Crash Land on Flutter - 1

shahriar inan

1st Sesh

Into To Flutter

Birth of Flutter

- **Sky**
    - Flutter's journey began as an experimental project called "Sky" within Google.
    - The goal was to explore ways to achieve consistently high performance across different platforms.
    - Sky showed promise in delivering smooth, jank-free (no stuttering or delays) user interfaces.
- **Flutter's Debut**
    - In 2015, Google unveiled Flutter at the Dart developer summit.
    - It was positioned as a UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.
    - The first stable release of Flutter came out in December 2018.

Key Principles

- **Everything is a Widget**
    - The core philosophy of Flutter is that everything you see on the screen is a widget.
    - Widgets are the building blocks of the user interface, ranging from simple buttons and text to complex layouts and animations.
    - This composable nature allows for a great deal of flexibility and customization.
- **Hot Reload**
    - Flutter introduced the "hot reload" feature, which allows developers to see the changes they make to the code instantly reflected in the running app.
    - This dramatically sped up the development process, enabling faster iteration and experimentation.
- **Cross-Platform Consistency**
    - Flutter aimed to solve the problem of fragmentation across different platforms.
    - By using its own rendering engine (Skia), Flutter could achieve a consistent look and feel across iOS, Android, web, and desktop, without relying on platform-specific UI components.

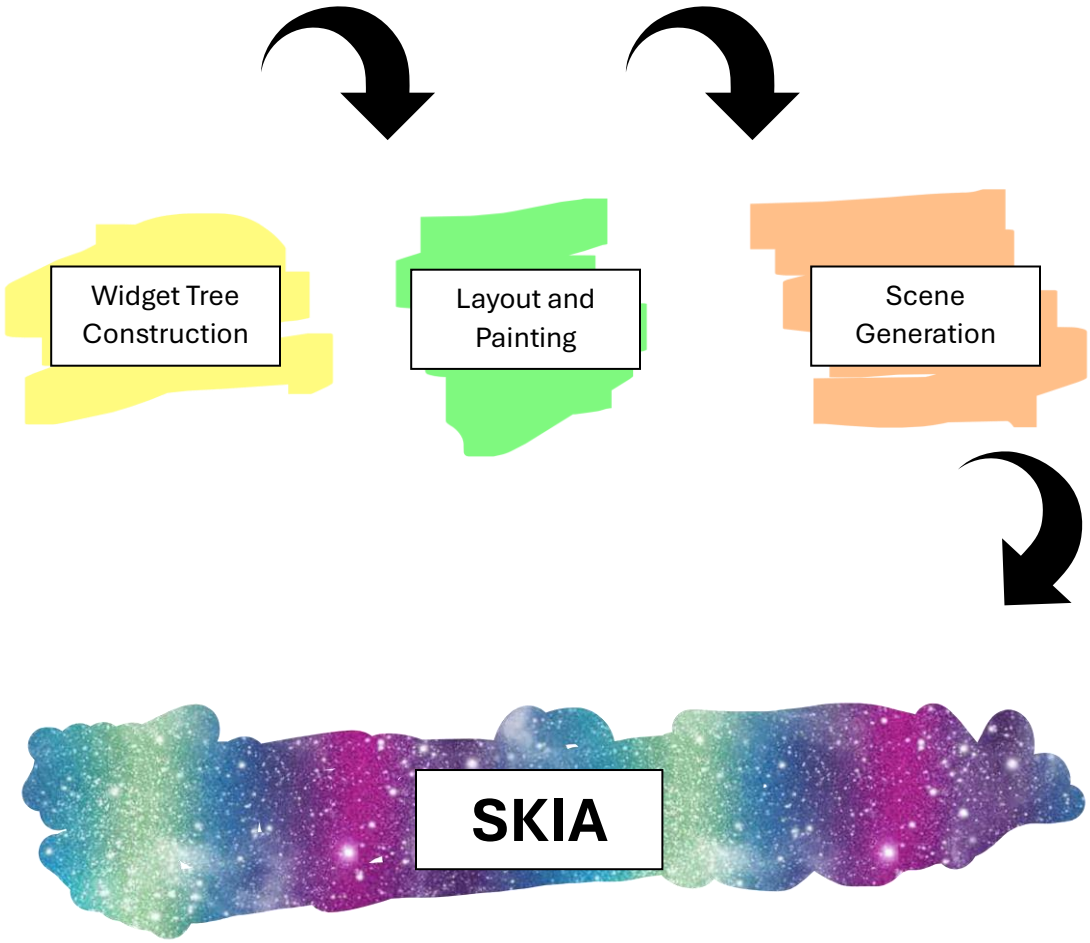
Growth and Adoption

- **Community Support**
    - The Flutter community quickly embraced the framework due to its ease of use, performance benefits, and the ability to build apps for multiple platforms from a single codebase.
    - Google actively fostered the community by providing resources, documentation, and support.
- **Industry Adoption**
    - Flutter's popularity grew steadily, and many companies started adopting it for their app development.
    - Notable companies using Flutter include Google (for Google Ads and Google Pay), Alibaba (for Xianyu), BMW, eBay, and many more.
- **Expanding Capabilities**
    - Over time, Flutter's capabilities expanded beyond mobile to include web and desktop support.
    - Flutter also integrated with other Google technologies like Firebase, making it easier to build full-featured applications.

Flutter Today

- **Maturity and Stability**
    - Flutter has matured into a stable and reliable framework for building high-quality apps across different platforms.
    - It continues to evolve with regular updates, new features, and performance improvements.
- **Wide Range of Use Cases**
    - Flutter is now used for a variety of applications, from consumer apps and games to enterprise solutions and embedded systems.
- **Thriving Ecosystem**
    - Flutter has a vibrant ecosystem of packages, plugins, and tools that further extend its capabilities.
    - The community continues to be a driving force behind its growth and success.

More on Skia



Flutter's Rendering Pipeline with Skia

1. **Widget Tree Construction:**
    - Flutter apps are built using a hierarchy of widgets, where each widget defines a part of the user interface.
    - When the app starts or when the UI needs to be updated, Flutter builds a tree-like structure of these widgets, known as the "widget tree."
2. **Layout and Painting:**
    - **Layout:** The widget tree is traversed, and each widget determines its size and position within the overall layout. This phase ensures that all widgets are correctly placed on the screen.

- **Painting:** Each widget then describes how it should be painted onto a canvas. Widgets provide instructions on drawing shapes, text, images, and applying visual effects.

3. **Scene Generation:**
    - Flutter combines the layout and painting information to create a "scene." This scene is an abstract representation of what should be rendered on the screen.
4. **Rasterization (with Skia):**
    - **Scene to Skia Commands:** The scene is then passed to Skia, Flutter's rendering engine.
    - **Skia Canvas:** Skia uses its own "canvas" concept to execute drawing commands.
    - **Drawing:** Skia takes the scene description and translates it into a series of low-level drawing commands, such as drawing lines, filling shapes, applying textures, and blending colors.
    - **Hardware Acceleration:** Skia leverages the capabilities of the device's GPU (graphics processing unit) or other hardware acceleration features whenever possible to perform these drawing operations efficiently.
5. **Display:**
    - The final rasterized image generated by Skia is then presented on the device's screen.

1: The ABCs

1. Basic Syntax and Structure

Hello World:

```dart
void main() {
  print("Hello World!");
}
```

- **void main():** This is the entry point of your Dart program. Execution begins here.
- **print():** This function displays the text within the parentheses to the console.

Comments:

```
// This is a single-line comment

/*
This is a
multi-line comment
*/
```

- **Single-line comments:** Start with // and continue to the end of the line.
- **Multi-line comments:** Enclosed within /* and */.

2. Variables

Declaring and initializing:

```
var name = "Alice"; // Type is inferred as String
final age = 30;     // Value cannot be changed after initialization
const pi = 3.14159;  // Compile-time constant

print(name);  // Output: Alice
print(age);   // Output: 30
print(pi);    // Output: 3.14159
```

- **var:** Use when you want the type to be inferred from the initial value.
- **final:** Use when the value should not change after initialization. The type can be inferred or explicitly declared.
- **const:** Use for compile-time constants. The value and type must be known at compile time.

3. Data Types

```
int count = 10;
double price = 19.99;
String message = "Hello there!";
bool isActive = true;
dynamic value = 42; // Can hold any type of data

print(count.runtimeType);    // Output: int
print(price.runtimeType);    // Output: double
print(message.runtimeType);  // Output: String
print(isActive.runtimeType); // Output: bool
print(value.runtimeType);    // Output: int (initially)

value = "Now it's a string";
print(value.runtimeType);    // Output: String
```

- **int:** Represents whole numbers (integers).
- **double:** Represents numbers with decimal points (floating-point numbers).
- **String:** Represents a sequence of characters (text).
- **bool:** Represents a logical value, either true or false.
- **dynamic:** Can hold any type of data. Use with caution, as it can lead to runtime type errors.

2. Control Flow Statements

Conditional Statements

- **if**

```
int age = 25;

if (age >= 18) {
  print("You are an adult.");
}
```

- **if-else**

```
int score = 75;

if (score >= 80) {
  print("Excellent!");
} else {
  print("Good job!");
}
```

- **if-else if-else**

```
String day = "Monday";

if (day == "Saturday" || day == "Sunday") {
  print("It's the weekend!");
} else if (day == "Friday") {
  print("Almost the weekend!");
} else {
  print("It's a weekday.");
}
```

- **switch**

```
int month = 9;

switch (month) {
  case 1: print("January"); break;
  case 2: print("February"); break;
  // ... other cases
  case 9: print("September"); break;
  default: print("Invalid month");
}
```

Loops

- **for loop**

```
for (int i = 0; i < 5; i++) {
  print("Iteration $i");
}
```

- **while loop**

```
int count = 0;
while (count < 3) {
  print("Count: $count");
  count++;
}
```

- **do-while loop**

```
int num = 10;
do {
  print("Num: $num");
  num--;
} while (num > 0);
```

- **forEach loop (for iterables like lists)**

```
List<String> fruits = ["apple", "banana", "orange"];

fruits.forEach((fruit) {
  print(fruit);
});
```

Explanation

- **Conditional Statements: These statements allow your program to make decisions and execute different code blocks based on specific conditions.**
    - **if: Executes a block of code if a condition is true.**
    - **else: Executes a block of code if the preceding if condition is false.**
    - **else if: Allows you to check multiple conditions in sequence.**
    - **switch: Provides a concise way to handle multiple possible values of a variable or expression.**
- **Loops: Loops enable you to repeat a block of code multiple times until a certain condition is no longer true.**
    - **for: Used when you know the number of iterations in advance.**
    - **while: Used when you want to repeat code as long as a condition is true.**
    - **do-while: Similar to while, but the code block is executed at least once before checking the condition.**
    - **forEach: A convenient way to iterate over elements in a collection (like a list).**

3: Functions

3.1. Defining Functions

A function is a block of code that performs a specific task. Here's how to define and use functions in

```dart
void main() {
  // Calling the function
  printHello();
}

// Defining the function
void printHello() {
  print('Hello, World!');
}
```

3.2. Parameters

Functions can take parameters to perform tasks with given inputs. Dart supports positional, named, and optional parameters.

3.2.1. Positional Parameters

Parameters are passed in the order they are defined.

```dart
void main() {
  // Calling the function with arguments
  greet('Alice', 25);
}

// Defining the function with positional parameters
void greet(String name, int age) {
  print('Hello, $name! You are $age years old.');
}
```

3.2.2. Named Parameters

Named parameters improve readability by naming each parameter during the function call. Named parameters can also be optional.

```
void main() {
  // Calling the function with named arguments
  greet(name: 'Alice', age: 25);
}

// Defining the function with named parameters
void greet({required String name, required int age}) {
  print('Hello, $name! You are $age years old.');
}
```

Default Values for Named Parameters

You can provide default values for named parameters, which are used if no value is passed during the function call.

```
void main() {
  // Calling the function without providing an age
  greet(name: 'Alice');
}

// Defining the function with a default value for a named parameter
void greet({required String name, int age = 18}) {
  print('Hello, $name! You are $age years old.');
}
```

3.2.3. Optional Parameters

Optional parameters can either be positional or named. Positional optional parameters are wrapped in square brackets [].

```
void main() {
  // Calling the function with an optional argument
  greet('Alice');
  greet('Bob', 30);
}

// Defining the function with an optional positional parameter
void greet(String name, [int? age]) {
  if (age != null) {
    print('Hello, $name! You are $age years old.');
  } else {
    print('Hello, $name!');
  }
}
```

3.3. Return Types

Functions can return values using the return keyword. The return type must be specified.

```
void main() {
  // Calling the function and storing the result
  int result = add(5, 3);
  print('Result: $result');
}

// Defining the function with a return type
int add(int a, int b) {
  return a + b;
}
```

3.4. Anonymous Functions

Anonymous functions (also known as lambdas or closures) are functions without a name. They are often used as arguments to higher-order functions.

```
void main() {
  // Using an anonymous function in a list's forEach method
  List<String> names = ['Alice', 'Bob', 'Charlie'];

  names.forEach((name) {
    print('Hello, $name!');
  });

  // Assigning an anonymous function to a variable
  var multiply = (int a, int b) {
    return a * b;
  };

  // Calling the anonymous function
  print('Product: ${multiply(4, 5)}');
}
```

Arrow Functions

For concise functions, you can use arrow syntax (=>) for functions that contain a single expression.

```
void main() {
  // Calling the arrow function
  int result = add(5, 3);
  print('Result: $result');
}

// Defining an arrow function
int add(int a, int b) => a + b;
```

4: Collections

Dart provides various collection types like Lists, Sets, and Maps. Here are detailed examples for each collection type:

4.1. Lists

Lists are ordered collections of elements. They can be of fixed or growable length.

Creating and Initializing Lists:

```
void main() {
  // Creating a fixed-length list
  var fixedList = List<int>.filled(3, 0);
  fixedList[0] = 1;
  fixedList[1] = 2;
  fixedList[2] = 3;
  print('Fixed List: $fixedList');

  // Creating a growable list
  var growableList = [1, 2, 3];
  growableList.add(4);
  print('Growable List: $growableList');
}
```

Manipulating Lists:

```
void main() {
  var numbers = [1, 2, 3, 4, 5];

  // Accessing elements
  print('First Element: ${numbers[0]}');

  // Updating elements
  numbers[0] = 10;
  print('Updated List: $numbers');

  // Removing elements
  numbers.remove(3);
  print('List after removal: $numbers');

  // Iterating over a list
  print('Iterating over list:');
  for (var number in numbers) {
    print(number);
  }
}
```

4.2. Sets

Sets are collections of unique elements.

Creating and Initializing Sets:

```
void main() {
  var names = <String>{'Alice', 'Bob', 'Charlie'};
  print('Set: $names');

  // Adding elements
  names.add('David');
  print('Set after adding: $names');

  // Removing elements
  names.remove('Alice');
  print('Set after removal: $names');
}
```

Manipulating Sets:

```
void main() {
  var set1 = {1, 2, 3};
  var set2 = {3, 4, 5};

  // Union of two sets
  var unionSet = set1.union(set2);
  print('Union: $unionSet');

  // Intersection of two sets
  var intersectionSet = set1.intersection(set2);
  print('Intersection: $intersectionSet');

  // Difference between two sets
  var differenceSet = set1.difference(set2);
  print('Difference: $differenceSet');
}
```

4.3. Maps

Maps are collections of key-value pairs.

Creating and Initializing Maps:

```
void main() {
  var person = {'name': 'Alice', 'age': 25};
  print('Map: $person');

  // Adding entries
  person['city'] = 'New York';
  print('Map after adding: $person');

  // Removing entries
  person.remove('age');
  print('Map after removal: $person');
}
```

Manipulating Maps:

```
void main() {
  var person = {'name': 'Alice', 'age': 25, 'city': 'New York'};

  // Accessing values
  print('Name: ${person['name']}');

  // Updating values
  person['age'] = 26;
  print('Updated Map: $person');

  // Iterating over a map
  print('Iterating over map:');
  person.forEach((key, value) {
    print('$key: $value');
  });
}
```

4.4. Iterables

In Dart, an Iterable is an abstract class representing a collection of elements that can be accessed sequentially. The Iterable class is the base class for collections such as List and Set. Iterables are very powerful and offer a wide range of methods for manipulation and transformation.

Creating an Iterable

You can create an Iterable using different collection types like List or Set, as they both implement the Iterable interface.

```
void main() {
  // Creating an Iterable using a List
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  print('Numbers: $numbers');

  // Creating an Iterable using a Set
  Iterable<String> names = {'Alice', 'Bob', 'Charlie'};
  print('Names: $names');
}
```

Accessing Elements

You can access elements of an Iterable using iteration methods like forEach or a for loop.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];

  // Using forEach method
  numbers.forEach((number) {
    print('Number: $number');
  });

  // Using a for loop
  for (var number in numbers) {
    print('Number: $number');
  }
}
```

Common Methods on Iterables

Here are some common methods provided by the Iterable class in Dart:

- map()
- where()
- reduce()
- fold()

- expand()
- take()
- skip()

1. map()

The map() method transforms each element of an Iterable using a provided function and returns a new Iterable.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  var doubled = numbers.map((number) => number * 2);
  print('Doubled: $doubled');
}
```

2. where()

The where() method filters elements based on a provided condition and returns a new Iterable.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  var evenNumbers = numbers.where((number) => number.isEven);
  print('Even Numbers: $evenNumbers');
}
```

3. reduce()

The reduce() method combines all elements of an Iterable into a single value using a provided function.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  var sum = numbers.reduce((value, element) => value + element);
  print('Sum: $sum');
}
```

4. fold()

The fold() method is similar to reduce(), but it allows you to specify an initial value.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  var sum = numbers.fold(0, (previousValue, element) => previousValue + element);
  print('Sum: $sum');
}
```

5. expand()

The expand() method replaces each element with an Iterable and combines the result into a single Iterable.

```
void main() {
  Iterable<int> numbers = [1, 2, 3];
  var expanded = numbers.expand((number) => [number, number * 2]);
  print('Expanded: $expanded');
}
```

6. take()

The take() method returns the first n elements from an Iterable.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  var firstTwo = numbers.take(2);
  print('First Two: $firstTwo');
}
```

7. skip()

The skip() method skips the first n elements and returns the rest of the Iterable.

```
void main() {
  Iterable<int> numbers = [1, 2, 3, 4, 5];
  var skipTwo = numbers.skip(2);
  print('Skip Two: $skipTwo');
}
```

Lazy Evaluation

Imagine you have a friend who loves to collect stamps. You ask your friend to show you all the stamps they have that are blue. Instead of going through their entire collection right away and picking out all the blue stamps to show you, your friend says, "Okay, I'll find the blue stamps, but only when you ask to see them one by one."

So, the next time you ask to see a blue stamp, your friend goes through their collection until they find one and then shows it to you. If you want to see another one, they start from where they left off and keep looking for the next blue stamp.

This way, your friend doesn't spend a lot of time upfront finding all the blue stamps but instead finds them as you need them.

Lazy Evaluation in Programming

In programming, lazy evaluation works in a similar way. Instead of performing all the calculations or operations at once, it waits until the result is actually needed. This can save time and resources, especially when dealing with large collections of data.

Here's a simple analogy using Dart:

Lazy Evaluation with Iterables

Let's say we have a list of numbers, and we want to find the even numbers. Using lazy evaluation, we don't filter all the even numbers immediately. Instead, we define a process to find even numbers and only perform the operation when we need to use the result.

```
void main() {
  // List of numbers
  var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  // Define the process to find even numbers (lazy evaluation)
  var evenNumbers = numbers.where((number) => number.isEven);

  // At this point, no computation is done yet
  print('Even numbers (lazy): $evenNumbers');

  // Computation happens here, when we actually iterate over the result
  for (var number in evenNumbers) {
    print(number);
  }
}
```

Key Points:

1. No Immediate Action: When you define a process using lazy evaluation, no computation or action is taken right away. The process is just set up.
2. On-Demand Computation: The actual computation or action happens only when you need the result. In our stamp analogy, the friend looks for blue stamps only when you ask to see one.
3. Efficiency: This approach can be more efficient because it avoids doing unnecessary work. If you only ask to see the first two blue stamps, your friend doesn't waste time finding all the blue stamps in the collection.

5: Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and methods. Dart is an object-oriented language, and understanding OOP is crucial for building complex and scalable applications in Dart.

Key Concepts of OOP

1. **Classes and Objects**
2. **Constructors**
3. **Instance Variables and Methods**
4. **Inheritance**
5. **Polymorphism**
6. **Abstract Classes and Interfaces**
7. **Encapsulation**

1. Classes and Objects

A class is a blueprint for creating objects. An object is an instance of a class.

Defining a Class

```
class Person {
  String name;
  int age;

  // Constructor
  Person(this.name, this.age);

  // Method
  void introduce() {
    print('Hi, I am $name and I am $age years old.');
  }
}

void main() {
  // Creating an object
  var person = Person('Alice', 30);
  person.introduce(); // Output: Hi, I am Alice and I am 30 years old.
}
```

2. Constructors

Constructors are special methods that are called when an object is instantiated. They initialize the object's properties.

Default Constructor

```
class Person {
  String name;
  int age;

  // Default constructor
  Person(this.name, this.age);
}

void main() {
  var person = Person('Bob', 25);
  print('Name: ${person.name}, Age: ${person.age}');
}
```

Named Constructors

```
class Person {
  String name;
  int age;

  // Named constructor
  Person.named(this.name, this.age);

  // Another named constructor
  Person.anonymous() {
    name = 'Anonymous';
    age = 0;
  }
}

void main() {
  var person1 = Person.named('Charlie', 40);
  var person2 = Person.anonymous();

  print('Name: ${person1.name}, Age: ${person1.age}');
  print('Name: ${person2.name}, Age: ${person2.age}');
}
```

3. Instance Variables and Methods

Instance variables (or fields) hold data for an object, and instance methods define behavior for the object.

```
class Car {
  String model;
  int year;

  Car(this.model, this.year);

  void displayInfo() {
    print('Model: $model, Year: $year');
  }
}

void main() {
  var car = Car('Toyota', 2020);
  car.displayInfo(); // Output: Model: Toyota, Year: 2020
}
```

4. Inheritance

Inheritance allows a class to inherit properties and methods from another class.

Base Class

```
class Animal {
  void makeSound() {
    print('Animal makes a sound');
  }
}
```

Subclass

```
class Dog extends Animal {
  @override
  void makeSound() {
    print('Dog barks');
  }
}

void main() {
  var dog = Dog();
  dog.makeSound(); // Output: Dog barks
}
```

5. Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon.

```
class Shape {
  void draw() {
    print('Drawing a shape');
  }
}

class Circle extends Shape {
  @override
  void draw() {
    print('Drawing a circle');
  }
}

class Square extends Shape {
  @override
  void draw() {
    print('Drawing a square');
  }
}

void main() {
  Shape shape;

  shape = Circle();
  shape.draw(); // Output: Drawing a circle

  shape = Square();
  shape.draw(); // Output: Drawing a square
}
```

6. Abstract Classes and Interfaces

Abstract classes cannot be instantiated and are used to define common behavior that other classes can implement.

Abstract Class

```dart
abstract class Animal {
  void makeSound();
}

class Cat extends Animal {
  @override
  void makeSound() {
    print('Cat meows');
  }
}

void main() {
  var cat = Cat();
  cat.makeSound(); // Output: Cat meows
}
```

Interface

In Dart, all classes implicitly define an interface. You can implement multiple interfaces by using the implements keyword.

```
class Printer {
  void printDocument() {
    print('Printing document');
  }
}

class Scanner {
  void scanDocument() {
    print('Scanning document');
  }
}

class AllInOnePrinter implements Printer, Scanner {
  @override
  void printDocument() {
    print('All-in-One Printer printing document');
  }

  @override
  void scanDocument() {
    print('All-in-One Printer scanning document');
  }
}

void main() {
  var device = AllInOnePrinter();
  device.printDocument(); // Output: All-in-One Printer printing document
  device.scanDocument(); // Output: All-in-One Printer scanning document
}
```

7. Encapsulation

Encapsulation is the bundling of data and methods that operate on the data within one unit, e.g., a class. It restricts direct access to some of the object's components.

Private Variables and Getters/Setters

```dart
class BankAccount {
  String _accountNumber; // Private variable
  double _balance;

  BankAccount(this._accountNumber, this._balance);

  // Getter
  double get balance => _balance;

  // Setter
  set deposit(double amount) {
    _balance += amount;
  }
}

void main() {
  var account = BankAccount('123456', 1000.0);

  account.deposit = 500.0;
  print('Balance: \$${account.balance}'); // Output: Balance: $1500.0
}
```

8. What are Mixins?

In Dart, a mixin is a way to reuse code from multiple class hierarchies. It's a special kind of class that doesn't have its own constructor and can't be instantiated directly. Instead, its purpose is to be "mixed in" with other classes, providing them with additional methods, variables, and even interfaces.

```dart
mixin Logger {
  void log(String message) {
    print('Log: $message');
  }
}

class User with Logger {
  String name;

  User(this.name);

  void greet() {
    log('Hello, $name!');
  }
}

void main() {
  var user = User('Alice');
  user.greet(); // Output: Log: Hello, Alice!
}
```

- The mixin keyword is used to define a mixin.
- The with keyword is used to include a mixin within a class.

6. Exception Handling

Try-Catch

The try-catch block is used to gracefully handle exceptions that might occur during the execution of your code.

```dart
try {
  // Code that might throw an exception
  int result = 10 ~/ 0; // Integer division by zero
} catch (e) {
  // Code to handle the exception
  print("An error occurred: $e");
}
```

- **try: Contains the code that you suspect might throw an exception.**

- **catch (e): Catches the exception and allows you to handle it. The e variable holds the exception object.**
- **You can have multiple catch blocks to handle different types of exceptions.**

Throwing Exceptions

You can explicitly throw exceptions using the throw keyword.

```
void checkAge(int age) {
  if (age < 0) {
    throw Exception("Age cannot be negative");
  }
}

try {
  checkAge(-5);
} catch (e) {
  print(e);
}
```

- **throw: Throws an exception object. You can create your own custom exception classes if needed.**

Finally

The finally block is used to execute code regardless of whether an exception occurred or not.

```
try {
  // Code that might throw an exception
} catch (e) {
  // Handle the exception
} finally {
  // Code that will always execute
  print("This will always be printed.");
}
```

- **finally: Contains code that will be executed whether an exception is thrown or not. It's often used for cleanup tasks like closing files or releasing resources.**

7. Asynchronous Programming

Futures

A Future represents a potential value or error that will be available at some time in the future. It's a way to handle operations that might take some time to complete, like network requests or file I/O, without blocking the main execution thread.

- **Creating a Future**

```
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2)); // Simulate a delay
  return "Data fetched!";
}
```

- **Using async and await**

```
void main() async {
  String data = await fetchData();
  print(data); // Output: Data fetched! (after 2 seconds)
}
```

- **async: Marks a function as asynchronous, allowing it to use await.**
- **await: Pauses the execution of the function until the Future completes and returns its value.**

Streams

A Stream is a sequence of asynchronous events. It's useful for handling data that arrives over time, like user input, sensor data, or real-time updates from a server.

- **Creating a Stream**

```
Stream<int> countStream() async* {
  for (int i = 1; i <= 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}
```

- **Listening to a Stream**

```dart
void main() {
  countStream().listen((data) {
    print(data); // Output: 1 (after 1s), 2 (after 2s), ... 5 (after 5s)
  });
}
```

- **async*: Marks a function as an asynchronous generator, allowing it to yield values to a stream.**
- **yield: Adds a value to the stream.**
- **listen: Subscribes to a stream to receive its events.**

8. Null Safety

Understanding Null Safety

Null safety is a feature introduced in Dart to help prevent null pointer exceptions at runtime. It ensures that variables can't hold null values unless you explicitly allow them to.

- **Non-nullable types: By default, variables are non-nullable. This means they must be assigned a non-null value at the time of declaration or later in the code.**

```dart
String name = "Alice"; // Valid
int? age;              // Nullable (can hold null)
age = 30;              // Valid
age = null;            // Valid because it's nullable
```

- **late keyword: The late keyword allows you to declare a non-nullable variable without initializing it immediately. You promise to initialize it before it's used.**

```dart
late String description;

void main() {
  description = "This is a late variable";
  print(description);
}
```

Null-aware operators:

- **?. (null-conditional operator): Safely accesses a property or method of an object that might be null.**

```
String? middleName;
String fullName = "Alice $middleName Smith"; // Error: middleName mig

String fullName = "Alice ${middleName?.toUpperCase()} Smith"; // Outp
```

- **?? (null-coalescing operator): Provides a default value if an expression is null.**

```
String? greeting;
String message = greeting ?? "Hello!"; // Output: "Hello!"
```

- **??= (null-aware assignment operator): Assigns a value to a variable only if it's currently null.**

```
String? welcomeMessage;
welcomeMessage ??= "Welcome!";
print(welcomeMessage); // Output: "Welcome!"
```

A Deeper Dive into Flutter Project Structure

1. lib Directory

- **main.dart:**
    - The entry point of your Flutter app.
    - Typically creates a MaterialApp or CupertinoApp widget to define the app's theme and routing.
    - Initializes any necessary state management providers.
- **widgets.dart:**
    - A common location for custom widgets that can be reused throughout your app.
    - Custom widgets encapsulate UI elements and their associated logic, making your code more modular and maintainable.
- **pages.dart:**
    - Contains definitions for different screens or pages within your app.
    - Each page is typically represented by a StatefulWidget or StatelessWidget.
- **providers.dart:**
    - If using state management solutions like Provider or Riverpod, this file houses your state providers.
    - Providers manage the application state and make it accessible to different parts of your app.

2. test Directory

- **Unit tests:**
    - Test individual functions or methods in isolation.
    - Verify that your code behaves as expected under different conditions.
- **Widget tests:**
    - Test the rendering and behavior of your custom widgets.
    - Ensure that your UI components are displayed correctly and respond to user interactions.

3. android and ios Directories

- **Platform-specific configuration:**
    - AndroidManifest.xml (Android): Defines app permissions, activities, and services.

- Info.plist (iOS): Specifies app metadata, icons, and capabilities.
- build.gradle (Android): Configures build settings, dependencies, and signing.
- Podfile (iOS): Manages external dependencies using CocoaPods.

4. pubspec.yaml

- **Dependencies:**
  - Lists external packages used by your app, including Flutter packages and third-party libraries.
  - Specifies versions to ensure compatibility and avoid conflicts.
- **Assets:**
  - Declares static assets like images, fonts, and audio files.
  - Allows Flutter to include these assets in your app's bundle.

Additional Considerations:

- **Code organization:**
  - Consider using a consistent naming convention for files, classes, and variables.
  - Group related files together to improve code readability.
- **State management:**
  - Choose a suitable state management solution based on your app's complexity.
  - Getx, Provider, Riverpod, BLoC, and Redux are popular options.
- **Testing:**
  - Write comprehensive tests to ensure code quality and prevent regressions.
  - Use testing frameworks like Flutter's built-in testing tools or third-party libraries.
- **Performance optimization:**
  - Profile your app to identify performance bottlenecks.
  - Optimize code, images, and layouts for better performance.
- **Internationalization:**
  - Use localization to support multiple languages.
  - Store translations in .arb files and use the intl package.

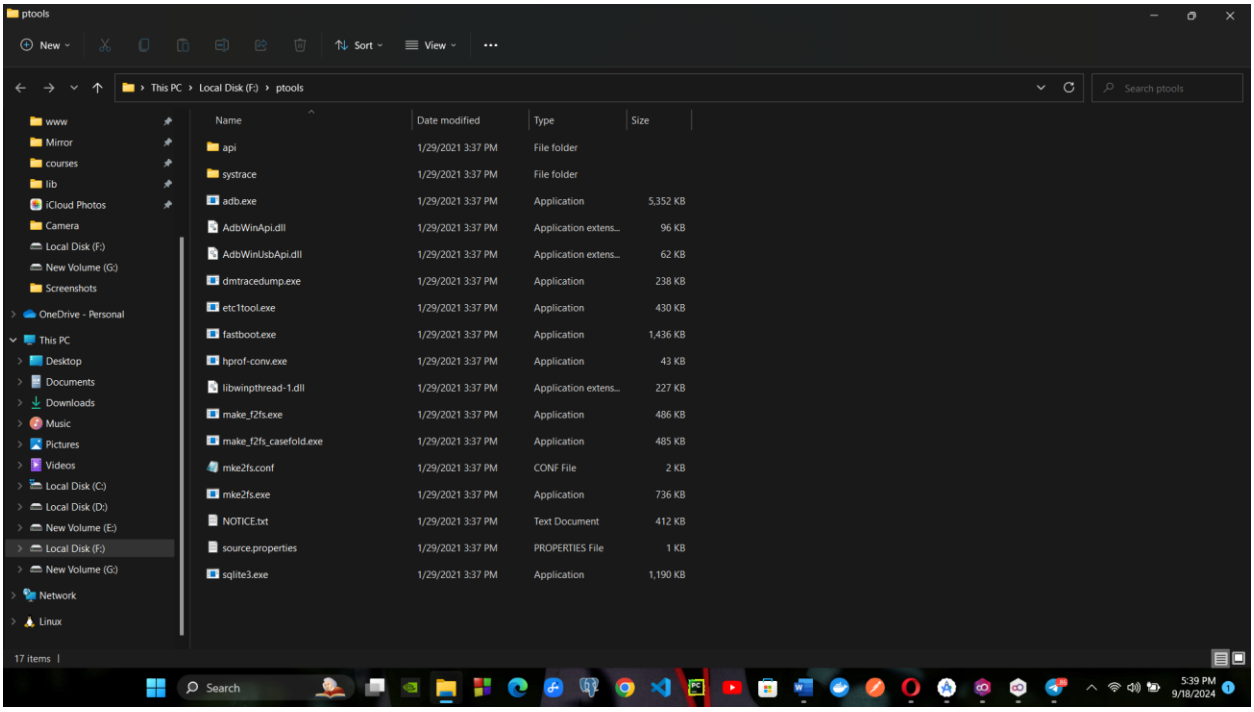
By following these guidelines and adapting the structure to your specific project needs, you can create a well-organized and maintainable Flutter application.

Connecting Real Device to Android Studio

1. Download Platform-tools

https://developer.android.com/tools/releases/platform-tools#:~:text=Download%20SDK%20Platform,Tools%20for%20Linux

2. Open platform-tools folder



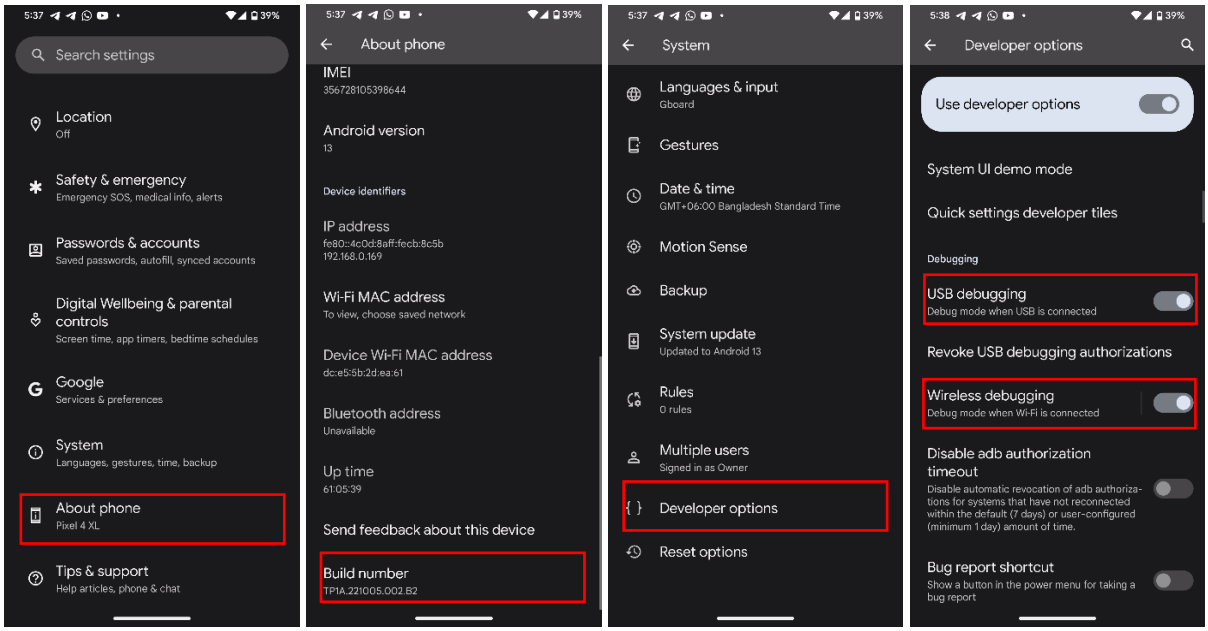
3. Set up your phone

Go to settings and open “About Phone”

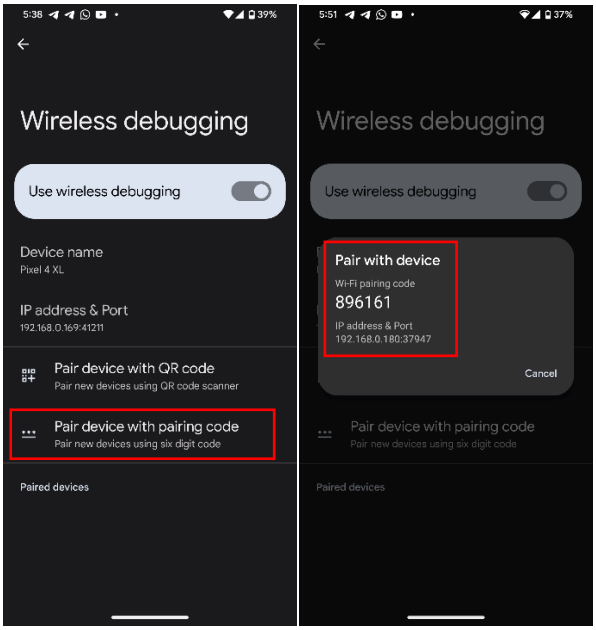
Keep tapping on “Build Number”

You’ll find “Developer options” under System

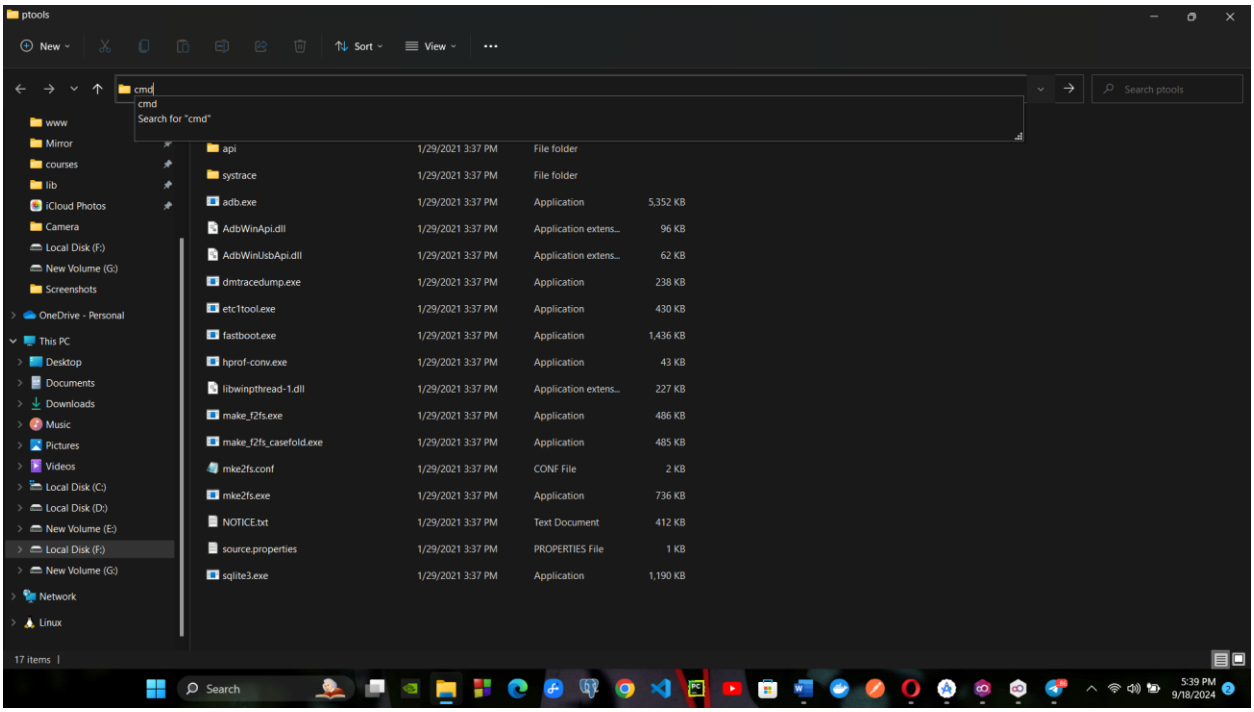
Scroll down and enable USB debugging and wireless debugging



Click on "pair with pairing code" **Take note of these values**

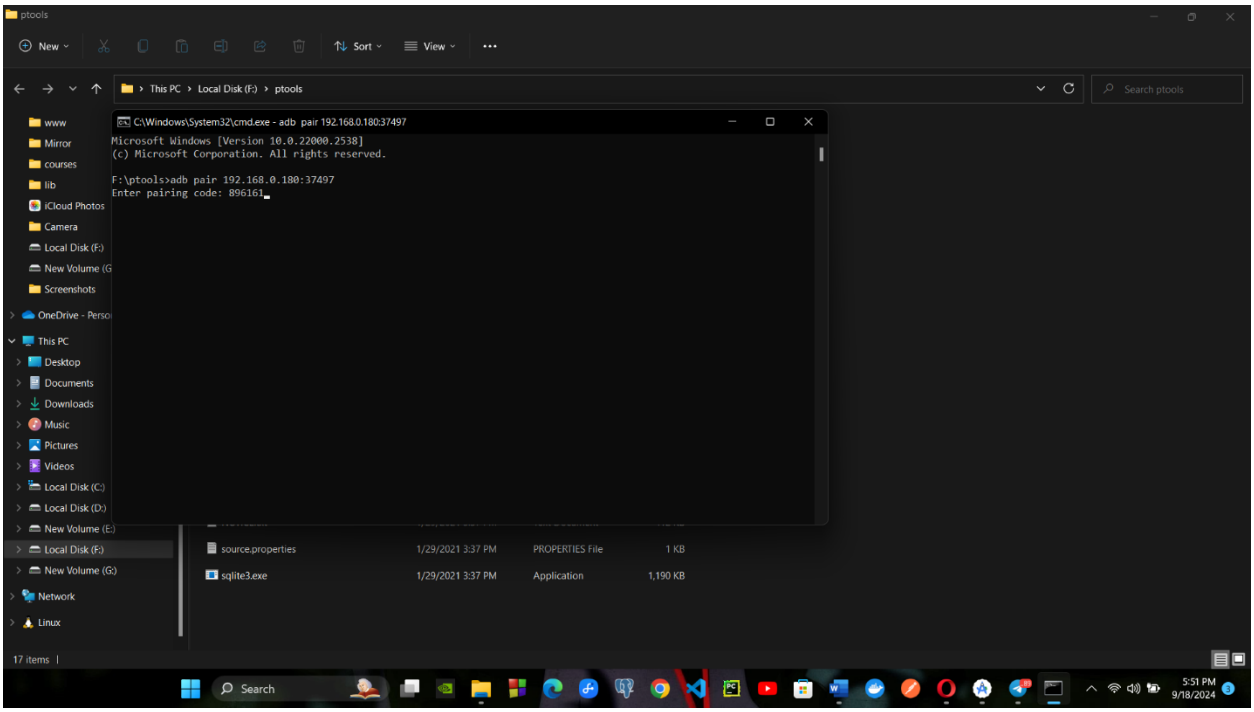


4. In platform-tools, type “cmd” in the address bar and hit enter



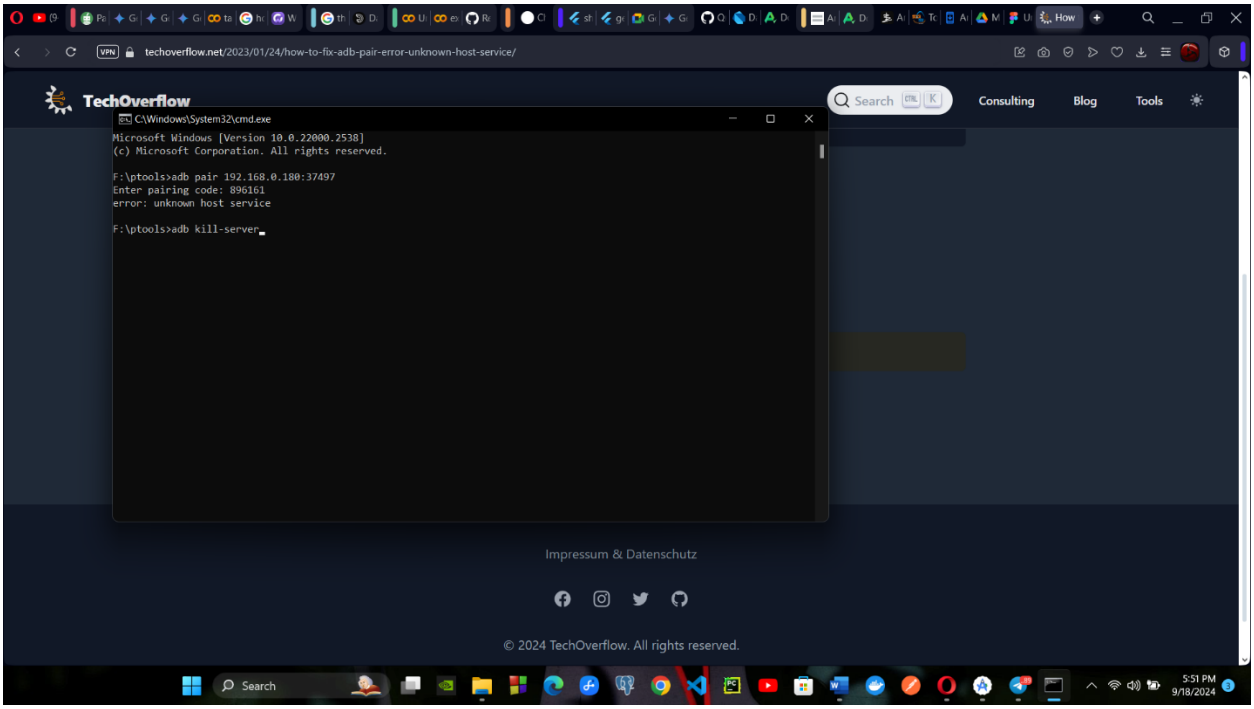
5. In Command Prompt, do the following

Write “adb pair your_ip:port” for example: adb pair 192.168.0.180:37497. You will get this info from the previous steps done in your phone. Check the images above. Hit enter and then it will ask for the pairing code. Enter pairing code from previous step. For example: 896161. Hit enter again.

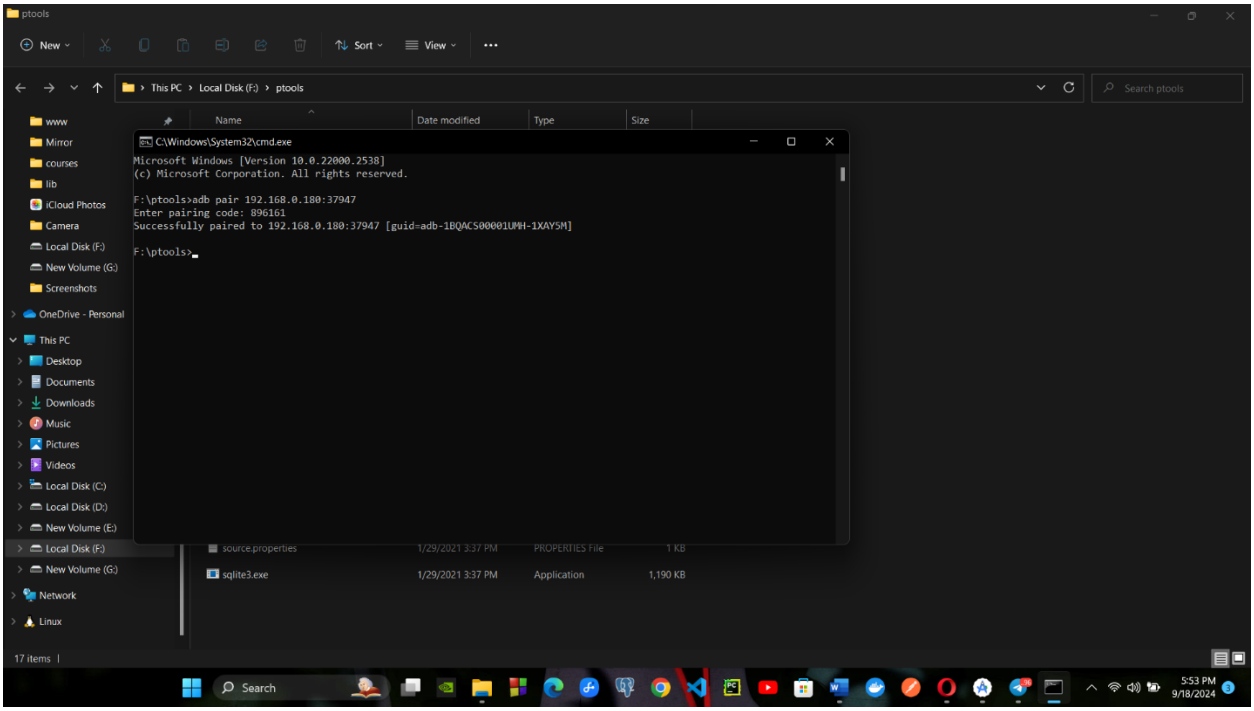


6. Fixing error (if encountered)

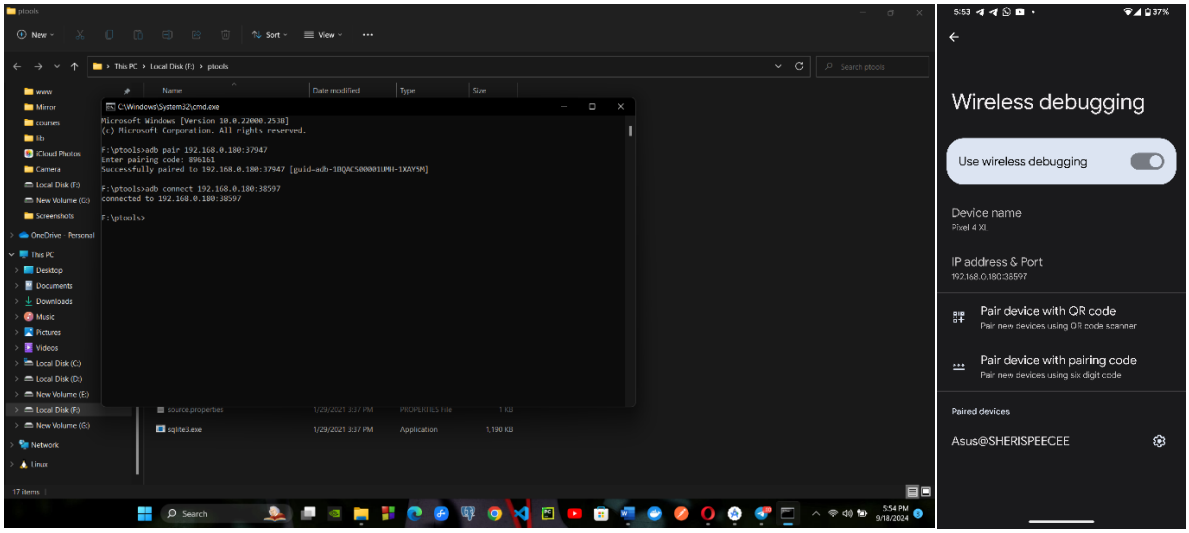
If you encounter the error “unknown host service”, just type “adb kill-server” and hit enter. Then close the cmd prompt, reopen it by typing “cmd” in the address bar of platform-tools. Then redo the pairing process. You will see it getting successfully paired.



A Successful Pairing

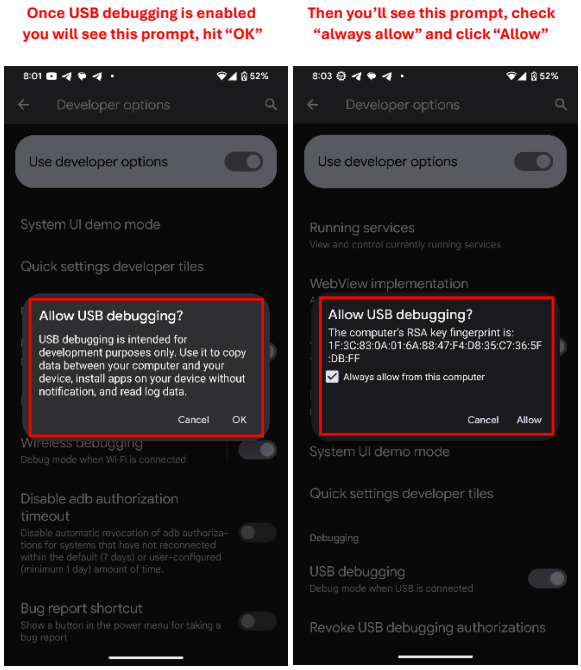


Once the pairing is done, connect the device using the following lines



And boom your device is connected over WiFi.

7. Connecting Over USB



Boom your device is connected via USB.

AVD or Android Virtual Device

Please download and use Genymotion instead of the one that comes with Android Studio.

Link

Flutter Hot Reload and Hot Restart: A Detailed Guide

Understanding Hot Reload and Hot Restart

Hot Reload and **Hot Restart** are powerful features in Flutter that allow developers to quickly iterate on their code and see changes reflected in the running app without having to restart the entire application.

Hot Reload

- **Purpose:** Updates only the modified code without restarting the entire app.
- **Process:**
    - Flutter analyzes the modified code and injects the changes into the running app.

    - The app's state is preserved, allowing you to see the effects of changes immediately.
- **Ideal for:** Making small changes and seeing the results quickly.

Hot Restart

- **Purpose:** Restarts the entire app, including state and dependencies.
- **Process:**
    - The app is terminated and restarted with the latest code changes.
    - Any state or data stored in the app is lost.
- **Ideal for:** Fixing state-related issues or when hot reload doesn't work as expected.

Using Hot Reload and Hot Restart

Android Studio Code Integration:

- **Hot Reload:** Press **Ctrl+S** (Windows/Linux) or **Cmd+S** (macOS) to save changes, and Flutter will automatically hot reload.
- **Hot Restart:** Right-click on the running app in the Debugger panel and select **Restart**.

Key Differences

Feature	Hot Reload	Hot Restart
Speed	Faster	Slower
State Preservation	Preserves state	Loses state
Use Cases	Small changes, rapid iteration	Fixing state issues, when hot reload fails

Best Practices

- **Use Hot Reload for most changes.** It's faster and preserves state, making it ideal for iterative development.
- **Use Hot Restart when necessary.** If you encounter state-related issues or if hot reload doesn't work as expected, a hot restart can help resolve the problem.
- **Be aware of state limitations.** Hot reload may not always preserve state perfectly, especially for complex state management scenarios.

- **Consider using Flutter's built-in debugger.** It can help you identify and fix issues more efficiently.

Mirroring Screen

Use scrcpy (an open source project) to mirror your phone’s screen.

Link