UIT UNIVERSITY

CSC-318 – Mobile Application Development

LAB 6: Animation in Flutter I

Name : _____

Roll No : _____

Section : _____

Semester : _____

FALL 2024

COMPUTER SCIENCE DEPARTMENT

LAB-6 | Animation in Flutter I

1. Objectives:

1. How to use the fundamental classes from the animation library to add animation to a widget.
2. When to use AnimatedWidget vs. AnimatedBuilder.

2. Lab Instructions:

1. Project Setup:

- Create a new Flutter project called AnimatorApp.
- Organize the project structure with separate folders for screens, widgets, and models.
-

3. Essential animation concepts and classes

#

4. What's the point?

- Animation, a core class in Flutter's animation library, interpolates the values used to guide an animation.
- An Animation object knows the current state of an animation (for example, whether it's started, stopped, or moving forward or in reverse), but doesn't know anything about what appears onscreen.
- An AnimationController manages the Animation.
- A CurvedAnimation defines progression as a non-linear curve.
- A Tween interpolates between the range of data as used by the object being animated. For example, a Tween might define an interpolation from red to blue, or from 0 to 255.
- Use Listeners and StatusListeners to monitor animation state changes.

The animation system in Flutter is based on typed Animation objects. Widgets can either incorporate these animations in their build functions directly by reading their current value and listening to their state changes or they can use the animations as the basis of more elaborate animations that they pass along to other widgets.

5. Animation<double>

#

In Flutter, an Animation object knows nothing about what is onscreen. An Animation is an abstract class that understands its current value and its state (completed or dismissed). One of the more commonly used animation types is Animation<double>.

An Animation object sequentially generates interpolated numbers between two values over a certain duration. The output of an Animation object might be linear, a curve, a step function, or any other mapping you can devise. Depending on how the Animation object is controlled, it could run in reverse, or even switch directions in the middle.

Animations can also interpolate types other than double, such as Animation<Color> or Animation<Size>.

An Animation object has state. Its current value is always available in the .value member.

An Animation object knows nothing about rendering or build() functions.

6. CurvedAnimation

#

A CurvedAnimation defines the animation's progress as a non-linear curve.

```
animation = CurvedAnimation(parent: controller, curve: Curves.easeIn);
```

The Curves class defines many commonly used curves, or you can create your own. For example:

```
import 'dart:math';

class ShakeCurve extends Curve {
  @override
  double transform(double t) => sin(t * pi * 2);
}
```

CurvedAnimation and AnimationController (described in the next section) are both of type Animation<double>, so you can pass them interchangeably. The CurvedAnimation wraps the object it's modifying—you don't subclass AnimationController to implement a curve.

7. AnimationController

#

AnimationController is a special Animation object that generates a new value whenever the hardware is ready for a new frame. By default, an AnimationController linearly produces the numbers from 0.0 to 1.0 during a given duration. For example, this code creates an Animation object, but does not start it running:

dart

```
controller =
    AnimationController(duration: const Duration(seconds: 2), vsync: this);
```

content_copy

AnimationController derives from Animation<double>, so it can be used wherever an Animation object is needed. However, the AnimationController has additional methods to control the animation. For example, you start an animation with the .forward() method. The generation of numbers is tied to the screen refresh, so typically 60 numbers are generated per second. After each number is generated, each Animation object calls the attached Listener objects. To create a custom display list for each child, see RepaintBoundary.

When creating an AnimationController, you pass it a vsync argument. The presence of vsync prevents offscreen animations from consuming unnecessary resources. You can use your stateful object as the vsync by adding SingleTickerProviderStateMixin to the class definition. You can see an example of this in animate1 on GitHub.

infoNote

In some cases, a position might exceed the AnimationController's 0.0-1.0 range. For example, the fling() function allows you to provide velocity, force, and position (via the Force object). The position can be anything and so can be outside of the 0.0 to 1.0 range.

A CurvedAnimation can also exceed the 0.0 to 1.0 range, even if the AnimationController doesn't. Depending on the curve selected, the output of the CurvedAnimation can have a wider range than the input. For example, elastic curves such as Curves.elasticIn significantly overshoots or undershoots the default range.

8. Tween

#

By default, the AnimationController object ranges from 0.0 to 1.0. If you need a different range or a different data type, you can use a Tween to configure an animation to interpolate to a different range or data type. For example, the following Tween goes from -200.0 to 0.0:

```
tween = Tween<double>(begin: -200, end: 0);
```

A Tween is a stateless object that takes only begin and end. The sole job of a Tween is to define a mapping from an input range to an output range. The input range is commonly 0.0 to 1.0, but that's not a requirement.
A Tween inherits from Animatable<T>, not from Animation<T>. An Animatable, like Animation, doesn't have to output double. For example, ColorTween specifies a progression between two colors.

```
colorTween = ColorTween(begin: Colors.transparent, end: Colors.black54);
```

A Tween object doesn't store any state. Instead, it provides the evaluate(Animation<double> animation) method that uses the transform function to map the current value of the animation (between 0.0 and 1.0), to the actual animation value.
The current value of the Animation object can be found in the .value method. The evaluate function also performs some housekeeping, such as ensuring that begin and end are returned when the animation values are 0.0 and 1.0, respectively.

9. Tween.animate

#
To use a Tween object, call animate() on the Tween, passing in the controller object. For example, the following code generates the integer values from 0 to 255 over the course of 500 ms.

```
AnimationController controller = AnimationController(
    duration: const Duration(milliseconds: 500), vsync: this);
Animation<int> alpha = IntTween(begin: 0, end: 255).animate(controller);
```

The animate() method returns an Animation, not an Animatable.

The following example shows a controller, a curve, and a Tween:

```
AnimationController controller = AnimationController(
    duration: const Duration(milliseconds: 500), vsync: this);
final Animation<double> curve =
    CurvedAnimation(parent: controller, curve: Curves.easeOut);
Animation<int> alpha = IntTween(begin: 0, end: 255).animate(curve);
```

10. Animation notifications

#
An Animation object can have Listeners and StatusListeners, defined with addListener() and addStatusListener(). A Listener is called whenever the value of the animation changes. The most common behavior of a Listener is to call setState() to cause a rebuild. A StatusListener is called when an animation begins, ends, moves forward, or moves reverse, as defined by AnimationStatus. The next section has an example of the addListener() method, and Monitoring the progress of the animation shows an example of addStatusListener().

11. Animation examples

#
This section walks you through few animation examples. Each section provides a link to the source code for that example.

12. Rendering animations

#
What's the point?

- How to add basic animation to a widget using addListener() and setState().
- Every time the Animation generates a new number, the addListener() function calls setState().
- How to define an AnimationController with the required vsync parameter.
- Understanding the ".." syntax in "..addListener", also known as Dart's *cascade notation*.
- To make a class private, start its name with an underscore (_).

So far you've learned how to generate a sequence of numbers over time. Nothing has been rendered to the screen. To render with an Animation object, store the Animation object as a member of your widget, then use its value to decide how to draw. Consider the following app that draws the Flutter logo without animation:

```
import 'package:flutter/material.dart';

void main() => runApp(const LogoApp());

class LogoApp extends StatefulWidget {
  const LogoApp({super.key});

  @override
  State<LogoApp> createState() => _LogoAppState();
}

class _LogoAppState extends State<LogoApp> {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        margin: const EdgeInsets.symmetric(vertical: 10),
        height: 300,
        width: 300,
        child: const FlutterLogo(),
      ),
    );
  }
}
```

The following shows the same code modified to animate the logo to grow from nothing to full size. When defining an AnimationController, you must pass in a vsync object. The vsync parameter is described in the AnimationController section. The changes from the non-animated example are highlighted:

```
class _LogoAppState extends State<LogoApp> {
class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
  late Animation<double> animation;
  late AnimationController controller;

  @override
  void initState() {
    super.initState();
    controller =
        AnimationController(duration: const Duration(seconds: 2), vsync: this);
    animation = Tween<double>(begin: 0, end: 300).animate(controller)
      ..addListener(() {
        setState(() {
          // The state that has changed here is the animation object's value.
        });
```

```
    });
  controller.forward();
 }

 @override
 Widget build(BuildContext context) {
  return Center(
   child: Container(
    margin: const EdgeInsets.symmetric(vertical: 10),
    height: 300,
    width: 300,
    height: animation.value,
    width: animation.value,
    child: const FlutterLogo(),
   ),
  );
 }

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

The addListener() function calls setState(), so every time the Animation generates a new number, the current frame is marked dirty, which forces build() to be called again. In build(), the container changes size because its height and width now use animation.value instead of a hardcoded value. Dispose of the controller when the State object is discarded to prevent memory leaks.

With these few changes, you've created your first animation in Flutter!

lightbulbDart language trick

You might not be familiar with Dart's cascade notation—the two dots in ..addListener(). This syntax means that the addListener() method is called with the return value from animate(). Consider the following example:

```
animation = Tween<double>(begin: 0, end: 300).animate(controller)
  ..addListener(() {
    // …
  });
```

content_copy

This code is equivalent to:

dart

```
animation = Tween<double>(begin: 0, end: 300).animate(controller);
animation.addListener(() {
    // …
  });
```

13. Simplifying with AnimatedWidget

#

What's the point?

- How to use the AnimatedWidget helper class (instead of addListener() and setState()) to create a widget that animates.
- Use AnimatedWidget to create a widget that performs a reusable animation. To separate the transition from the widget, use an AnimatedBuilder, as shown in the Refactoring with AnimatedBuilder section.
- Examples of AnimatedWidgets in the Flutter API: AnimatedBuilder, AnimatedModalBarrier, DecoratedBoxTransition, FadeTransition, PositionedTransition, Relative PositionedTransition, RotationTransition, ScaleTransition, SizeTransition, SlideTransition.

The AnimatedWidget base class allows you to separate out the core widget code from the animation code. AnimatedWidget doesn't need to maintain a State object to hold the animation. Add the following AnimatedLogo class:

```
class AnimatedLogo extends AnimatedWidget {
  const AnimatedLogo({super.key, required Animation<double> animation})
      : super(listenable: animation);

  @override
  Widget build(BuildContext context) {
    final animation = listenable as Animation<double>;
    return Center(
      child: Container(
        margin: const EdgeInsets.symmetric(vertical: 10),
        height: animation.value,
        width: animation.value,
        child: const FlutterLogo(),
      ),
    );
  }
}
```

AnimatedLogo uses the current value of the animation when drawing itself.
The LogoApp still manages the AnimationController and the Tween, and it passes the Animation object to AnimatedLogo:

```
void main() => runApp(const LogoApp());

class AnimatedLogo extends AnimatedWidget {
  const AnimatedLogo({super.key, required Animation<double> animation})
      : super(listenable: animation);

  @override
  Widget build(BuildContext context) {
    final animation = listenable as Animation<double>;
    return Center(
      child: Container(
        margin: const EdgeInsets.symmetric(vertical: 10),
        height: animation.value,
        width: animation.value,
        child: const FlutterLogo(),
      ),
    );
```

```
  }
}

class LogoApp extends StatefulWidget {
  // ...

  @override
  void initState() {
    super.initState();
    controller =
        AnimationController(duration: const Duration(seconds: 2), vsync: this);
    animation = Tween<double>(begin: 0, end: 300).animate(controller)
      ..addListener(() {
        setState(() {
          // The state that has changed here is the animation object's value.
        });
      });
    animation = Tween<double>(begin: 0, end: 300).animate(controller);
    controller.forward();
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        margin: const EdgeInsets.symmetric(vertical: 10),
        height: animation.value,
        width: animation.value,
        child: const FlutterLogo(),
      ),
    );
  }
  Widget build(BuildContext context) => AnimatedLogo(animation: animation);

  // ...
}
```

14. Monitoring the progress of the animation

#

What's the point?

- Use addStatusListener() for notifications of changes to the animation's state, such as starting, stopping, or reversing direction.
- Run an animation in an infinite loop by reversing direction when the animation has either completed or returned to its starting state.

It's often helpful to know when an animation changes state, such as finishing, moving forward, or reversing. You can get notifications for this with addStatusListener(). The following code modifies the previous example so that it listens for a state change and prints an update. The highlighted line shows the change:

dart

```
class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
```

```
  late Animation<double> animation;
  late AnimationController controller;

  @override
  void initState() {
    super.initState();
    controller =
        AnimationController(duration: const Duration(seconds: 2), vsync: this);
    animation = Tween<double>(begin: 0, end: 300).animate(controller)
      ..addStatusListener((status) => print('$status'));
    controller.forward();
  }
  // ...
}
```

content_copy

Running this code produces this output:

```
AnimationStatus.forward
AnimationStatus.completed
```

content_copy

Next, use addStatusListener() to reverse the animation at the beginning or the end. This creates a "breathing" effect:

dart

```
void initState() {
  super.initState();
  controller =
      AnimationController(duration: const Duration(seconds: 2), vsync: this);
  animation = Tween<double>(begin: 0, end: 300).animate(controller);
  animation = Tween<double>(begin: 0, end: 300).animate(controller)
    ..addStatusListener((status) {
      if (status == AnimationStatus.completed) {
        controller.reverse();
      } else if (status == AnimationStatus.dismissed) {
        controller.forward();
      }
    })
    ..addStatusListener((status) => print('$status'));
  controller.forward();
```

Task 1: Implement Basic Animation

- **Objective**: Use `AnimationController` and `Tween` to animate a widget.
- **Instructions**:
    1. In your `_LogoAppState`, create an `AnimationController` with a duration of 2 seconds.
    2. Create a `Tween<double>` that interpolates between `0` and `300`.
    3. Use the animation value to control the size of a widget (e.g., `FlutterLogo`).

4. Add an `addListener` to the animation to call `setState()` on value changes.
5. Start the animation using `controller.forward()` in `initState()`.

Task 2: Use AnimatedWidget

- **Objective**: Refactor your animation implementation using `AnimatedWidget`.
- **Instructions**:
    1. Create a new widget class named `AnimatedLogo` that extends `AnimatedWidget`.
    2. Pass the `Animation<double>` to the `AnimatedLogo` and use it to control the size of the `FlutterLogo`.
    3. Modify the `build` method of `_LogoAppState` to use the `AnimatedLogo` instead of directly using the `FlutterLogo`.
    4. Ensure the animation still works correctly without using `setState()`.

Task 3: Monitor Animation Progress

- **Objective**: Add listeners to monitor and react to animation state changes.
- **Instructions**:
    1. In your `_LogoAppState`, add a `StatusListener` to your animation.
    2. Print the current animation status to the console (e.g., `AnimationStatus.forward`, `AnimationStatus.completed`).
    3. Implement logic in the status listener to reverse the animation when it completes and start it again when it is dismissed, creating a "breathing" effect.
    4. Test and document how the animation behaves with these status changes.

Bonus Task: Create a Custom Curve

- **Objective**: Implement a custom animation curve.
- **Instructions**:
    1. Create a class that extends `Curve` and overrides the `transform` method to create a unique curve (e.g., a shake effect).
    2. Apply this custom curve to your animation by using `CurvedAnimation`.
    3. Document the impact of using the custom curve on the animation's appearance.