12

Writing Platform-Native Code

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to use platform channels to send and receive messages from the Flutter app to iOS and Android to access specific API functionality
- How to write native platform code in iOS Swift and Android Kotlin to access device information
- How to use `MethodChannel` to send messages from the Flutter app (on the client side)
- How to use `FlutterMethodChannel` on iOS and `MethodChannel` on Android to receive calls and send back results (on the host side)

Platform channels give you the ability to use native features such as accessing the device information, GPS location, local notifications, local file system, sharing, and many more. In the "External Packages" section of Chapter 2, "Creating a Hello World App," you learned how to use third-party packages to add functionality to your apps. In this chapter, instead of relying on third-party packages, you'll learn how to add custom functionality to your apps by using platform channels and writing the API code yourself. You'll build an app that asks the iOS and Android platforms to return the device information.

UNDERSTANDING PLATFORM CHANNELS

When you need to access platform-specific APIs for iOS and Android, you use platform channels to send and receive messages. The Flutter app is the client, and the platform-native code for iOS and Android is the host. If needed, it is also possible to have the platform-native code to act as a client to call methods written in the Flutter app dart code.

The messages between the client and host are asynchronous, making sure that the UI remains responsive and not blocked. In Chapter 3, "Learning Dart Basics," you learned that `async` functions perform time-consuming operations without waiting for those operations to complete.

Beginning Flutter®: A Hands On Guide To App Development, First Edition.
Marco L. Napoli.
© 2020 John Wiley & Sons, Inc. Published 2020 by John Wiley & Sons, Inc.

For the client side (Flutter app), you use the `MethodChannel` from an `async` method to send messages that contain the method call to be executed by the host side (iOS and Android). Once the host sends the response back, you can update the UI to display the information received.

For the host side, you use the `FlutterMethodChannel` on iOS and the `MethodChannel` on Android. Once the client call is received by the host, the native platform code executes the called method and then sends back the result (Figure 12.1).

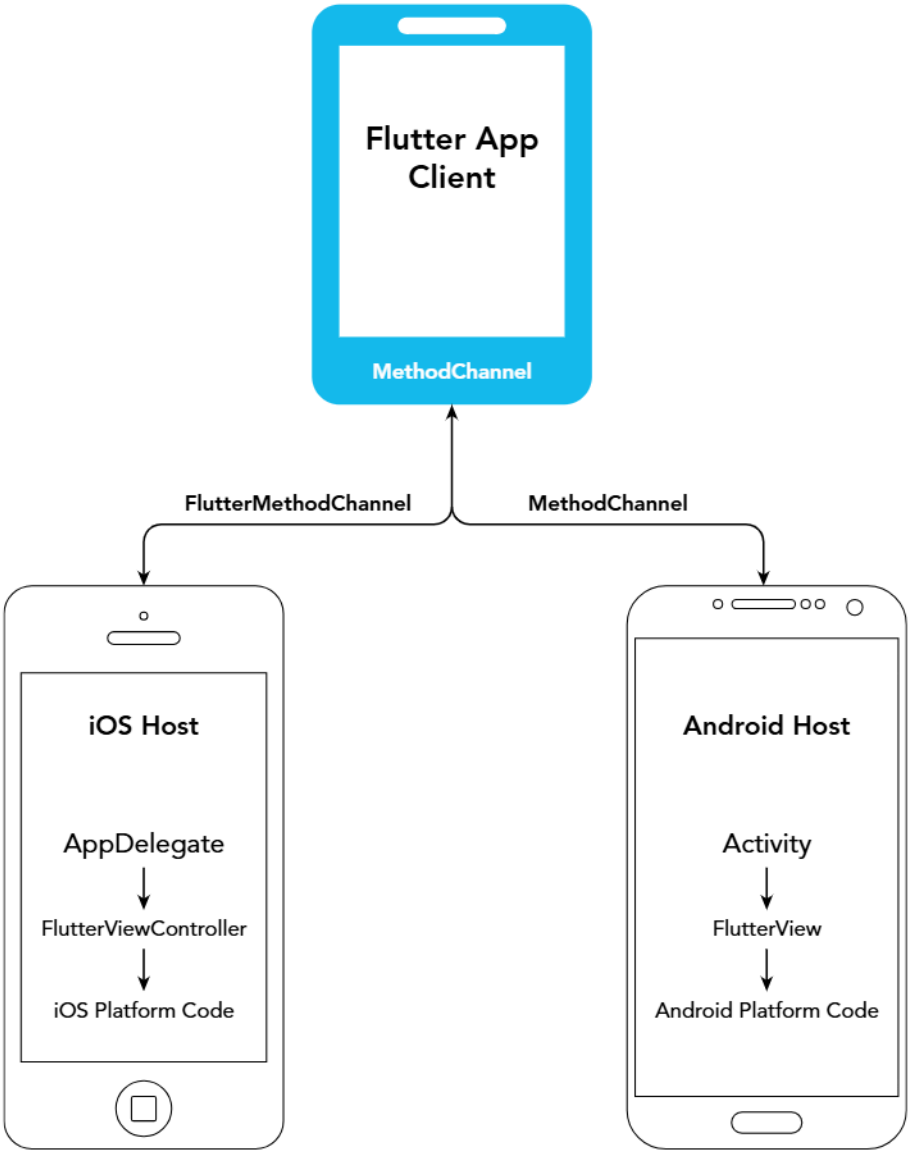


FIGURE 12.1: Platform channel messages

IMPLEMENTING THE CLIENT PLATFORM CHANNEL APP

To start communication from the Flutter client app to the iOS and Android platforms, you use the `MethodChannel`. A `MethodChannel` uses asynchronous method calls, and the channel requires a unique name. The channel name needs to be the same for the client as for the iOS and Android host. I suggest when you're creating a unique name for the channel that you use the app name, a domain prefix, and a descriptive name for the task such as `platformchannel.companyname.com/deviceinfo`.

```
// Name template
appname.domain.com/taskname
// Channel name
platformchannel.companyname.com/deviceinfo
```

At first, it seems that you are going overboard naming the channel, so why is it important for the name to be unique? If you have multiple named channels and they share the same name, they will cause conflicts with each other's messages.

To implement a channel, you create the `MethodChannel` through the default constructor by passing the unique channel name. The default constructor takes two arguments: the first is the channel name, and the second (which is optional) declares the default `MethodCodec`. The `MethodCodec` is the `StandardMethodCodec`, which uses Flutter's standard binary encoding; this means the serialization of data sent between the client and the host is automatically handled. Since you know the name of the channel at compile time and it will not change, you create the `MethodChannel` to a `static const` variable. Make sure you use the `static` keyword, or you will receive the error "Only static fields can be declared as const."

```
static const platform = const
MethodChannel('platformchannel.companyname.com/deviceinfo');
```

Table 12.1 displays the supported value types for Dart, iOS, and Android.

TABLE 12.1: StandardMessageCodec-Supported Value Types

DART	iOS	ANDROID
`null`	`nil`	`null`
`bool`	`NSNumber numberWithBool:`	`java.lang.Boolean`
`int`	`NSNumber numberWithInt:`	`java.lang.Integer`
`int` (bigger than 32 bits)	`NSNumber numberWithLong:`	`java.lang.Long`
`double`	`NSNumber numberWithDouble:`	`java.lang.Double`
`String`	`NSString`	`java.lang.String`
`Uint8List`	`FlutterStandardTypedData typedDataWithBytes:`	`byte[]`

continues

TABLE 12.1: *(continued)*

DART	iOS	ANDROID
`Int32List`	`FlutterStandardTypedData typedDataWithInt32:`	`int[]`
`Int64List`	`FlutterStandardTypedData typedDataWithInt64:`	`long[]`
`Float64List`	`FlutterStandardTypedData typedDataWithFloat64:`	`double[]`
`List`	`NSArray`	`java.util.ArrayList`
`Map`	`NSDictionary`	`java.util.HashMap`

To call and specify which method to execute on the iOS and Android host, you use the `invokeMethod` constructor to pass the method name as a `String`. The `invokeMethod` is called from inside a `Future` method since the call is `asynchronous`.

```
String deviceInfo = await platform.invokeMethod('getDeviceInfo');
```

Once the client and the iOS and Android platform channels are implemented, the Flutter client side of the app will display the appropriate device information depending on the device (Figure 12.2).

TRY IT OUT Creating the Client Platform Channel App

In this example, you want to display the running device information such as the manufacturer, device model, name, operating system, and a few other details. The Flutter app client is written in Dart (as usual) and implements the `MethodChannel` to initiate a call to the iOS and Android host.

- The iOS host is written in Swift to access the platform API call to the `UIDevice` to query the device information and uses the `FlutterMethodChannel` to receive and return the requested information.
- The Android host is written in Kotlin to access the platform API call to the `Build` to query the device information and uses the `MethodChannel` to receive and return the requested information.

This app is divided into three different "Try It Out" exercises. In this first one, you'll concentrate on creating the client-side request. In the second one, "Creating the iOS Host Platform Channel," you'll build the iOS Host Platform Channel, and in the third exercise, "Creating the Android Host Platform Channel," you'll build the Android Host Platform Channel. If you run the app after this first section is completed, you will receive the error "Failed to get device info" since you have not written the iOS and Android host code yet. Note that the iOS and Android projects are independent of each other, and you can target both or only one, and you will receive the error on the platform that you run.

1. Create a new Flutter project and name it `ch12_platform_channel`; as always, you can follow the instructions in Chapter 4, "Creating a Starter Project Template." For this project, you need to create only the `pages` folder.

2. Open the `home.dart` file and add to the `body` a `SafeArea` with the `child` as a `ListTile`.

```
body: SafeArea(
  child: ListTile(),
),
```

3. After the `class _HomeState extends State<Home>` and before `@override`, add the `static const` variable `_methodChannel` initialized by the `MethodChannel` with the name `platformchannel.companyname.com/deviceinfo`.

```
static const _methodChannel = const MethodChannel('platformchannel.companyname.
com/deviceinfo');
```

4. Declare the `_deviceInfo String` variable that will receive the device information from the iOS and Android host call.

```
// Get device info
String _deviceInfo = '';
```

5. Create the `_getDeviceInfo() async` call that uses the `_methodChannel.invokeMethod()` to initiate the call to the iOS and Android host. This method is declared as a `Future<void>` and marked as `async`.

```
Future<void> _getDeviceInfo() async {
}
```

6. Create the local `deviceInfo String` variable that receives the device information.

```
String deviceInfo;
```

It's good practice to use the `try-catch` exception handling when calling the `methodChannel .invokeMethod('getDeviceInfo')` just in case the call fails. The `invokeMethod` takes the `getDeviceInfo` method name that needs to be the same one that you declare in both the iOS and Android host code.

```
try {
  deviceInfo = await methodChannel.invokeMethod('getDeviceInfo');
} on PlatformException catch (e) {
  deviceInfo = "Failed to get device info: '${e.message}'.";
}
```

7. Add the `setState()` method that populates the `_deviceInfo` (available class-wide) variable from the local `deviceInfo` value. In Chapter 3, "Learning Dart Basics," in the "Asynchronous Programming" section, you learned how to use the `Future` object.

```
Future<void> _getDeviceInfo() async {
  String deviceInfo;
  try {
    deviceInfo = await _methodChannel.invokeMethod('getDeviceInfo');
  } on PlatformException catch (e) {
    deviceInfo = "Failed to get device info: '${e.message}'.";
  }
```

```
    setState(() {
      _deviceInfo = deviceInfo;
    });
  }
```

8. Override the `initState()` to call the `_getDeviceInfo()` method. Once the app starts, it makes the `_getDeviceInfo()` call start retrieving device information.

```
@override
void initState() {
  super.initState();
  _getDeviceInfo();
}
```

9. Let's go back to the `body` property and finish the `ListTile` widget to display the device information. For the `title` property, add a `Text` widget to show the Device Info heading, and set the `TextStyle` to `fontSize` 24.0 and `FontWeight.bold`.

10. For the `subtitle` property, add a `Text` widget with the `_deviceInfo` variable that shows the actual device information, and set the `TextStyle` to `fontSize` 18.0 and `FontWeight.bold`.

11. Add a `contentPadding` property to the `ListTile`, and set the `EdgeInsets.all()` to 16.0 to add some nice padding around the information.

```
body: SafeArea(
  child: ListTile(
    title: Text(
      'Device info:',
      style: TextStyle(
        fontSize: 24.0,
        fontWeight: FontWeight.bold,
      ),
    ),
    subtitle: Text(
      _deviceInfo,
      style: TextStyle(
        fontSize: 18.0,
        fontWeight: FontWeight.bold,
      ),
    ),
    contentPadding: EdgeInsets.all(16.0),
  ),
),
```

HOW IT WORKS

In this client section of the app, you created the `MethodChannel` with a unique name to a `static const _methodChannel` variable. The `methodChannel` is used to communicate between the client and the host using the `asynchronous` method call. Using the `_getDeviceInfo() Future<void>` method, the `_methodChannel.invokeMethod('getDeviceInfo')` calls the host to execute the `getDeviceInfo` method in the iOS and Android platforms. Once the data is returned, the `setState()` method populates the `_deviceInfo` variable, and the `ListTile subtitle` is updated with the device information.

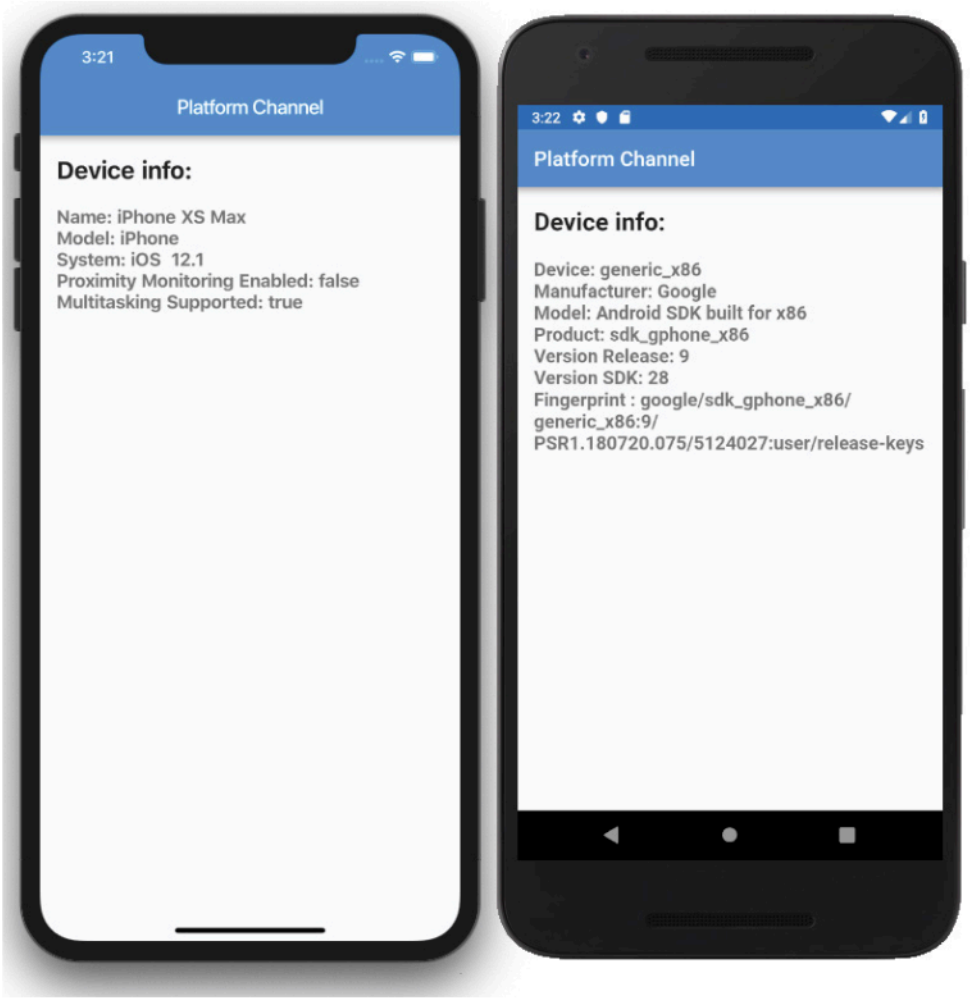


FIGURE 12.2: iOS and Android device information

IMPLEMENTING THE iOS HOST PLATFORM CHANNEL

The host is responsible for listening to incoming messages from the client. Once a message is received, the channel checks for a matching method name, executes the call method, and returns the appropriate result. In iOS, you use the `FlutterMethodChannel` for listening to incoming messages that take two parameters. The first parameter is the same platform channel name—'`platformchannel.companyname.com/deviceinfo`'—as the client. The second is the `FlutterViewController`, which is the main `rootViewController` of an iOS app. The `rootViewController` is the root view controller for the iOS app window that provides the content view of the window.

```
let flutterViewController: FlutterViewController = window?.rootViewController as!
FlutterViewController
let deviceInfoChannel = FlutterMethodChannel(name: "platformchannel.companyname.
com/deviceinfo", binaryMessenger: controller)
```

You then use the `setMethodCallHandler` (`Future` handler) to set up a callback for a matching method name that executes the iOS native platform code. Once completed, it sends back the result to the client.

```
deviceInfoChannel.setMethodCallHandler({
    (call: FlutterMethodCall, result: FlutterResult) -> Void in
    // Check for incoming method call name and return a result
})
```

Both the `FlutterMethodChannel` and the `setMethodCallHandler` will be placed in the `didFinishLaunchingWithOptions` method of the iOS app `AppDelegate.swift` file. The `didFinishLaunchingWithOptions` is responsible for notifying the app delegate that the app launch process is almost done.

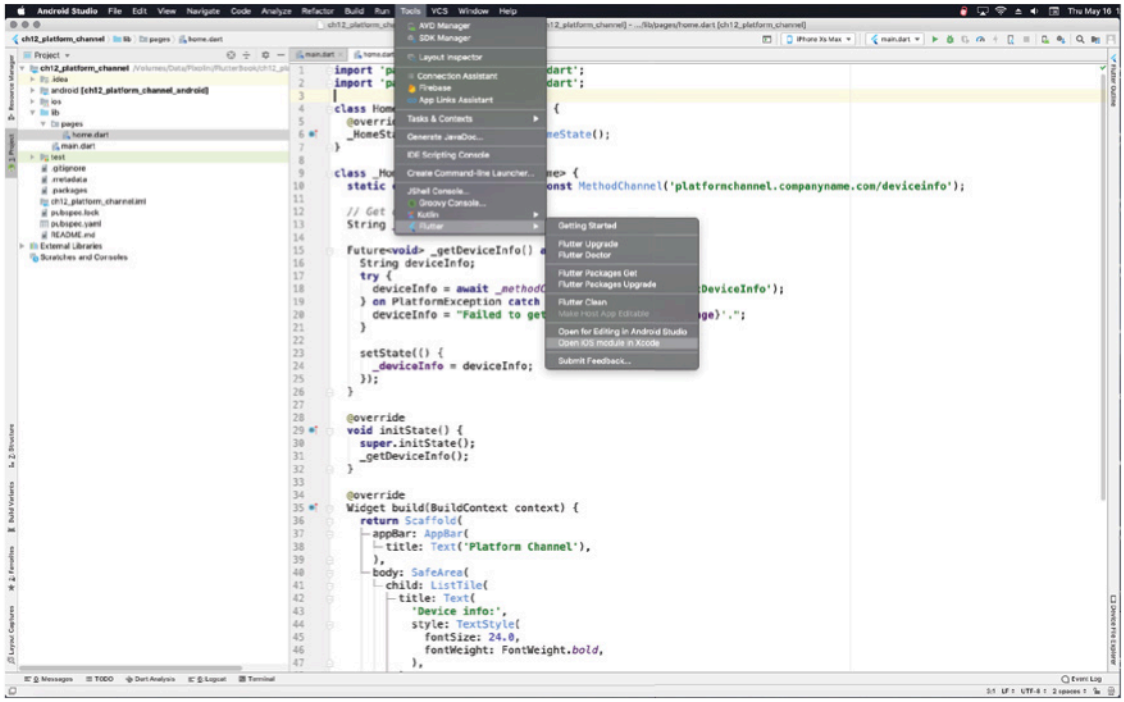
```
override func application(
  _ application: UIApplication,
  didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?
) -> Bool {
  // Code
}
```

TRY IT OUT Creating the iOS Host Platform Channel

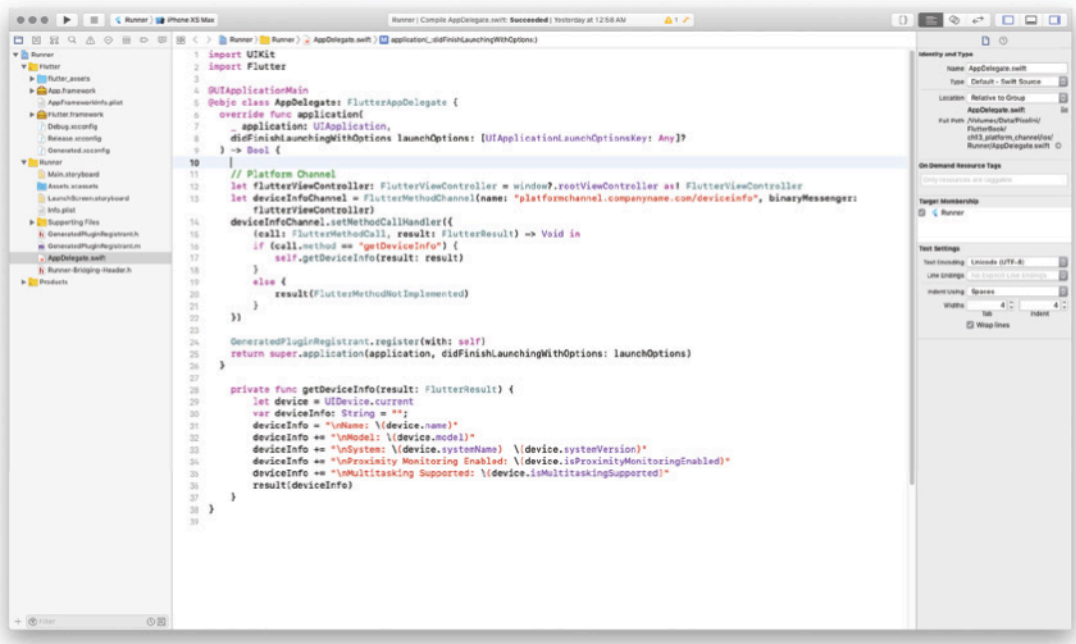
In this example, you want to retrieve the running device information such as the manufacturer, device model, name, operating system, and a few other details. The iOS host is written in Swift to access the platform API call to the `UIDevice` object to query the device information and uses the `FlutterMethodChannel` to receive communication from the client. Once the message is received, the `setMethodCallHandler` handles the request and returns the result.

In this section, you'll open Xcode and edit the native iOS Swift code.

1. If you closed the Flutter project `ch12_platform_channel`, reopen it. Click the Android Studio Tools menu bar and select Flutter ➪ Open iOS module in Xcode. Note that this menu item selection opens the Xcode app with the iOS project, but you can also open the iOS project manually by double-clicking the `Runner.xcworkspace` file located in the `ios` folder. A Mac computer with Xcode installed is required to edit the iOS host project.



2. On the navigator area (left side), expand the `Runner` folder by clicking the arrow and then select the `AppDelegate.swift` file.



3. Edit the `didFinishLaunchingWithOptions` method; you'll be adding code before the line `GeneratedPluginRegistrant` call. Declare the `flutterViewController` variable as a `FlutterViewController` and initiate it with the `window?.rootViewController as! FlutterViewController`.

```
    let flutterViewController: FlutterViewController = window?.rootViewController
as! FlutterViewController
```

4. In the next line, declare the `deviceInfoChannel` variable by initiating it with the `FlutterMethodChannel`.

5. For the `FlutterMethodChannel`'s first parameter, `name`, pass the Flutter channel name, the same one declared in the client. The second parameter, `binaryMessenger`, takes the `flutterViewController` variable.

```
    let deviceInfoChannel = FlutterMethodChannel(name: "platformchannel
.companyname.com/deviceinfo", binaryMessenger: flutterViewController)
```

6. Add the `deviceInfoChannel.setMethodCallHandler` to set up a callback that matches the incoming method name of `getDeviceInfo` by using an `if-else` statement. If the `call.method` matches (`==`) the `getDeviceInfo`, then you call the `self.getDeviceInfo(result: result)` method (that you create in the next step) that retrieves the device information.

   If the `call.method` does not match the incoming method name, the `else` statement returns the `result(FlutterMethodNotImplemented)`. The `FlutterMethodNotImplemented` is a constant that responds to the `call` that the method is unknown, or not implemented.

```
    deviceInfoChannel.setMethodCallHandler({
        (call: FlutterMethodCall, result: FlutterResult) -> Void in
        if (call.method == "getDeviceInfo") {
            self.getDeviceInfo(result: result)
        }
        else {
            result(FlutterMethodNotImplemented)
        }
    })
```

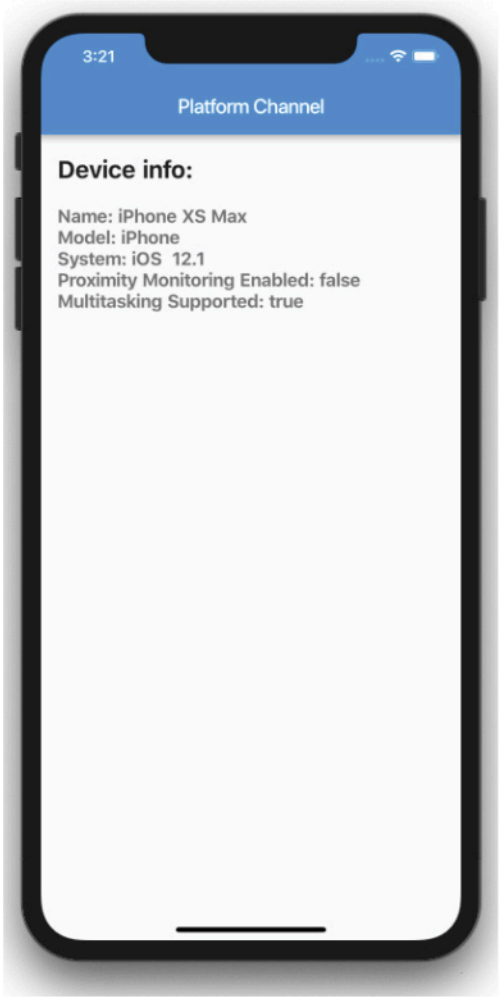
7. After the `didFinishLaunchingWithOptions` method, add the `getDeviceInfo(result: FlutterResult)` method that uses the iOS Swift `UIDevice.current` to query and retrieve the current device information.

8. Declare the `let device` variable by initializing it with the `UIDevice.current`. The `let` keyword is similar to the Dart `final` keyword, telling the compiler the value will not change. Declare the `deviceInfo String` variable and initialize it to an empty string by using double quotes (`""`).

   To format the result to the `deviceInfo` variable, you use the `\n` characters to start a new line for each piece of information. By using string concatenation, you use the `=+` sign to add each line to the `deviceInfo` variable. In Swift, inside a `String`, you use the `\()` character combination to extract the value of the expression inside.

```
    private func getDeviceInfo(result: FlutterResult) {
        let device = UIDevice.current
        var deviceInfo: String = ""
        deviceInfo = "\nName: \(device.name)"
```

```
        deviceInfo += "\nModel: \(device.model)"
        deviceInfo += "\nSystem: \(device.systemName)  \(device.systemVersion)"
        deviceInfo += "\nProximity Monitoring Enabled: \(device.
isProximityMonitoringEnabled)"
        deviceInfo += "\nMultitasking Supported: \(device.isMultitaskingSupported)"
        result(deviceInfo)
    }
```



HOW IT WORKS

In the iOS host section of the app, you are listening for incoming messages by using the `FlutterMethodChannel`. The `FlutterMethodChannel` expects two parameters, the `name` and the `binaryMessenger`. The `name` parameter is the Flutter channel name declared in the client app `platformchannel.companyname.com/deviceinfo`. The `binaryMessenger` parameter is the `FlutterViewController` initiated by the iOS app `window?.rootViewController`.

You set up a callback with the `setMethodCallHandler` to match the incoming method name of `getDeviceInfo` by using an `if-else` statement. If the `call.method` matched the method name, you call the `getDeviceInfo` method to retrieve the device information and return a result. The device information is obtained by querying the `UIDevice.current` object. If the `call.method` does not match the incoming method name, the `else` statement returns the `FlutterMethodNotImplemented`.

IMPLEMENTING THE ANDROID HOST PLATFORM CHANNEL

The host is responsible for listening for incoming messages from the client. Once a message is received, the channel checks for a matching method name, executes the call method, and returns the appropriate result. In Android, you use the `MethodChannel` to listen to incoming messages that take two parameters. The first parameter is the `FlutterView`, which extends the `Activity` of an Android app screen and, by using the `flutterView` variable as the parameter, is the same as calling the `getFlutterView()` (`FlutterView`) method from the `FlutterActivity` class. The second parameter is the same platform channel name `platformchannel.companyname.com/deviceinfo` as the client.

```
private val DEVICE_INFO_CHANNEL = "platformchannel.companyname.com/deviceinfo"
val methodChannel = MethodChannel(flutterView, DEVICE_INFO_CHANNEL)
```

You then use the `setMethodCallHandler` (`Future` handler) to set up a callback for a matching method name that executes the Android native platform code. Once completed, it sends the result to the client.

```
methodChannel.setMethodCallHandler { call, result ->
  // Check for incoming method call name and return a result
}
```

Both the `MethodChannel` and the `setMethodCallHandler` are placed in the `onCreate` method of the Android app's `MainActivity.kt` file. The `onCreate` is called when the activity is first created.

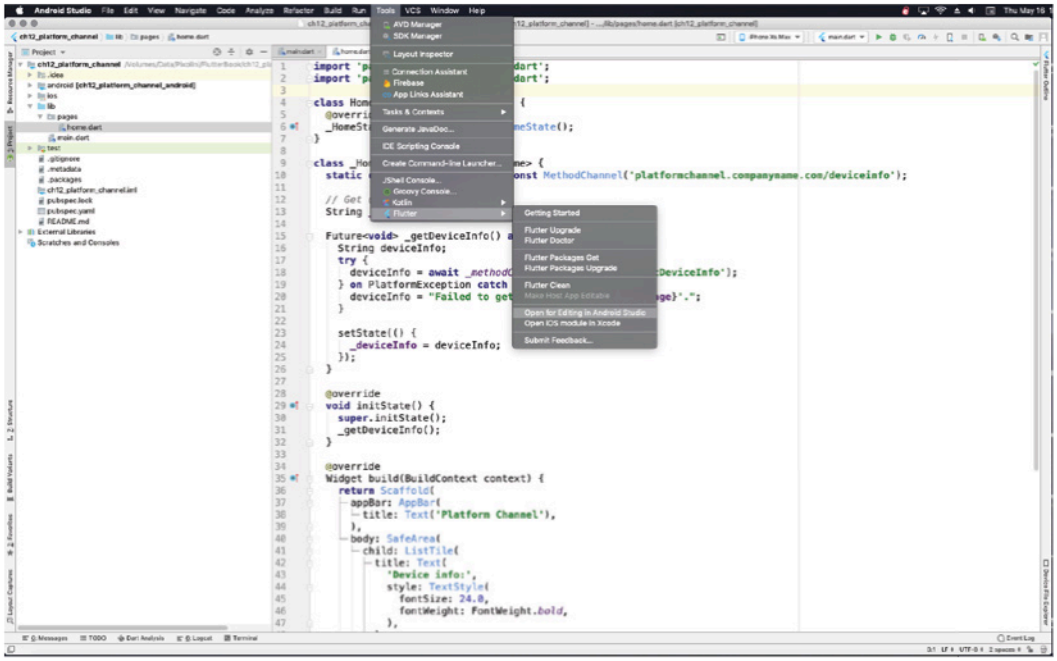
```
override fun onCreate(savedInstanceState: Bundle?) {
  // Code
}
```

TRY IT OUT Creating the Android Host Platform Channel

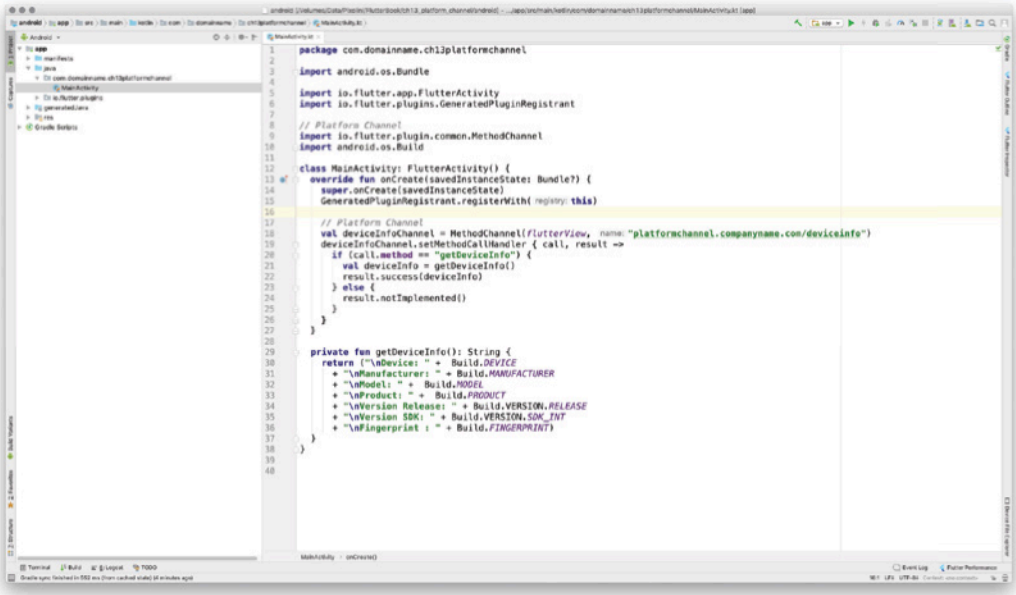
In this example, you want to retrieve the running device information such as the manufacturer, device model, name, operating system, and a few other details. The Android host is written in Kotlin to access the platform API call to the `Build` class to query the device information and uses the `MethodChannel` to receive communication from the client. Once the message is received, the `setMethodCallHandler` handles the request and returns the result.

In this section, you'll open another instance of Android Studio and edit the native Android Kotlin code.

1. If you closed the Flutter project `ch12_platform_channel`, reopen it. Click on the Android Studio Tools menu bar and select Flutter ➪ Open for Editing in Android Studio.



2. On the tool window area (on the left side), expand the `app` folder by clicking the arrow, open the `java` folder, open the `com.domainname.ch12platformchannel` folder, and then select the `MainActivity.kt` file. Note the `domainname` might be different depending on the name that you chose when creating the Flutter project.



3. Add two `import` statements before the `MainActivity` class. The first `import` statement adds support for the `MethodChannel`, and the second adds support to use the `Build` to query for the device information.

```
import io.flutter.plugin.common.MethodChannel
import android.os.Build
```

4. Edit the `onCreate` method, and you'll be adding code after the `GeneratedPluginRegistrant` call. Declare the `deviceInfoChannel` variable by initiating it with the `MethodChannel`.

5. For the `MethodChannel`'s first parameter, `BinaryMessenser`, pass the `flutterView` variable; note that you did not declare this variable. You do not need to declare it since it's the same as calling the `getFlutterView()` method from the `FlutterActivity` class.

6. For the second parameter, pass the Flutter channel name `platformchannel.companyname.com/deviceinfo`, the same one declared in the client.

```
    val deviceInfoChannel = MethodChannel(flutterView, "platformchannel
.companyname.com/deviceinfo")
```

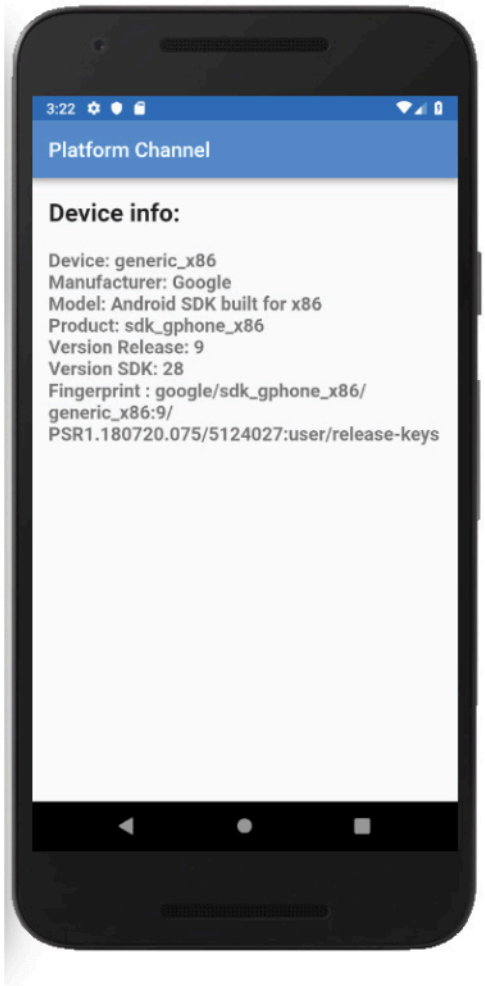
7. Add the `deviceInfoChannel.setMethodCallHandler` to set up a callback that matches the incoming method name of `getDeviceInfo` by using an `if-else` statement. If the `call.method` matches (`==`) the `getDeviceInfo`, then you call the `getDeviceInfo()` method (that you create in the next step) that retrieves the device information, and the result is saved to the `deviceInfo` variable. If the `call.method` does not match the incoming method name, the `else` statement returns the `result.notImplemented()`. The `notImplemented()` method responds to the `call` that the method is unknown, or not implemented.

```
deviceInfoChannel.setMethodCallHandler { call, result ->
  if (call.method == "getDeviceInfo") {
    val deviceInfo = getDeviceInfo()
    result.success(deviceInfo)
  } else {
    result.notImplemented()
  }
}
```

8. After the `onCreate` method, add the `getDeviceInfo(): String` method that uses the Android `Build` to query and retrieve the current device information.

   To format the result, you use the `\n` characters to start a new line for each piece of information. By using string concatenation, you use the `+` sign to add each line and return the formatted result. The entire string concatenation is enclosed in open and close parentheses `("...")`.

```
private fun getDeviceInfo(): String {
  return ("\nDevice: " +  Build.DEVICE
    + "\nManufacturer: " + Build.MANUFACTURER
    + "\nModel: " +  Build.MODEL
    + "\nProduct: " +  Build.PRODUCT
    + "\nVersion Release: " + Build.VERSION.RELEASE
    + "\nVersion SDK: " + Build.VERSION.SDK_INT
    + "\nFingerprint : " + Build.FINGERPRINT)
}
```



HOW IT WORKS

In the Android host section of the app, you are listening for incoming messages by using the `MethodChannel`. The `MethodChannel` expects two parameters, the `binaryMessenger` and the `name`. The `binaryMessenger` parameter is the `flutterView`, which is the `getFlutterView()` method from the `FlutterActivity` class that extends the `Activity` of an Android app screen. The `name` parameter is the Flutter channel name declared in the client app `platformchannel.companyname.com/deviceinfo`.

You set up a callback with the `setMethodCallHandler` to match the incoming method name of `getDeviceInfo` by using an `if-else` statement. If the `call.method` matched the method name, you call the `getDeviceInfo` method to retrieve the device information and return a result. The device information is obtained by querying the `Build` class. If the `call.method` does not match the incoming method name, the `else` statement returns the `result.notImplemented()`.

SUMMARY

In this chapter, you learned how to access and communicate with iOS and Android platform-specific API code by implementing platform channels. Platform channels are a way for the Flutter app (client) to communicate (messages) with iOS and Android (host) to request and receive results specific to the operating system (OS). For the UI to remain responsive and not blocked, the messages between the client and host are asynchronous.

To start communicating from the Flutter app (client), you learned to use the `MethodChannel` that sends messages that contain method calls to be executed by the iOS and Android (host) side. Once the host processes the method requested, it then sends back a response to the client, and you update the UI to display the information. The `MethodChannel` uses a unique name, and you used a combination of the app name, the domain prefix, and the task name like `platformchannel.companyname.com/deviceinfo`. To start the call from the client and specify which method to execute on the host, you learned to use the `invokeMethod` constructor, and it is called from inside a `Future` method since calls are `asynchronous`.

For the iOS and Android host, you learned to use the Flutter `FlutterMethodChannel` on iOS and the `MethodChannel` on Android to start receiving communications from the client. The host is responsible for listening to incoming messages from the client. You used the `setMethodCallHandler` to set up a callback for an incoming matching method name that executes on the native platform-specific API code. Once the method completes, it sends the result to the client.

In the next chapter, you'll learn to use local persistence to save data locally to the device storage area.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Implementing `MethodChannel` (client)	This enables communication from the Flutter app (client) by sending messages that contain method calls to be executed by the iOS and Android (host).
Implementing `invokeMethod` (client)	This initiates (invokes) and specifies which method call to execute on the host side.
Implementing `FlutterMethodChannel` (iOS host) Implementing `MethodChannel` (Android host)	This enables communication from the host to receive method calls to execute platform-specific API code.
Implementing `setMethodCallHandler` (iOS and Android host)	This sets up a callback for incoming matching method names to execute platform-specific API code.