47. Let's trace the execution step-by-step 👇

```
void myFunc() {
  print("A");
  myFuture().then((_) => print("C"));
  print("B");
}
```

We don't see what `myFuture()` does, but since it's a **Future**, we know it runs **asynchronously** — it won't block the next statements.

Execution order:

1. `print("A");` → runs immediately → **A** printed first.
2. `myFuture().then((_) => print("C"));` → schedules a future callback (`print("C")`) to run **later**, after `myFuture()` completes.
3. `print("B");` → runs next → **B** printed second.
4. When `myFuture()` completes, the `.then()` callback executes → **C** printed last.

✅ **Final output (visual result):**

```
A
B
C
```

So:

- "A" and "B" appear immediately (in order),
- "C" appears later, after the asynchronous task finishes.

46

.Why doesn't the number on screen update? b) Show the two key changes to fix it (class + state update line)

```
class MyCounter extends StatelessWidget {
  int counter = 0;
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
```

```
      onPressed: () { counter++; print(counter); },
      child: Text('$counter'),
    );
  }
}
```

Ans:Good question — the reason the **number on screen doesn't update** is because:

✅ `StatelessWidget` **cannot rebuild itself** when data changes.
✅ The `counter` variable is updated, but the UI is never told to rebuild.

🔧 Two Key Changes to Fix It:

1️⃣ Change the class to `StatefulWidget`

A `StatefulWidget` allows Flutter to rebuild the widget when the state changes.

```
class MyCounter extends StatefulWidget {
  @override
  _MyCounterState createState() => _MyCounterState();
}
```

2️⃣ Call `setState()` to update the UI

Wrap your counter increment inside `setState()` so the framework knows to rebuild the widget.

```
class _MyCounterState extends State<MyCounter> {
  int counter = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        setState(() { counter++; }); // ✅ rebuilds the UI
        print(counter);
      },
      child: Text('$counter'),
    );
  }
}
```

✅ Summary:

Problem	Fix
Using `StatelessWidget`	Change to `StatefulWidget`
Direct variable change (`counter++`)	Wrap in `setState(() { counter++; })`

These two changes make the number update on screen properly.

45. In **Flutter**, a **Platform Channel** is the **communication bridge** between **Dart (Flutter code)** and the **native platform code** — that is, **Android (Kotlin/Java)** and **iOS (Swift/Objective-C)**.

🧠 In simple words:

A **Platform Channel** lets your Flutter app talk to the device's native APIs — for example:

- Accessing the camera, Bluetooth, battery info, or sensors
- Using native libraries or SDKs not available in Flutter

⚙️ How It Works (Conceptually)

Flutter and the native side communicate using **asynchronous message passing**.

1. **Flutter side (Dart)** sends a message through a channel.
2. **Native side (Android/iOS)** receives the message and performs the requested action.
3. Native side sends a **response** back to Flutter.

🔗 Channel Types

Type	Direction	Description
MethodChannel	Two-way	Call a native function and get a result back
EventChannel	Native → Flutter	Stream of data from native (e.g., sensor updates)

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

BasicMessageChannel	Both ways	For sending simple string or binary messages

44.Local vs Push notifications—one line each.

Local Notification: Triggered by the app itself on the device (no internet needed).
Push Notification: Sent from a remote server via the internet to the device (needs backend or cloud service).

Local Notification: `flutterLocalNotificationsPlugin.show(id, title, body, details);`
Push Notification: `FirebaseMessaging.onMessage.listen((message) => showNotification(message));`

43.**Purpose:** To let Flutter (Dart) communicate with native platform code (Android/iOS) for accessing device features not available in Flutter.

Example Method Call:

```
final batteryLevel = await
MethodChannel('samples.flutter.dev/battery').invokeMethod('getBatteryLevel');
```

42.
Forms & validators: purpose + tiny validation example.

Purpose: To collect user input in Flutter and ensure it meets certain rules (like required fields, email format, length).

Tiny Validation Example:

```
TextFormField(
  validator: (value) => value!.isEmpty ? 'Cannot be empty' : null,
)
```

41.One thing DevTools helps inspect (e.g., memory/frames).

Flutter DevTools helps inspect **widget tree layout and rebuilds**.

40.
Two Flutter test types (name them).

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

1. **Unit Test** – tests individual functions, methods, or classes.
2. **Widget Test** – tests UI widgets and their interaction.

39.One difference between debug and release builds.

Debug Build: Includes debugging info and asserts, slower, used during development.
Release Build: Optimized, no debugging info, used for production deployment.

38.**Where to declare Android permissions:** In the `AndroidManifest.xml` file.

When to request runtime permission: For **dangerous permissions** (e.g., camera, location, storage) on **Android 6.0+**, request **at runtime** before accessing the feature.

37.benefit of the Repository Pattern. [2 mark] with eg

Benefit: Separates **data access logic** from **business logic**, making code **cleaner, reusable, and easier to test**.

Example:

```
class UserRepository {

  Future<User> getUser(int id) => ApiService().fetchUser(id);

}
```

Here, the UI calls `UserRepository.getUser()` instead of directly calling `ApiService`.

36.Outline **MVVM** (Model/View/ViewModel) in one line each.

- **Model:** Manages app data and business logic.
- **View:** Displays the UI and observes the ViewModel.
- **ViewModel:** Connects Model and View, exposing data and handling UI logic.

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

35.Firebase **Security Rules**—purpose.

Purpose: To **control read and write access** to Firebase databases or storage, ensuring only authorized users can access or modify data.

34.REST vs RPC—one core difference. with full form

REST (Representational State Transfer): Operates on **resources** using standard HTTP methods (GET, POST, PUT, DELETE).

RPC (Remote Procedure Call): Calls **specific functions or procedures** on the server directly.

33.One security consideration for local storage (prefs/SQLite).

Security Consideration: Data stored locally can be accessed by anyone with device access, so sensitive data should be **encrypted**.

32.How to show Cupertino look in a Material app on iOS (high-level approach).

High-level Approach: Wrap your app or widgets in **`CupertinoApp`** or use **`Cupertino` widgets** (like `CupertinoButton`, `CupertinoNavigationBar`) conditionally when `Platform.isIOS`.

31.Identify the null-aware **`?.`** and **`??`** with 1-line examples.

- **`?.` (null-aware access):** Access a member only if the object isn't null.

int? len = myString?.length;

- **`??` (if-null operator):** Provide a default value if the expression is null.

int value = myNullableInt ?? 0;

30.Dart `dynamic` vs `Object` (concise).

- **`dynamic`**: Type is **checked at runtime**, allows any operation, no static safety.

dynamic x = 5; x = "hello"; // OK

- `Object`: Base type of all objects, **static type-safe**, requires casting for specific operations.

Object y = 5; // y.toString() works, y + 2 requires cast

29.Two `Container` decoration options (just name them)

1. `color` – sets the background color.

Container(color: Colors.blue)

2. `BoxDecoration` – allows gradient, border, shape, and shadow.

Container(decoration: BoxDecoration(borderRadius: BorderRadius.circular(8), color: Colors.red))

28.Choose **LayoutBuilder** vs **MediaQuery** for widget-specific breakpoints—why?

Use `LayoutBuilder` for widget-specific breakpoints because it provides the **exact constraints of the parent widget**, whereas `MediaQuery` only gives the **whole screen size**.

27.Two responsiveness tactics besides `MediaQuery`.

1. **LayoutBuilder:** Adjusts widget layout based on parent constraints.
2. **Flexible/Expanded:** Makes widgets adapt proportionally within Row, Column, or Flex.

26.Compare Get.to() with Navigator.push (one line).

Get.to(): Simplified navigation without `BuildContext`; **Navigator.push:** Standard Flutter navigation requiring `BuildContext`.

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Here’s a concise answer sheet for all your questions:

---

1. Trace layout with two Spacers and an Icon [2]

```
Row(
  children: [
    Spacer(),
    Icon(Icons.star),
    Spacer(),
  ],
)
```

- **Purpose:** Spaces icon evenly using flexible empty space.

---

2. Purpose of `dispose()`; example resource [2]

- **Purpose:** Clean up resources to prevent memory leaks when a `StatefulWidget` is removed.
- **Example:** `TextEditingController`, `AnimationController`, `StreamSubscription`.

```
@override
void dispose() {
  myController.dispose();
  super.dispose();
}
```

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

3. Define a BLoC Event [1]

```
abstract class CounterEvent {}

class Increment extends CounterEvent {}
```

- **Purpose:** Represents an action that triggers a state change in BLoC.

---

4. Row with 1:2 space using Expanded [2]

```
Row(
  children: [
    Expanded(flex: 1, child: Container(color: Colors.red)),
    Expanded(flex: 2, child: Container(color: Colors.blue)),
  ],
)
```

---

5. Idempotent method that replaces a resource [1]

- **Answer:** `PUT` HTTP method

---

6. What does `await` do? Tiny demo [2]

- **Answer:** Pauses execution until a `Future` completes.

```
void fetchData() async {
  var data = await Future.delayed(Duration(seconds: 1), () => "Hello");
  print(data);
}
```

---

7. Dart List vs Map with micro-examples [2]

- **List:** Ordered collection of items accessed by **index**.

```
List<String> names = ['Ali', 'Sara'];
print(names[0]); // Ali
```

- **Map:** Collection of **key-value pairs** accessed by **key**.

```
Map<String, int> scores = {'Ali': 90, 'Sara': 85};
print(scores['Sara']); // 85
```

---

8. Data types for JSON `{"id":101,"friends":["Ali","Sara"]}` [2]

- Whole JSON → `Map<String,dynamic>`
- `"friends"` → `List<String>`

---

9. Lifecycle method called once; typical use [2]

- **Method:** `initState()`
- **Use:** Initialize controllers, API calls, or subscriptions.

---

10. build() return type + role of BuildContext [2]

- **Return type:** `Widget`
- **BuildContext:** Provides location in widget tree for theme, inherited widgets, and navigation.

---

11. Method to trigger rebuild + why inside callback [2]

- **Method:** `setState(() { ... })`
- **Why:** Wrapping state changes inside `setState` tells Flutter to **rebuild the widget**, so the UI reflects the updated data.

Example:

```
onPressed: () {
  setState(() {
    counter++;
  });
}
```

---

12. FutureBuilder vs StreamBuilder: difference + scenario each [3]

Difference:

- **FutureBuilder:** Handles a **single asynchronous result** (`Future`).
- **StreamBuilder:** Handles **continuous or multiple asynchronous values** (`Stream`).

- **Scenario Example:**
  - **FutureBuilder:** Fetch user profile once from an API.
  - **StreamBuilder:** Listen to live chat messages or real-time sensor data.

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

---

13. PUT vs PATCH (profile) [2]

- **PUT:** Replaces the **entire resource** (e.g., full user profile update).
- **PATCH:** Updates **only specific fields** of the resource (e.g., just the email or name).

---

14. Print order for A/B/C in:

```
void myFunc() {
  print("A");
  myFuture().then((_) => print("C"));
  print("B");
}
```

- **Order:** A → B → C (because `then` is async)

---

15. RenderFlex/unbounded: cause + fixed snippet [3]

- **Cause:** Column/Row inside scrollable without constraints.
- **Fix:**

Expanded(child: ListView(...))

Cause: A `Row` or `Column` has **unbounded space** (e.g., inside a scrollable) and children try to expand infinitely.

Fixed Snippet: Wrap the child with `Expanded` or constrain its size.

Column(

```
 children: [
   Expanded(
     child: ListView(
       children: [Text('Item 1'), Text('Item 2')],
     ),
   ),
 ],
)
```

---

16. Stateless counter: why not updating + two code changes [3]

- **Why:** StatelessWidget can't rebuild; `counter++` doesn't trigger UI update.
- **Fix:**
    1. Change to `StatefulWidget`
    2. Wrap increment in `setState(() { counter++; })`

---

17. One tech advantage (Native) + one business advantage (Flutter) [2]

- **Native:** Full device API access, high performance.
- **Flutter:** Faster development & single codebase → lower cost.

---

18. Two ways for a Row child to fill remaining width [2]

`Expanded:` Fills all available space proportionally.

```
Expanded(child: Container(color: Colors.blue))
```

2. **Flexible:** Fills available space but can be limited with flex and fit.

```
Flexible(flex: 1, child: Container(color: Colors.red))
```

---

*19. async returns what? [2]**

- **Answer:** A `Stream` that emits multiple values over time.
  `async` **function returns a** `Future` that completes with the function's return value.

  Example:

  Future<String> fetchData() async {

  return "Hello";

  }

-

---

20. API in build(): why repeats + correct lifecycle [3]

- **Why:** `build()` runs on every UI rebuild → API called repeatedly.
- **Correct:** Call API in `initState()` and store result in state.

---

21. HTTP 404 & 200 [1]

- **200:** Success
- **404:** Not Found

---

22. Firebase Auth purpose + one provider [2]

- **Purpose:** Authenticate users securely.
- **Provider:** Google Sign-In

---

23. await vs .then(): difference in readability/flow [1]

- **await:** Linear, easier to read

- void fetchData() async {
- String data = await Future.delayed(Duration(seconds: 1), () => "Hello");
- print(data); // Prints after 1 second
- }
- **then():** Chained, nested callbacks

- Future.delayed(Duration(seconds: 1), () => "Hello").then((data) {
- print(data); // Prints after 1 second
- });

---

24. Two Animation widgets + classify [2]

- **Implicit Animation Widget:** AnimatedContainer – automatically animates when properties change.
- **Explicit Animation Widget:** AnimatedBuilder – controlled via an AnimationController for fine-grained animations.

---

25. One role of Drawer & one of AppBar beyond title [1]

- **Drawer:** Provides app-wide navigation
- **AppBar:** Holds actions, tabs, or search widgets

---

26. When prefer BottomNavigationBar over Drawer? [1]

- When **main navigation has ≤5 top-level destinations** for quick switching.

If you want, I can **convert all of these into a super concise 1-page Flutter cheat sheet** for exam prep. Do you want me to do that?

AVAILABLE AT:
Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)