**Module 9: Animations and Transitions**

**Q. 1) Explain the difference between implicit and explicit animations in Flutter.**

**A.** In Flutter, animations are mainly categorized into implicit and explicit animations. The difference lies in how much control you have over the animation and how much work Flutter does for you.

**Implicit Animations**

* **Definition:** Implicit animations are predefined, high-level widgets provided by Flutter that automatically handle the animation for you. You just tell Flutter *what final value* you want, and it smoothly transitions between the old value and the new one.
* **Control:** Minimal. You only set the start and end values, along with the duration and curve. Flutter takes care of the rest.
* **Examples:**
  + AnimatedContainer
  + AnimatedOpacity
  + AnimatedAlign
  + AnimatedPositioned

**Use Case:** Great for simple UI changes where you don’t need full control over the animation sequence.

**Code Example:**

AnimatedContainer(

duration: Duration(seconds: 1),

curve: Curves.easeInOut,

width: \_isExpanded ? 200 : 100,

height: 100,

color: \_isExpanded ? Colors.blue : Colors.red,

)

**Explicit Animations**

* **Definition:** Explicit animations give fine-grained control over how an animation progresses. You use an AnimationController and Tween to define and run the animation manually.
* **Control:** Full. You decide when to start, stop, reverse, or repeat the animation.
* **Examples:**
  + AnimationController
  + Tween
  + AnimatedBuilder
  + FadeTransition, ScaleTransition, etc. (used with controllers)
* **Use Case:** Needed for complex, custom, or interactive animations, like physics-based animations or chained sequences.
* **Code Example:**

class MyAnimation extends StatefulWidget {

@override

\_MyAnimationState createState() => \_MyAnimationState();

}

class \_MyAnimationState extends State<MyAnimation>

with SingleTickerProviderStateMixin {

late AnimationController \_controller;

late Animation<double> \_animation;

@override

void initState() {

super.initState();

\_controller = AnimationController(

duration: Duration(seconds: 2),

vsync: this,

);

\_animation = Tween<double>(begin: 0, end: 200).animate(\_controller);

\_controller.forward();

}

@override

void dispose() {

\_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return AnimatedBuilder(

animation: \_animation,

builder: (context, child) {

return Container(

width: \_animation.value,

height: 100,

color: Colors.blue,

);

},

);

}

}

**Quick Comparison**

| **Feature** | **Implicit Animation** | **Explicit Animation** |
| --- | --- | --- |
| **Ease of Use** | Very easy (just set values) | More complex (need controller) |
| **Control** | Limited | Full |
| **Flexibility** | Simple property changes | Complex, custom animations |
| **Setup** | Minimal | Requires AnimationController |
| **Best For** | Simple UI transitions | Advanced, interactive effects |

**In short:**

* Use implicit animations for quick, simple, and common UI transitions.
* Use explicit animations when you need full control, multiple effects, or custom behaviors.

**Q. 2) Describe the purpose of AnimationController and its usage.**

**A. Purpose of AnimationController in Flutter**

The AnimationController is the engine that drives explicit animations in Flutter.

* It generates values over a given time duration, usually between 0.0 and 1.0.
* These values can be fed into a Tween or used directly to animate widgets.
* It gives you full control: start, stop, pause, reverse, or repeat an animation.

**Think of it as a timeline manager for animations.**

**Key Properties of AnimationController**

* **duration →** how long the animation should run.
* **vsync →** improves performance by syncing the animation with the device’s refresh rate.
* **value →** the current value of the animation (0.0–1.0 by default).
* **status →** tells whether the animation is forward, reverse, completed, or dismissed.

**Usage of AnimationController**

**1. Create the controller**

\_controller = AnimationController(

duration: const Duration(seconds: 2),

vsync: this, // requires SingleTickerProviderStateMixin

);

**2. Attach a Tween (optional)**

\_animation = Tween<double>(begin: 0, end: 200).animate(\_controller);

**3. Start the animation**

\_controller.forward(); // runs forward (0 → 1)

\_controller.reverse(); // runs backward (1 → 0)

\_controller.repeat(); // repeats indefinitely

**4. Use it in a widget**

**Use AnimatedBuilder (recommended) or addListener to rebuild UI.**

AnimatedBuilder(

animation: \_animation,

builder: (context, child) {

return Container(

width: \_animation.value,

height: 100,

color: Colors.blue,

);

},

);

5. Dispose the controller

Always clean up to avoid memory leaks:

@override

void dispose() {

\_controller.dispose();

super.dispose();

}

**Example: Simple Box Animation with AnimationController**

class MyAnimationPage extends StatefulWidget {

@override

\_MyAnimationPageState createState() => \_MyAnimationPageState();

}

class \_MyAnimationPageState extends State<MyAnimationPage>

with SingleTickerProviderStateMixin {

late AnimationController \_controller;

late Animation<double> \_animation;

@override

void initState() {

super.initState();

\_controller = AnimationController(

duration: Duration(seconds: 2),

vsync: this,

);

\_animation = Tween<double>(begin: 50, end: 200).animate(\_controller);

\_controller.forward(); // Start animation automatically

}

@override

void dispose() {

\_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: AnimatedBuilder(

animation: \_animation,

builder: (context, child) {

return Container(

width: \_animation.value,

height: \_animation.value,

color: Colors.blue,

);

},

),

),

);

}

}

**In summary:**

* AnimationController controls time and state of animations.
* It’s essential for explicit animations.
* You combine it with Tween + AnimatedBuilder (or listeners) to bring widgets to life.

**Q. 3) Explain the concept of Hero animations in Flutter.**

**A. What is a Hero Animation in Flutter?**

A Hero animation is a shared element transition between two screens (routes) in Flutter.

* When you navigate from one screen to another, a widget (the Hero) can fly smoothly from its position on the first screen to its position on the second screen.
* It creates a continuous visual connection between UI elements, making navigation feel natural.

**Think of it like:  
A product image on a list screen →** smoothly expands into the product detail screen.

**How Hero Works**

* You wrap the widget you want to animate with the Hero widget.
* Each hero has a unique tag (like an ID).
* Flutter matches heroes with the same tag across the source and destination screens.
* During navigation, Flutter automatically animates the transition between them.

**Key Properties of Hero**

* **tag:** A unique identifier to match heroes across screens.
* **child:** The widget you want to animate.
* **flightShuttleBuilder (optional):** Lets you customize the transition animation.

**Example: Hero Animation**

**Screen 1 – List of Images**

class FirstPage extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text("First Page")),

body: Center(

child: GestureDetector(

onTap: () {

Navigator.push(

context,

MaterialPageRoute(builder: (\_) => SecondPage()),

);

},

child: Hero(

tag: 'hero-image',

child: Image.asset('assets/flutter.png', width: 100),

),

),

),

);

}

}

**Screen 2 – Enlarged Image**

class SecondPage extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text("Second Page")),

body: Center(

child: Hero(

tag: 'hero-image',

child: Image.asset('assets/flutter.png', width: 300),

),

),

);

}

}

**Result:** When you tap the small image, it flies smoothly into the larger image on the next page.

**Why Use Hero Animations?**

* Makes navigation more engaging and fluid.
* Helps users maintain context (they see the same element transform, rather than disappear and reappear).
* Common in apps like shopping (list → detail), photo galleries, and social media.

**Advanced Usage**

* **flightShuttleBuilder →** Customize the transition (e.g., rotate, fade, scale).
* **Hero inside ListView →** Multiple items can have different hero tags.
* **Custom routes (PageRouteBuilder) →** Combine Hero with other animations.

**In summary:**A Hero animation connects the same widget across two screens with a smooth transition, using the Hero widget and a shared tag.