  }
}
```

✅ Output:

```
Car color: Red
Car speed: 120
The car is driving.
```

✅ Key Features of an Object:

✓ Stores unique values for class properties.

✓ Can call methods defined in the class.

✓ Multiple objects can be created from the same class.

📌 **Example Use Case:**

- **In an Android RecyclerView, each item is an object of a Product class containing name, price, and imageURL.**

CHAPTER 4: UNDERSTANDING CONSTRUCTORS IN JAVA

4.1 What Is a Constructor?

🔹 A **constructor** is a special method used to initialize objects automatically when they are created.

🔹 It has the **same name as the class** and **does not have a return type**.

📌 **Example: Default Constructor**

```
public class Car {
  String color;
  int speed;

  // Constructor
  public Car() {
    System.out.println("Car object is created.");
  }
}

public class Main {
```

```
    public static void main(String[] args) {
        Car myCar = new Car(); // Constructor is automatically called
    }
}
```

✅ **Output:**

Car object is created.

4.2 Parameterized Constructor

- A **parameterized constructor** allows passing values while creating an object.
- Helps in initializing object properties during instantiation.

📌 **Example: Using a Parameterized Constructor**

```
public class Car {
    String color;
    int speed;

    // Parameterized Constructor
    public Car(String c, int s) {
        color = c;
        speed = s;
    }

    void displayCar() {
```

```
        System.out.println("Car color: " + color);
        System.out.println("Car speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Blue", 150);
        myCar.displayCar();
    }
}
```

✅ **Output:**

```
Car color: Blue
Car speed: 150
```

📌 **Example Use Case:**

- **An Employee class with a parameterized constructor initializes name, designation, and salary upon object creation.**

4.3 Constructor Overloading

🔹 **Constructor overloading** allows defining multiple constructors with different parameters.

🔹 Java determines which constructor to use based on the **number and type of arguments**.

📌 Example: Overloaded Constructors

```
public class Car {
  String color;
  int speed;

  // Default Constructor
  public Car() {
    color = "Black";
    speed = 100;
  }

  // Parameterized Constructor
  public Car(String c, int s) {
    color = c;
    speed = s;
  }

  void displayCar() {
    System.out.println("Car color: " + color);
    System.out.println("Car speed: " + speed);
  }
}
```

```
public class Main {
   public static void main(String[] args) {
      Car car1 = new Car(); // Calls default constructor
      car1.displayCar();

      Car car2 = new Car("White", 180); // Calls parameterized constructor
      car2.displayCar();
   }
}
```

✅ **Output:**

```
Car color: Black
Car speed: 100

Car color: White
Car speed: 180
```

📌 **Example Use Case:**

- **An Order class can have multiple constructors to allow order creation with default values or user-defined inputs.**

Chapter 5: Best Practices for Classes, Objects, and Constructors

5.1 Best Practices for Classes

✅ Use **camel case** for class names (UserProfile).
✅ Follow **Single Responsibility Principle (SRP)** – Each class should have **one job**.
✅ **Use encapsulation** – Keep variables private and provide getter and setter methods.

5.2 Best Practices for Objects

✅ Always **initialize objects properly** before using them.
✅ Avoid **creating unnecessary objects** to save memory.
✅ Use **static methods** for utility functions that do not require object creation.

5.3 Best Practices for Constructors

✅ Use **constructors for mandatory properties** and setters for optional values.
✅ Avoid **long constructor parameter lists** (use builder pattern for complex initialization).
✅ Use **constructor overloading** for flexibility.

📌 **Example:**

- **An Account class should enforce the presence of accountNumber and ownerName using a constructor, while balance can be optional.**

Exercise: Test Your Understanding

- What is the difference between a class and an object?
- How does a constructor help in object initialization?
- What is constructor overloading, and how is it useful?
- How can encapsulation improve the security of class properties?
- Why should we use private variables and public getter/setter methods?

Conclusion

✅ Classes provide a blueprint for creating objects in Java.
✅ Objects are instances of a class, containing specific values for attributes.
✅ Constructors initialize objects automatically, ensuring proper setup.
✅ Using best practices in class design leads to scalable and maintainable Android applications.

Inheritance, Polymorphism, and Encapsulation in Object-Oriented Programming (OOP)

Chapter 1: Introduction to Object-Oriented Programming (OOP)

1.1 What is Object-Oriented Programming (OOP)?

🔹 **Object-Oriented Programming (OOP)** is a programming paradigm based on **objects and classes**. It helps in designing scalable, reusable, and maintainable software.

🔹 **Key Principles of OOP:**

✅ **Encapsulation** – Hiding the details of how an object works.

✅ **Inheritance** – Creating new classes from existing ones.

✅ **Polymorphism** – Using a single interface for different data types.

✅ **Abstraction** – Hiding implementation details and exposing only necessary functionalities.

📌 **Example:**

- **A Car is an object with properties (color, speed) and methods (drive, brake). Different car models can inherit common properties.**

Chapter 2: Inheritance in OOP

2.1 What is Inheritance?

🔹 **Inheritance** allows a class (child/subclass) to **inherit properties and methods** from another class (parent/superclass).

🔹 It enables **code reuse** and **hierarchical relationships** between classes.

✅ **Key Benefits of Inheritance:**

✓ Promotes **code reusability** – Common properties/methods do not need to be rewritten.

✓ Establishes a **relationship between classes** – Child classes inherit behavior from parent classes.

✓ Supports **extensibility** – New features can be added without modifying existing code.

2.2 Types of Inheritance

Type	Description	Example
Single Inheritance	One class inherits from another.	Car → ElectricCar
Multiple Inheritance	A class inherits from multiple parent classes.	FlyingCar inherits from both Car and Airplane
Multilevel Inheritance	A subclass inherits from another subclass.	Vehicle → Car → ElectricCar
Hierarchical Inheritance	Multiple subclasses inherit from a single parent class.	Animal → Dog, Cat
Hybrid Inheritance	A combination of multiple inheritance types.	A mix of hierarchical and multiple inheritance

2.3 Inheritance in Python

```
# Parent class
class Animal:
    def speak(self):
        print("Animal makes a sound")

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):
        print("Dog barks")

# Using the inherited method
pet = Dog()
pet.speak()  # Output: Dog barks
```

2.4 Inheritance in Java

```
// Parent class
class Animal {
    void speak() {
        System.out.println("Animal makes a sound");
    }
```

```
}


// Child class inheriting from Animal
class Dog extends Animal {
    void speak() {
        System.out.println("Dog barks");
    }
}


public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.speak();  // Output: Dog barks
    }
}
```

📌 **Example:**

- **A "Bird" class can inherit from an "Animal" class and define its unique behaviors like "fly".**

Chapter 3: Polymorphism in OOP

3.1 What is Polymorphism?

🔹 **Polymorphism** means **"many forms"** – it allows different classes to be treated as instances of the same class through **a common**

interface.

- A method can behave **differently based on the object** that calls it.

✅ **Benefits of Polymorphism:**

✓ Allows **code flexibility** and **extensibility**.

✓ Helps in **reducing redundancy**.

✓ Enables **dynamic method execution**.

3.2 Types of Polymorphism

Type	Description	Example
Method Overriding (Runtime Polymorphism)	A subclass redefines a method from its parent class.	Dog.speak() overrides Animal.speak()
Method Overloading (Compile-time Polymorphism)	Multiple methods with the same name but different parameters.	add(int, int) and add(float, float)

3.3 Polymorphism in Python (Method Overriding)

```
class Animal:
    def speak(self):
        print("Animal makes a sound")


class Cat(Animal):
    def speak(self):
```

```
    print("Cat meows")


# Polymorphism in action
animals = [Animal(), Cat()]
for animal in animals:
  animal.speak()
```

- **Output:**

```
Animal makes a sound
Cat meows
```

3.4 Polymorphism in Java (Method Overloading & Overriding)

```
class MathOperations {
  // Method Overloading: Same method, different parameters
  int add(int a, int b) {
    return a + b;
  }

  double add(double a, double b) {
    return a + b;
  }
}
```

```
public class Main {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();
        System.out.println(obj.add(5, 3));      // Output: 8
        System.out.println(obj.add(5.5, 2.5));   // Output: 8.0
    }
}
```

📌 **Example:**

- **A "Shape" class has a method draw(). A "Circle" and a "Rectangle" class can override it to provide different implementations.**

CHAPTER 4: ENCAPSULATION IN OOP

4.1 What is Encapsulation?

🔹 **Encapsulation** means **hiding the internal implementation details** of a class and exposing only the necessary functionalities.

🔹 This is achieved using **private variables and public methods**.

✅ **Benefits of Encapsulation:**

✓ Protects **data from accidental modification**.

✓ Provides **controlled access** using getter and setter methods.

✓ Increases **code maintainability and security**.

4.2 Encapsulation in Python

```
class BankAccount:
```

```
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance  # Getter method

# Creating an object
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

- **Here, __balance is private and cannot be accessed directly.**

4.3 Encapsulation in Java

```
class BankAccount {
    private double balance; // Private variable

    public BankAccount(double balance) {
        this.balance = balance;
    }
```

```
    public void deposit(double amount) {
        balance += amount;
    }

    public double getBalance() { // Getter method
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount(1000);
        myAccount.deposit(500);
        System.out.println(myAccount.getBalance()); // Output: 1500
    }
}
```

📌 **Example:**

- **A "User" class with private attributes (password, email) ensures sensitive data is protected using getters and setters.**

Exercise: Test Your Understanding

- What is the difference between inheritance and polymorphism?
- Why is encapsulation important in OOP?
- How does method overriding differ from method overloading?
- Explain the concept of hierarchical inheritance with an example.
- How do getter and setter methods contribute to encapsulation?

Conclusion

✅ Inheritance allows code reuse by letting child classes inherit properties and methods from parent classes.
✅ Polymorphism enables methods to take different forms, improving flexibility and scalability.
✅ Encapsulation protects data from direct modification, ensuring better security and maintainability.
✅ Mastering these OOP concepts is essential for building robust, scalable, and maintainable applications.

ASSIGNMENT:

BUILD A SIMPLE **CALCULATOR APP** USING JAVA/KOTLIN

Assignment Solution: Building a Simple Calculator App Using Java/Kotlin (Android)

Objective

The goal of this assignment is to **develop a basic calculator app** in Android using **Java or Kotlin**. The app will support **addition, subtraction, multiplication, and division** operations with a user-friendly UI.

Step 1: Setting Up the Android Project

1.1 Create a New Android Project

1. Open **Android Studio** and click **New Project**.
2. Select **Empty Activity** and click **Next**.
3. Name your project **SimpleCalculator**.
4. Choose **Java or Kotlin** as the language.
5. Click **Finish** to create the project.

Step 2: Designing the UI (activity_main.xml)

2.1 Open res/layout/activity_main.xml and add UI Components

📌 **Code for activity_main.xml (User Interface)**

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:padding="20dp">

<EditText
   android:id="@+id/etNumber1"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:hint="Enter first number"
   android:inputType="numberDecimal"/>

<EditText
   android:id="@+id/etNumber2"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:hint="Enter second number"
   android:inputType="numberDecimal"/>

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

```
    android:layout_height="wrap_content"
    android:text="Result: "
    android:textSize="20sp"
    android:textStyle="bold"
    android:gravity="center"
    android:padding="10dp"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:gravity="center">

    <Button
      android:id="@+id/btnAdd"
      android:layout_width="0dp"
      android:layout_weight="1"
      android:layout_height="wrap_content"
      android:text="+"/>

    <Button
      android:id="@+id/btnSubtract"
```

```
        android:layout_width="odp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="-"/>

    <Button
        android:id="@+id/btnMultiply"
        android:layout_width="odp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="×"/>

    <Button
        android:id="@+id/btnDivide"
        android:layout_width="odp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="÷"/>
  </LinearLayout>
</LinearLayout>
```

✅ **UI Features:**

✓ Two EditText fields for user input.

✓ TextView to display the result.

✓ Four Button components for arithmetic operations.

✓ Linear layout for better organization.

📌 **Example UI Preview:**

Enter first number: [12.5]

Enter second number: [3.5]

Result: 16.0

[+] [-] [×] [÷]

Step 3: Writing the Java/Kotlin Code for Functionality

3.1 Java Code for MainActivity.java

📌 **Navigate to app/src/main/java/com/example/simplecalculator/MainActivity.java and update the code:**

```
package com.example.simplecalculator;


import android.os.Bundle;

import android.view.View;

import android.widget.Button;

import android.widget.EditText;

import android.widget.TextView;

import androidx.appcompat.app.AppCompatActivity;


public class MainActivity extends AppCompatActivity {
```

```
EditText etNumber1, etNumber2;
TextView tvResult;
Button btnAdd, btnSubtract, btnMultiply, btnDivide;

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

    etNumber1 = findViewById(R.id.etNumber1);
    etNumber2 = findViewById(R.id.etNumber2);
    tvResult = findViewById(R.id.tvResult);
    btnAdd = findViewById(R.id.btnAdd);
    btnSubtract = findViewById(R.id.btnSubtract);
    btnMultiply = findViewById(R.id.btnMultiply);
    btnDivide = findViewById(R.id.btnDivide);

    btnAdd.setOnClickListener(view -> performOperation("+"));
    btnSubtract.setOnClickListener(view -> performOperation("-"));
    btnMultiply.setOnClickListener(view -> performOperation("*"));
    btnDivide.setOnClickListener(view -> performOperation("/"));
}
```

```
private void performOperation(String operator) {
    String num1 = etNumber1.getText().toString();
    String num2 = etNumber2.getText().toString();

    if (num1.isEmpty() || num2.isEmpty()) {
        tvResult.setText("Please enter both numbers");
        return;
    }

    double number1 = Double.parseDouble(num1);
    double number2 = Double.parseDouble(num2);
    double result = 0;

    switch (operator) {
        case "+":
            result = number1 + number2;
            break;
        case "-":
            result = number1 - number2;
            break;
        case "*":
```

```
            result = number1 * number2;
            break;
        case "/":
            if (number2 != 0) {
                result = number1 / number2;
            } else {
                tvResult.setText("Error: Division by zero");
                return;
            }
            break;
    }

    tvResult.setText("Result: " + result);
  }
}
```

3.2 Kotlin Code for MainActivity.kt

📌 **Navigate to app/src/main/java/com/example/simplecalculator/MainActivity.kt and update the code:**

```
package com.example.simplecalculator


import android.os.Bundle
```

```
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    private lateinit var etNumber1: EditText
    private lateinit var etNumber2: EditText
    private lateinit var tvResult: TextView
    private lateinit var btnAdd: Button
    private lateinit var btnSubtract: Button
    private lateinit var btnMultiply: Button
    private lateinit var btnDivide: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        etNumber1 = findViewById(R.id.etNumber1)
        etNumber2 = findViewById(R.id.etNumber2)
        tvResult = findViewById(R.id.tvResult)
        btnAdd = findViewById(R.id.btnAdd)
```

```
    btnSubtract = findViewById(R.id.btnSubtract)
    btnMultiply = findViewById(R.id.btnMultiply)
    btnDivide = findViewById(R.id.btnDivide)

    btnAdd.setOnClickListener { performOperation("+") }
    btnSubtract.setOnClickListener { performOperation("-") }
    btnMultiply.setOnClickListener { performOperation("*") }
    btnDivide.setOnClickListener { performOperation("/") }
}

private fun performOperation(operator: String) {
    val num1 = etNumber1.text.toString()
    val num2 = etNumber2.text.toString()

    if (num1.isEmpty() || num2.isEmpty()) {
        tvResult.text = "Please enter both numbers"
        return
    }

    val number1 = num1.toDouble()
    val number2 = num2.toDouble()
    val result = when (operator) {
```

```
            "+" -> number1 + number2
            "-" -> number1 - number2
            "*" -> number1 * number2
            "/" -> if (number2 != 0.0) number1 / number2 else {
                tvResult.text = "Error: Division by zero"
                return
            }
            else -> 0.0
        }

        tvResult.text = "Result: $result"
    }
}
```

Conclusion

✅ **We successfully built a simple calculator app in Android using Java/Kotlin.**

✅ **The app performs basic arithmetic operations: addition, subtraction, multiplication, and division.**

✅ **It features a user-friendly UI and handles edge cases like division by zero.**