ZOLAXTECH

Android App Development
The Complete Guide

From Zero to Java Mastery to Published Apps

A Beginner-to-Advanced Roadmap for Java, Android Studio & Modern App Architecture

---

zolaxtech.com.et

Free Learning Resource

Table of Contents

Table of Contents 2
Welcome 3
What you will be able to build by the end 3
Prerequisites 3
1.1 Installing the JDK 5
1.2 Your First Program 5
Breaking it down 5
2.1 Declaring Variables 6
2.2 Primitive Data Types 6
2.3 Operators 6
3.1 Conditional Statements 8
3.2 Switch Statements 8
3.3 Loops 8
4.1 Classes and Objects 10
4.2 The Four Pillars of OOP 10
Encapsulation 10
Inheritance 11
Polymorphism 11
Abstraction 11
5.1 Arrays 13
5.2 ArrayList 13
5.3 HashMap 13
7.1 Installation ..... **Error! Bookmark not defined.**
7.2 Creating Your First Project 17
7.3 Project Structure 17
9.1 A Basic Layout 19
9.2 Connecting Java to XML 19
9.3 Common Layout Types 19
10.1 Explicit Intents 21
10.2 Receiving Data 21
10.3 Implicit Intents 21
11.1 The Model Class 23
11.2 The Adapter 23

13.1 SharedPreferences for small key-value data 26
13.2 Room for structured, queryable data 26
18.1 Unit Tests (JUnit) 32
18.2 UI Tests (Espresso) 32
Suggested Learning Path 34
Quick Reference: Java vs. Android Concepts 34

Welcome

This guide takes you from complete beginner to confident, job-ready Android developer. It is built around one core idea: you cannot build great Android apps without genuinely understanding Java. So before a single line of Android code appears, this guide gives you a full, practical grounding in the Java language variables, control flow, object-oriented programming, collections, and exception handling with runnable examples throughout.

Once your Java foundation is solid, the guide moves into Android Studio, the building blocks of an app (Activities, Layouts, Intents), then intermediate skills (RecyclerView, Fragments, local storage, networking), and finally advanced, professional-grade topics (MVVM architecture, ViewModel and LiveData, coroutines, Firebase, and publishing to the Play Store).

> **How to use this guide?**
> Read it in order the first time through. Each chapter builds on the last. Type out the code examples yourself rather than copy-pasting muscle memory is how syntax becomes second nature. Every chapter ends with a short practice exercise; do not skip these.

What you will be able to build by the end

- A multi-screen Android app with navigation between activities and fragments
- Apps that fetch and display live data from the internet
- Apps that save data locally and remember user preferences
- Apps built on a clean, testable MVVM architecture used by professional teams
- A finished app packaged and ready to publish on the Google Play Store

Prerequisites

None. This guide assumes no prior programming experience. If you have coded before, feel free to skim Part 1 and move faster through the Java fundamentals.

PART ONE

Java Foundations

Java is the classic language of Android development, and even in a Kotlin-first world, Java skills remain essential the Android SDK itself is written in Java, countless production codebases are Java, and the object-oriented concepts you learn here transfer directly. This part is a self-contained crash course. Work through it carefully.

CHAPTER 1
Getting Started with Java

Setting up your environment and writing your first program

1.1 Installing the JDK

Java programs run on the Java Virtual Machine (JVM), and you write them using the Java Development Kit (JDK). Download the latest JDK (17 or newer) from Oracle or use the free OpenJDK distribution (Adoptium/Temurin is a good choice). After installing, confirm it from a terminal:

```
java -version
javac -version
```

If both commands print version numbers, you are ready to compile and run Java code.

1.2 Your First Program

Every Java application starts with a class containing a main method — this is the entry point the JVM looks for when it runs your program.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, ZolaxTech!");
    }
}
```

Save this as HelloWorld.java, then compile and run it:

```
javac HelloWorld.java
java HelloWorld
```

Breaking it down

- public class HelloWorld — declares a class; the filename must match the class name.
- public static void main(String[] args) — the method the JVM calls first.
- System.out.println(...) — prints text to the console, followed by a new line.

✎ Try it yourself

1. Modify the program to print your own name instead of “ZolaxTech”.
2. Add a second println statement that prints today’s date as plain text.

CHAPTER 2

Variables, Data Types & Operators

Storing and manipulating data in Java

2.1 Declaring Variables

A variable is a named container for a value. Java is statically typed, meaning you must declare the type of data a variable will hold before you use it.

```
int age = 25;
double price = 19.99;
char grade = 'A';
boolean isActive = true;
String name = "Selam";
```

2.2 Primitive Data Types

Type	Size	Example	Used for
int	4 bytes	int count = 10;	Whole numbers
double	8 bytes	double gpa = 3.75;	Decimal numbers
boolean	1 bit	boolean isDone = false;	True/false values
char	2 bytes	char letter = 'z';	Single characters
long	8 bytes	long views = 9000000000L;	Very large whole numbers

2.3 Operators

Java supports the standard set of arithmetic, comparison, and logical operators:

```
int sum = 5 + 3;        // arithmetic: + - * / %
boolean isEqual = (5 == 5);   // comparison: == != > < >= <=
boolean result = (true && false); // logical: && || !
```

Common beginner mistake

Use == to compare primitive values, but never use == to compare the contents of two String objects use .equals() instead. This trips up almost every new Java developer at least once.

✎ Try it yourself

1. Declare variables for a product name, price, and quantity, then calculate and print the total cost.
2. Write an expression that checks whether a number is even using the % operator.

CHAPTER 3

Control Flow

Making decisions and repeating actions

3.1 Conditional Statements

```
int score = 82;

if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else {
    System.out.println("Grade: C or below");
}
```

3.2 Switch Statements

```
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    default: System.out.println("Another day");
}
```

3.3 Loops

Loops let you repeat a block of code. Java has three main loop types:

```
// for loop   known number of repetitions
for (int i = 0; i < 5; i++) {
    System.out.println("Count: " + i);
}

// while loop   repeats while a condition is true
int n = 0;
while (n < 3) {
    System.out.println("n is " + n);
    n++;
}

// do-while   runs the body at least once
```

```
int x = 0;
do {
   System.out.println("x = " + x);
   x++;
} while (x < 2);
```

✎ Try it yourself

1. Write a loop that prints all even numbers between 1 and 20.
2. Write a program that uses if/else to categorize a temperature as Cold, Mild, or Hot.

CHAPTER 4
Object-Oriented Programming

Classes, objects, and the four pillars of OOP

Object-oriented programming (OOP) is the backbone of Java and of Android development, where nearly everything (an Activity, a Button, a data model) is an object built from a class. Master this chapter and Android's structure will make much more sense.

4.1 Classes and Objects

A class is a blueprint; an object is an instance created from that blueprint.

```
public class Car {
    // Fields (state)
    String model;
    int year;

    // Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    // Method (behavior)
    public void displayInfo() {
        System.out.println(year + " " + model);
    }
}

// Creating and using an object
Car myCar = new Car("Toyota Corolla", 2022);
myCar.displayInfo();
```

4.2 The Four Pillars of OOP

Encapsulation

Bundling data and the methods that operate on it inside a class, and restricting direct access to that data using private fields with public getter/setter methods.

```
public class Account {
    private double balance;
```

```
    public double getBalance() { return balance; }
    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
}
```

Inheritance

A class can inherit fields and methods from another class using extends, letting you reuse and specialize behavior.

```
public class Animal {
    public void eat() { System.out.println("Eating..."); }
}

public class Dog extends Animal {
    public void bark() { System.out.println("Woof!"); }
}

Dog myDog = new Dog();
myDog.eat();   // inherited from Animal
myDog.bark();  // defined in Dog
```

Polymorphism

The same method name behaves differently depending on the object calling it, typically through method overriding.

```
public class Shape {
    public double area() { return 0; }
}
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    @Override
    public double area() { return Math.PI * radius * radius; }
}
```

Abstraction

Hiding implementation detail behind a simple interface, using abstract classes or interfaces so callers only need to know what a method does, not how.

```
public interface Payable {
```

```
    double calculatePayment();
}

public class Employee implements Payable {
    private double hours, rate;
    public Employee(double hours, double rate) {
        this.hours = hours; this.rate = rate;
    }
    @Override
    public double calculatePayment() { return hours * rate; }
}
```

Why this matters for Android?

Every Activity you write extends the Android AppCompatActivity class (inheritance). You will implement listener interfaces like View.OnClickListener (abstraction/polymorphism). And you'll encapsulate app data inside model classes constantly. OOP is not optional theory here it's the daily grammar of Android code.

✎ Try it yourself

1. Create a Shape hierarchy with Circle, Rectangle, and Triangle, each overriding an area() method.
2. Create a BankAccount class with private balance, and public deposit() and withdraw() methods that validate input.

CHAPTER 5

Collections & Data Structures

Storing groups of data efficiently

5.1 Arrays

```
int[] numbers = {10, 20, 30, 40};
String[] names = new String[3];
names[0] = "Abel";
System.out.println(numbers[1]); // 20
```

5.2 ArrayList

Unlike arrays, an ArrayList can grow and shrink dynamically the collection you will use constantly, including inside RecyclerView adapters later in this guide.

```
import java.util.ArrayList;
import java.util.List;

List<String> tasks = new ArrayList<>();
tasks.add("Buy groceries");
tasks.add("Finish app UI");
tasks.remove("Buy groceries");

for (String task : tasks) {
    System.out.println(task);
}
```

5.3 HashMap

Stores key-value pairs for fast lookups by key.

```
import java.util.HashMap;
import java.util.Map;

Map<String, Integer> prices = new HashMap<>();
prices.put("Coffee", 120);
prices.put("Tea", 90);
System.out.println(prices.get("Coffee")); // 120
```

✎ Try it yourself

1. Build an ArrayList of student names and print them using a for-each loop.

2. Build a HashMap that maps product names to prices, then print the total value of all products.

CHAPTER 6

Exception Handling

Writing code that fails gracefully

Real apps encounter unexpected situations missing network connections, bad user input, null values. Java's try/catch mechanism lets you handle these gracefully instead of crashing.

```
try {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]); // out of bounds
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("This always runs.");
}
```

You can also define and throw your own exceptions to enforce rules in your own code:

```
public void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
    this.age = age;
}
```

Why this matters for Android?

You will use try/catch constantly parsing JSON from a network response, reading a file, or converting text input into a number. Handling exceptions well is the difference between an app that crashes and one that shows the user a friendly message.

PART TWO

Android Fundamentals

With Java under your belt, it's time to build real apps. This part covers Android Studio, the anatomy of an app, and the essential building blocks every Android developer uses every day.

CHAPTER 7

Setting Up Android Studio

Your development environment

7.1 Installation

1. Download Android Studio from developer.android.com/studio.
2. Run the installer and accept the default components (Android SDK, Android Virtual Device, SDK tools).
3. On first launch, open the SDK Manager and install the latest stable Android SDK Platform.
4. Create a Virtual Device (AVD) via Device Manager, or connect a physical Android phone with USB debugging enabled.

7.2 Creating Your First Project

1. Choose File > New Project.
2. Select “Empty Views Activity”.
3. Set the Language to Java (this is important Android Studio defaults to Kotlin).
4. Set a minimum SDK of API 24 (Android 7.0) for broad device compatibility.

7.3 Project Structure

Folder / File	Purpose
app/java/	Your Java source code (Activities, classes, adapters)
app/res/layout/	XML files that define screen layouts
app/res/values/	Strings, colors, dimensions, and themes
app/res/drawable/	Images and vector icons
AndroidManifest.xml	Declares your app's components, permissions, and metadata
build.gradle (Module)	App-level dependencies and build configuration

CHAPTER 8

Activities and the Lifecycle

The building block of every screen

An Activity represents a single screen in your app. Understanding its lifecycle — the sequence of methods Android calls as a screen is created, shown, hidden, and destroyed — is essential to writing bug-free apps.

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

    @Override
    protected void onStart() { super.onStart(); }

    @Override
    protected void onResume() { super.onResume(); }

    @Override
    protected void onPause() { super.onPause(); }

    @Override
    protected void onStop() { super.onStop(); }

    @Override
    protected void onDestroy() { super.onDestroy(); }
}
```

Method	Called when…
onCreate()	The activity is first created — inflate layout here
onStart()	The activity becomes visible
onResume()	The activity starts interacting with the user
onPause()	Another activity takes focus (this one is partially hidden)
onStop()	The activity is no longer visible
onDestroy()	The activity is being removed from memory

CHAPTER 9

Layouts and Views

Designing screens with XML

9.1 A Basic Layout

```
<?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="16dp">

    <TextView
        android:id="@+id/titleText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to ZolaxTech"
        android:textSize="22sp" />

    <Button
        android:id="@+id/actionButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Continue" />

</LinearLayout>
```

9.2 Connecting Java to XML

```
TextView titleText = findViewById(R.id.titleText);
Button actionButton = findViewById(R.id.actionButton);

actionButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        titleText.setText("Button clicked!");
    }
});
```

9.3 Common Layout Types

- LinearLayout arranges children in a single row or column

- ConstraintLayout positions children using flexible constraints; the modern default
- RelativeLayout positions children relative to each other or the parent
- FrameLayout stacks children on top of one another

✎ Try it yourself

1. Build a login screen layout with two EditText fields and one Button.
2. Wire up the button's onClick listener to show a Toast message using the entered username.

CHAPTER 10

Intents & Navigation

Moving between screens

An Intent is a messaging object used to request an action from another app component most commonly, to start a new Activity.

10.1 Explicit Intents

```
Intent intent = new Intent(MainActivity.this, DetailActivity.class);
intent.putExtra("username", "selam123");
startActivity(intent);
```

10.2 Receiving Data

```
String username = getIntent().getStringExtra("username");
```

10.3 Implicit Intents

Used to ask the system to perform an action, letting any capable app respond for example, opening a web page or dialing a number:

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

PART THREE

Intermediate Skills

These are the skills that separate a tutorial app from a real one: showing lists of data, breaking screens into reusable pieces, saving data, and talking to the internet.

CHAPTER 11

RecyclerView

Displaying scrollable lists efficiently

RecyclerView is the standard way to display large or dynamic lists in Android it recycles item views as the user scrolls, keeping performance smooth.

11.1 The Model Class

```
public class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name; this.price = price;
    }
    public String getName() { return name; }
    public double getPrice() { return price; }
}
```

11.2 The Adapter

```
public class ProductAdapter extends RecyclerView.Adapter<ProductAdapter.ViewHolder> {
    private List<Product> productList;

    public ProductAdapter(List<Product> productList) {
        this.productList = productList;
    }

    static class ViewHolder extends RecyclerView.ViewHolder {
        TextView nameText, priceText;
        ViewHolder(View itemView) {
            super(itemView);
            nameText = itemView.findViewById(R.id.nameText);
            priceText = itemView.findViewById(R.id.priceText);
        }
    }

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

```
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        Product product = productList.get(position);
        holder.nameText.setText(product.getName());
        holder.priceText.setText(String.valueOf(product.getPrice()));
    }

    @Override
    public int getItemCount() { return productList.size(); }
}
```

CHAPTER 12

Fragments

Reusable, modular pieces of UI

A Fragment represents a portion of a UI within an Activity, and can be reused across multiple activities essential for tablets, navigation drawers, and bottom navigation bars.

```
public class HomeFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_home, container, false);
    }
}
```

Fragments are added to an Activity using the FragmentManager:

```
getSupportFragmentManager()
    .beginTransaction()
    .replace(R.id.fragmentContainer, new HomeFragment())
    .commit();
```

CHAPTER 13

Local Data Storage

SharedPreferences, SQLite, and Room

13.1 SharedPreferences for small key-value data

```
SharedPreferences prefs = getSharedPreferences("AppPrefs", MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.putString("username", "selam123");
editor.apply();

String username = prefs.getString("username", "guest");
```

13.2 Room for structured, queryable data

Room is the modern, recommended library for local databases, built on top of SQLite with compile-time query verification.

```
@Entity
public class Note {
    @PrimaryKey(autoGenerate = true)
    public int id;
    public String title;
    public String content;
}

@Dao
public interface NoteDao {
    @Insert
    void insert(Note note);

    @Query("SELECT * FROM Note")
    List<Note> getAllNotes();
}

@Database(entities = {Note.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract NoteDao noteDao();
}
```

CHAPTER 14
Networking

Talking to web APIs with Retrofit

Almost every real app fetches data from a server. Retrofit is the industry-standard HTTP client for Android, turning a REST API into a simple Java interface.

```
public interface ApiService {
    @GET("users/{id}")
    Call<User> getUser(@Path("id") int id);
}

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();

ApiService api = retrofit.create(ApiService.class);
api.getUser(1).enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        User user = response.body();
    }
    @Override
    public void onFailure(Call<User> call, Throwable t) {
        Log.e("API", "Request failed", t);
    }
});
```

Don't forget

Add the internet permission to AndroidManifest.xml: <uses-permission android:name="android.permission.INTERNET" />

PART FOUR

Advanced Android

This part covers how professional teams structure real production apps: separating concerns with MVVM, handling background work safely, integrating cloud services, and shipping to users.

CHAPTER 15

MVVM Architecture

Model – View – ViewModel

MVVM separates an app into three layers, making code easier to test, maintain, and extend:

Layer	Responsibility
Model	Data and business logic (e.g. Room entities, repositories, network models)
View	The Activity/Fragment displays data and forwards user actions
ViewModel	Holds UI state, survives configuration changes, exposes data via LiveData

```
public class ProductViewModel extends ViewModel {
    private final MutableLiveData<List<Product>> products = new MutableLiveData<>();

    public LiveData<List<Product>> getProducts() { return products; }

    public void loadProducts() {
        // fetch from repository, then:
        products.setValue(fetchedList);
    }
}
```

The Activity simply observes the data it never fetches or owns it directly:

```
ProductViewModel viewModel = new ViewModelProvider(this).get(ProductViewModel.class);
viewModel.getProducts().observe(this, productList -> {
    adapter.updateList(productList);
});
viewModel.loadProducts();
```

CHAPTER 16

Background Work & Threading

Keeping the UI responsive

Android will crash your app (an ANR Application Not Responding) if you block the main thread with slow work like network calls or database queries. Modern Android apps typically use Executors or the WorkManager library for background tasks.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Handler mainHandler = new Handler(Looper.getMainLooper());

executor.execute(() -> {
   List<Note> notes = database.noteDao().getAllNotes(); // background thread
   mainHandler.post(() -> {
      adapter.setNotes(notes); // back on the UI thread
   });
});
```

Rule of thumb

Never touch a View from a background thread. Always hop back to the main thread before updating UI.

CHAPTER 17

Firebase Integration

Cloud backend without building your own server

- Firebase Authentication email/password, Google, and phone-number sign-in
- Cloud Firestore a scalable, real-time NoSQL database
- Firebase Cloud Messaging (FCM) push notifications
- Firebase Crashlytics real-time crash reporting

```
FirebaseFirestore db = FirebaseFirestore.getInstance();
Map<String, Object> user = new HashMap<>();
user.put("name", "Selam");
user.put("email", "selam@example.com");

db.collection("users").add(user)
    .addOnSuccessListener(docRef -> Log.d("Firebase", "Saved: " + docRef.getId()))
    .addOnFailureListener(e -> Log.e("Firebase", "Error saving user", e));
```

CHAPTER 18

Testing Your App

Catching bugs before your users' do

18.1 Unit Tests (JUnit)

```
public class CalculatorTest {
    @Test
    public void addition_isCorrect() {
        Calculator calc = new Calculator();
        assertEquals(4, calc.add(2, 2));
    }
}
```

18.2 UI Tests (Espresso)

```
onView(withId(R.id.actionButton)).perform(click());
onView(withId(R.id.titleText)).check(matches(withText("Button clicked!")));
```

CHAPTER 19

Publishing to Google Play

Shipping your app to the world

1. Generate a signed APK or Android App Bundle (Build > Generate Signed Bundle / APK).
2. Create a Google Play Developer account (one-time registration fee).
3. In Play Console, create a new app entry and fill in its store listing (title, description, screenshots, icon).
4. Complete the content rating questionnaire and privacy policy requirement.
5. Upload your App Bundle to a testing or production track.
6. Submit for review approval typically takes a few hours to a few days.

> **Tip**
> Always test a release build (not just a debug build) on a real device before submitting some bugs, especially around code shrinking (ProGuard/R8), only appear in release mode.

APPENDIX

Where to Go Next

Suggested Learning Path

1. Rebuild every code example in this guide by hand in Android Studio.
2. Build three small practice apps: a to-do list, a unit converter, and a weather app using a public API.
3. Learn Kotlin — Android's other first-class language — once Java and Android fundamentals feel comfortable.
4. Study Jetpack libraries in depth: Navigation Component, WorkManager, Paging, DataStore.
5. Contribute to or read the source of an open-source Android app on GitHub.
6. Publish a small but complete app to the Play Store — finishing and shipping teaches lessons tutorials cannot.

Quick Reference: Java vs. Android Concepts

Java Concept	Where You'll Use It in Android
Classes & Objects	Every Activity, Fragment, Adapter, and data Model
Inheritance	extends AppCompatActivity, extends Fragment
Interfaces	OnClickListener, Callback, Dao
Collections (List/Map)	RecyclerView data, JSON parsing, caching
Exception Handling	Network errors, file I/O, input parsing
Generics	RecyclerView.Adapter<T>, LiveData<T>, Retrofit Call<T>

Thank you for learning with ZolaxTech

More free resources at zolaxtech.com.et