UIT UNIVERSITY

CSC-318 – Mobile Application Development

LAB 7: Animation in Flutter II

Name	: ____________________
Roll No	: ____________________
Section	: ____________________
Semester	: ____________________

FALL 2024

COMPUTER SCIENCE DEPARTMENT

LAB-7 | Animation in Flutter II

1. Objectives:

1. How to use the Hero animation classes from the animation library to add animation to a widget.
2. How to use Standard Hero Animation and Radial Hero Animation.

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. Hero animations

4. What's the point?

1. The hero refers to the widget that flies between screens.
2. Create a hero animation using Flutter's Hero widget.
3. Fly the hero from one screen to another.
4. Animate the transformation of a hero's shape from circular to rectangular while flying it from one screen to another.
5. The Hero widget in Flutter implements a style of animation commonly known as shared element transitions or shared element animations.

5. Hero Animation

You've probably seen hero animations many times. For example, a screen displays a list of thumbnails representing items for sale. Selecting an item flies it to a new screen, containing more details and a "Buy" button. Flying an image from one screen to another is called a hero animation in Flutter, though the same motion is sometimes referred to as a shared element transition.

You can create this animation in Flutter with Hero widgets. As the hero animates from the source to the destination route, the destination route (minus the hero) fades into view. Typically, heroes are small parts of the UI, like images, that both routes have in common. From the user's perspective the hero "flies" between the routes. This lab shows how to create the following hero animations:

6. Standard Hero Animation

A standard hero animation flies the hero from one route to a new route, usually landing at a different location and with a different size.

The following figure shows a typical example. Tapping the flippers in the center of the route flies them to the upper left corner of a new, blue route, at a smaller size. Tapping the flippers in the blue route (or using the device's back-to-previous-route gesture) flies the flippers back to the original route.

7. Radial Hero Animation

In radial hero animation, as the hero flies between routes its shape appears to change from circular to rectangular.

At the start, a row of three circular images appears at the bottom of the route. Tapping any of the circular images flies that image to a new route that displays it with a square shape. Tapping the square image flies the hero back to the original route, displayed with a circular shape.

8. Basic structure of a hero animation

- Use two hero widgets in different routes but with matching tags to implement the animation.
- The Navigator manages a stack containing the app's routes.
- Pushing a route on or popping a route from the Navigator's stack triggers the animation.
- The Flutter framework calculates a rectangle tween, RectTween that defines the hero's boundary as it flies from the source to the destination route. During its flight, the hero is moved to an application overlay, so that it appears on top of both routes.

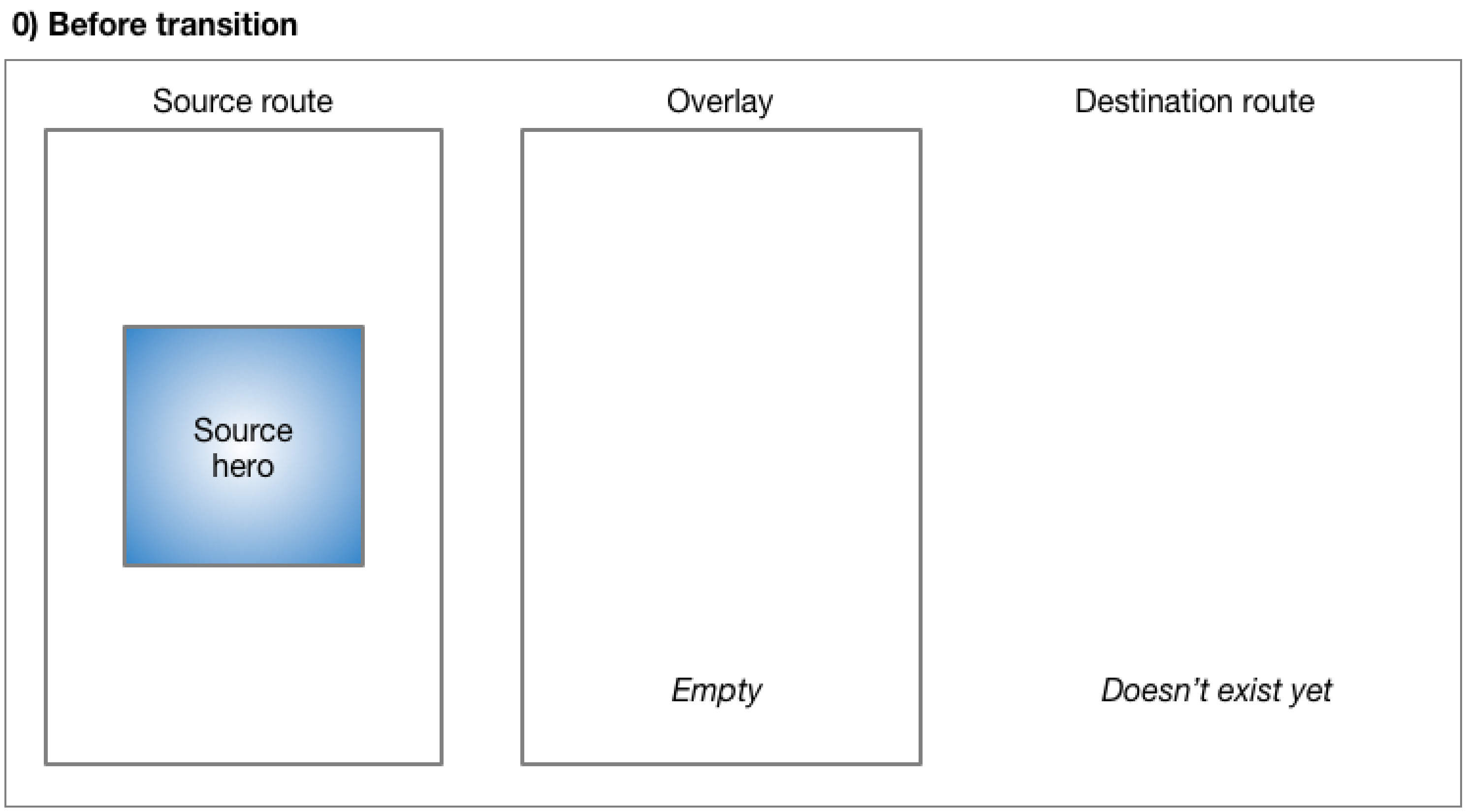
Hero animations are implemented using two Hero widgets: one describing the widget in the source route, and another describing the widget in the destination route. From the user's point of view, the hero appears to be shared, and only the programmer needs to understand this implementation detail. Hero animation code has the following structure:

1. Define a starting Hero widget, referred to as the source hero. The hero specifies its graphical representation (typically an image), and an identifying tag, and is in the currently displayed widget tree as defined by the source route.
2. Define an ending Hero widget, referred to as the destination hero. This hero also specifies its graphical representation, and the same tag as the source hero. It's essential that both hero widgets are created with the same tag, typically an object that represents the underlying data. For best results, the heroes should have virtually identical widget trees.
3. Create a route that contains the destination hero. The destination route defines the widget tree that exists at the end of the animation.
4. Trigger the animation by pushing the destination route on the Navigator's stack. The Navigator push and pop operations trigger a hero animation for each pair of heroes with matching tags in the source and destination routes.

5. Flutter calculates the tween that animates the Hero's bounds from the starting point to the endpoint (interpolating size and position), and performs the animation in an overlay.

The next section describes Flutter's process in greater detail.

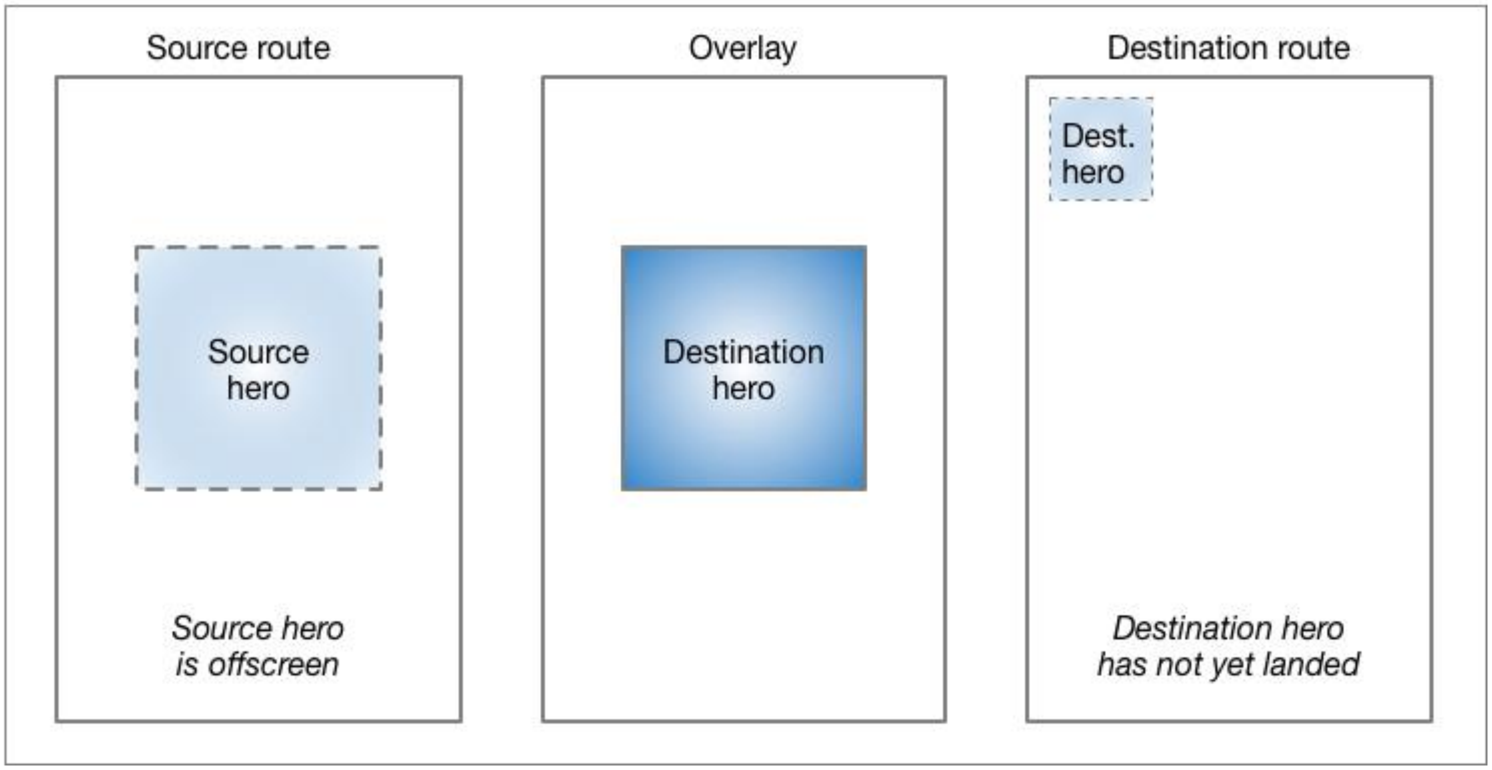
9. Behind the Scenes



Before transition, the source hero waits in the source route's widget tree. The destination route does not yet exist, and the overlay is empty.

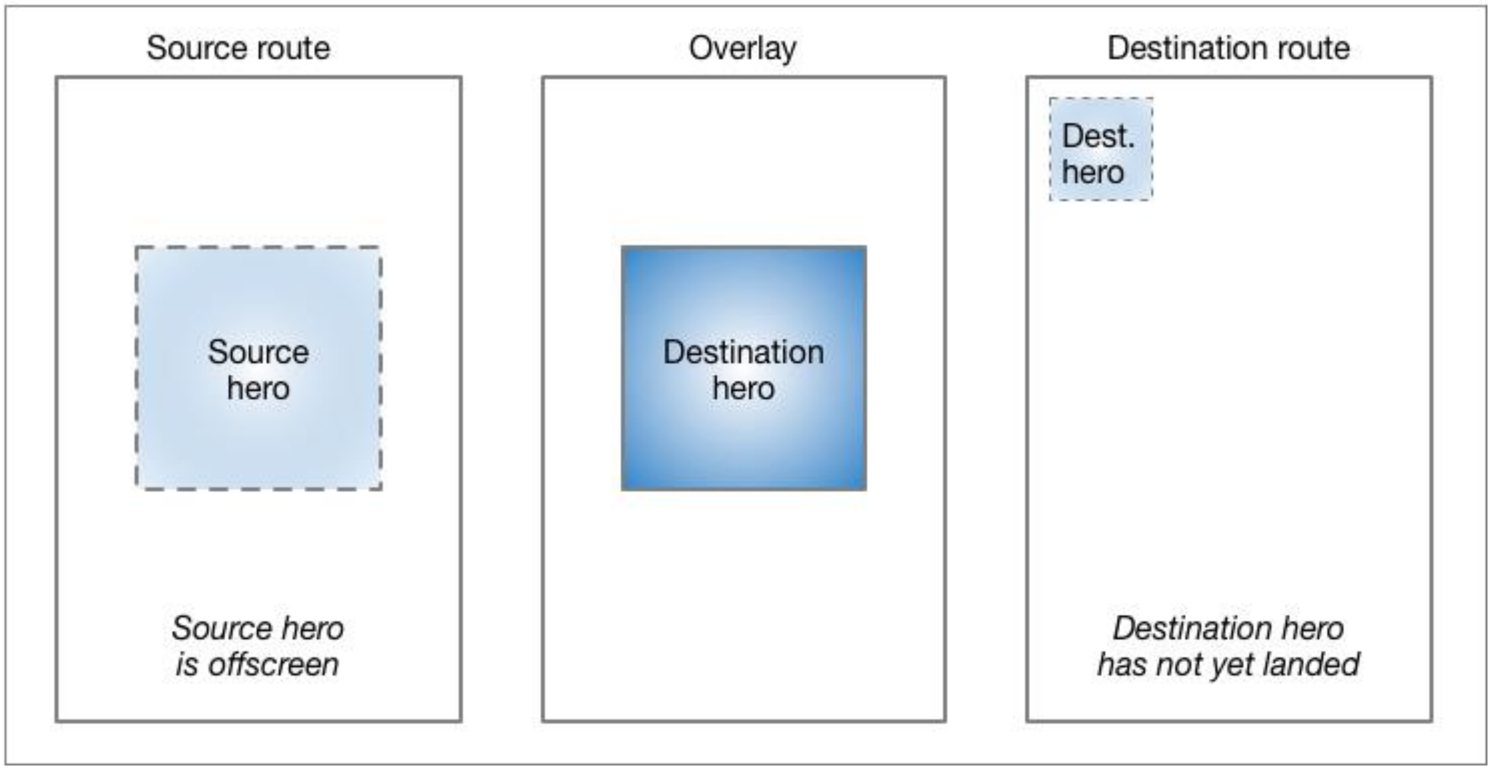
1) Transition begins when destination route is pushed to Navigator

t = 0.0



1) Transition begins when destination route is pushed to Navigator

t = 0.0



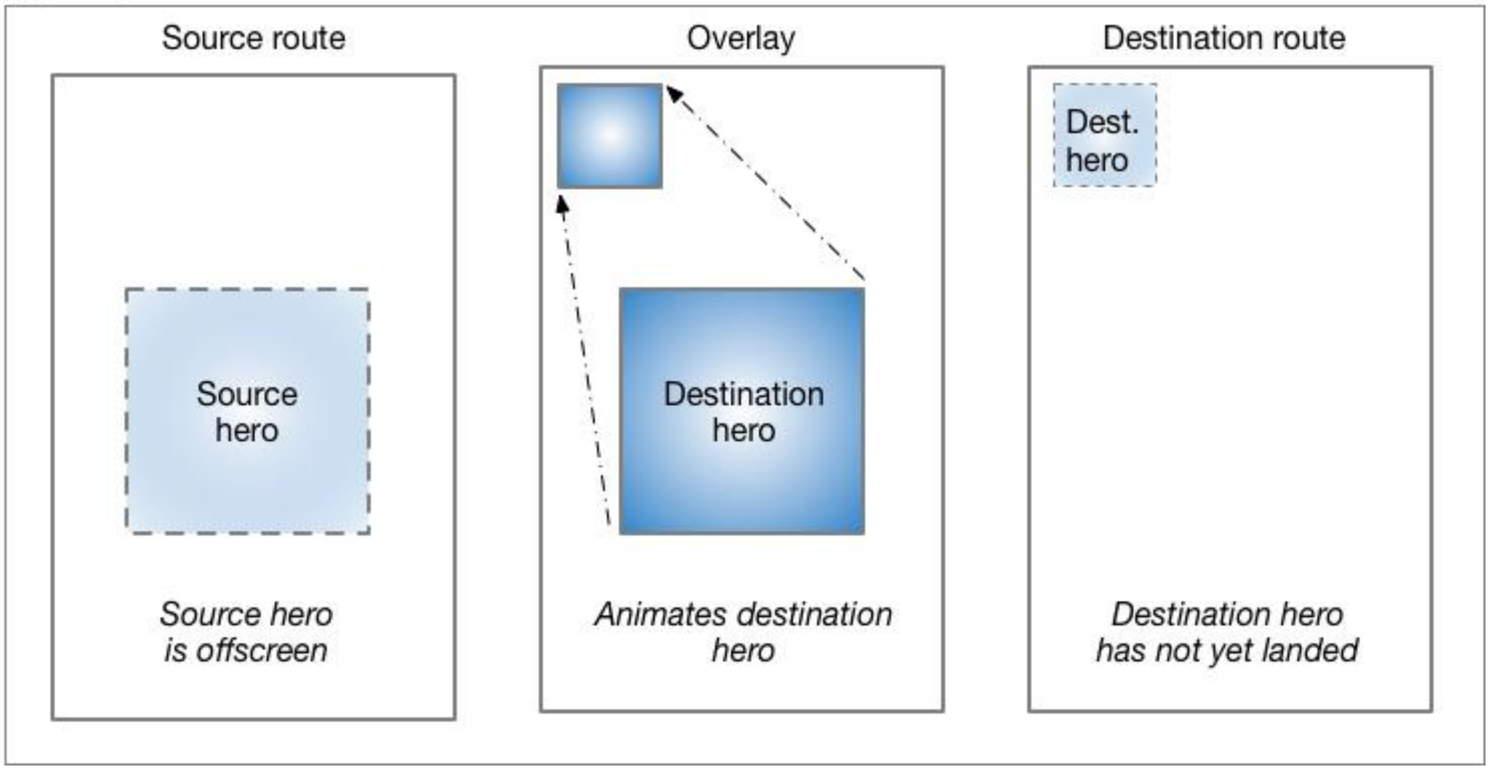
Pushing a route to the Navigator triggers the animation. At t=0.0, Flutter does the following:

Calculates the destination hero's path, offscreen, using the curved motion as described in the Material motion spec. Flutter now knows where the hero ends up.

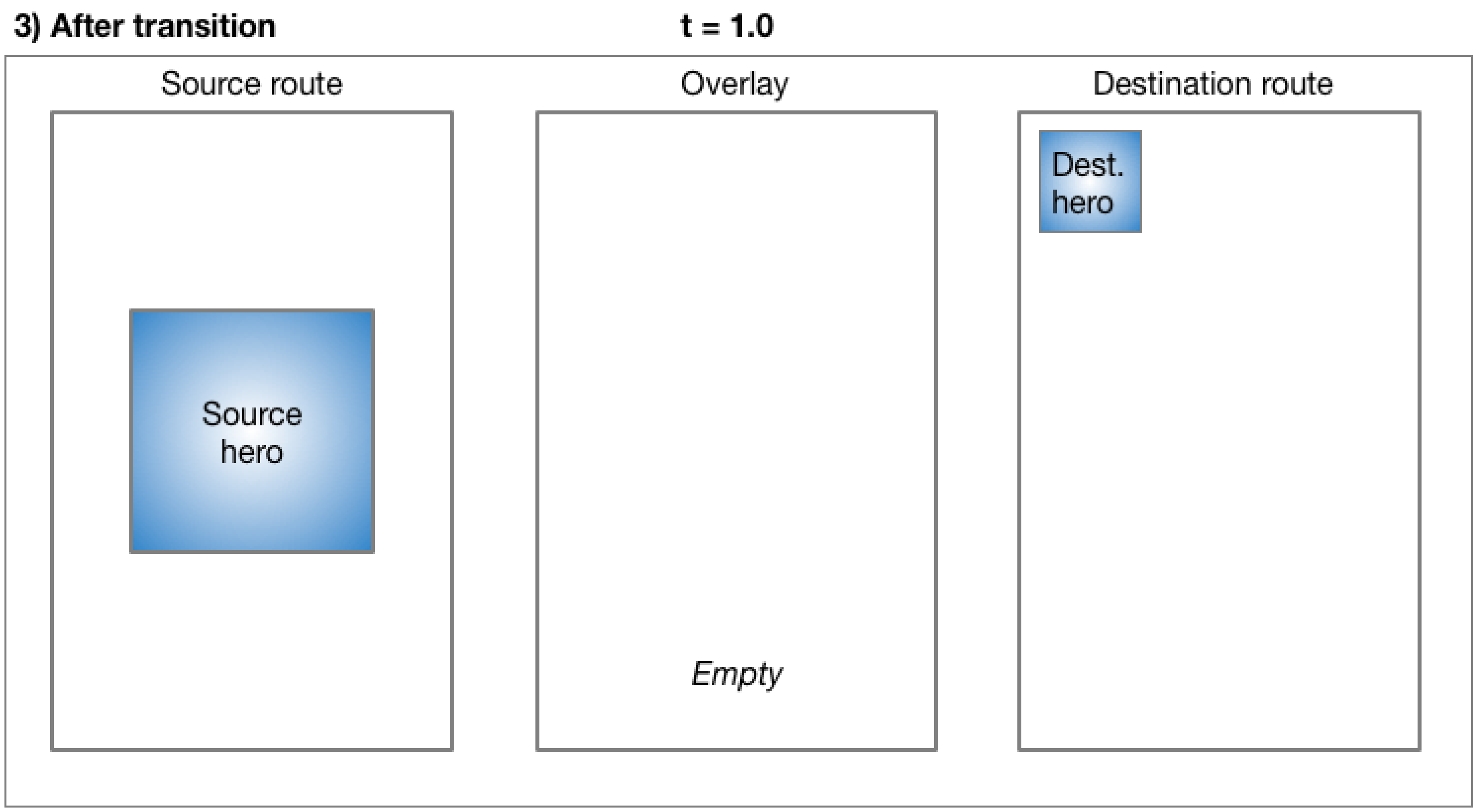
Places the destination hero in the overlay, at the same location and size as the source hero. Adding a hero to the overlay changes its Z-order so that it appears on top of all routes.

Moves the source hero offscreen.

2) In flight



As the hero flies, its rectangular bounds are animated using Tween<Rect>, specified in Hero's createRectTween property. By default, Flutter uses an instance of MaterialRectArcTween, which animates the rectangle's opposing corners along a curved path.



When the flight completes:

- Flutter moves the hero widget from the overlay to the destination route. The overlay is now empty.
- The destination hero appears in its final position in the destination route.
- The source hero is restored to its route.

Popping the route performs the same process, animating the hero back to its size and location in the source route.

10. Essential classes

The examples in this lab use the following classes to implement hero animations:

Hero
The widget that flies from the source to the destination route. Define one Hero for the source route and another for the destination route, and assign each the same tag. Flutter animates pairs of heroes with matching tags.
InkWell
Specifies what happens when tapping the hero. The InkWell's onTap() method builds the new route and pushes it to the Navigator's stack.
Navigator
The Navigator manages a stack of routes. Pushing a route on or popping a route from the Navigator's stack triggers the animation.
Route
Specifies a screen or page. Most apps, beyond the most basic, have multiple routes.
Standard hero animations

What's the point?

1. Specify a route using MaterialPageRoute, CupertinoPageRoute, or build a custom route using PageRouteBuilder. The examples in this section use MaterialPageRoute.
2. Change the size of the image at the end of the transition by wrapping the destination's image in a SizedBox.

3. Change the location of the image by placing the destination's image in a layout widget. These examples use Container.

11. Standard hero animation code

Each of the following examples demonstrates flying an image from one route to another. This lab describes the first example.

hero_animation
Encapsulates the hero code in a custom PhotoHero widget. Animates the hero's motion along a curved path, as described in the Material motion spec.

basic_hero_animation
Uses the hero widget directly. This more basic example, provided for your reference, isn't described in this lab.

What's going on?

Flying an image from one route to another is easy to implement using Flutter's hero widget. When using MaterialPageRoute to specify the new route, the image flies along a curved path, as described by the Material Design motion spec.

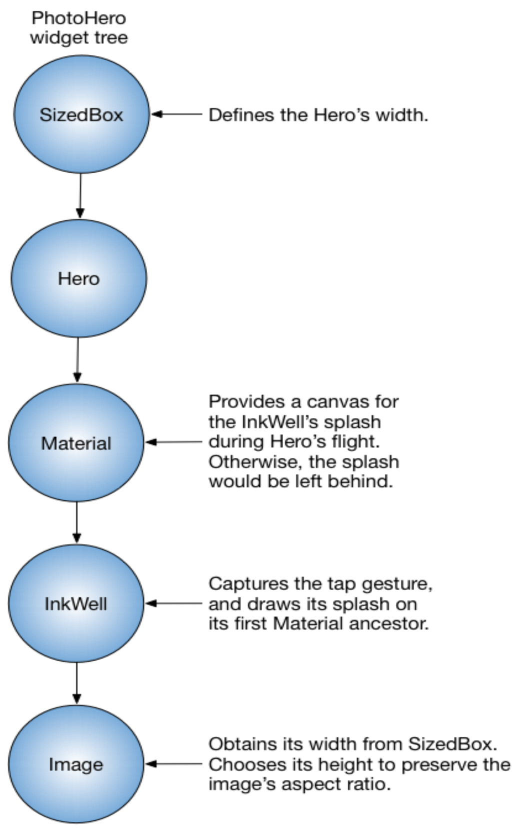
Create a new Flutter example and update it using the files from the hero_animation.

To run the example:

Tap on the home route's photo to fly the image to a new route showing the same photo at a different location and scale.
Return to the previous route by tapping the image, or by using the device's back-to-the-previous-route gesture.
You can slow the transition further using the timeDilation property.

PhotoHero class

The custom PhotoHero class maintains the hero, and its size, image, and behavior when tapped. The PhotoHero builds the following widget tree:


PhotoHero widget tree
SizedBox
Defines the Hero's width.
Hero
Material
Provides a canvas for the InkWell's splash during Hero's flight. Otherwise, the splash would be left behind.
InkWell
Captures the tap gesture, and draws its splash on its first Material ancestor.
Image
Obtains its width from SizedBox. Chooses its height to preserve the image's aspect ratio.

PhotoHero class widget tree

Here's the code:

```
class PhotoHero extends StatelessWidget {
  const PhotoHero({
    super.key,
    required this.photo,
    this.onTap,
    required this.width,
  });

  final String photo;
  final VoidCallback? onTap;
  final double width;

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: width,
      child: Hero(
        tag: photo,
        child: Material(
          color: Colors.transparent,
          child: InkWell(
            onTap: onTap,
            child: Image.asset(
              photo,
              fit: BoxFit.contain,
            ),
          ),
        ),
      ),
    );
  }
}
```

Key information:

1. The starting route is implicitly pushed by MaterialApp when HeroAnimation is provided as the app's home property.
2. An InkWell wraps the image, making it trivial to add a tap gesture to the both the source and destination heroes.
3. Defining the Material widget with a transparent color enables the image to "pop out" of the background as it flies to its destination.
4. The SizedBox specifies the hero's size at the start and end of the animation.
5. Setting the Image's fit property to BoxFit.contain, ensures that the image is as large as possible during the transition without changing its aspect ratio.

12. HeroAnimation class

The HeroAnimation class creates the source and destination PhotoHeroes, and sets up the transition.

Here's the code:

```
class HeroAnimation extends StatelessWidget {
  const HeroAnimation({super.key});

  Widget build(BuildContext context) {
    timeDilation = 5.0; // 1.0 means normal animation speed.

    return Scaffold(
      appBar: AppBar(
        title: const Text('Basic Hero Animation'),
      ),
      body: Center(
        child: PhotoHero(
          photo: 'images/flippers-alpha.png',
          width: 300.0,
          onTap: () {
            Navigator.of(context).push(MaterialPageRoute<void>(
              builder: (context) {
                return Scaffold(
                  appBar: AppBar(
                    title: const Text('Flippers Page'),
                  ),
                  body: Container(
                    // Set background to blue to emphasize that it's a new route.
                    color: Colors.lightBlueAccent,
                    padding: const EdgeInsets.all(16),
                    alignment: Alignment.topLeft,
                    child: PhotoHero(
                      photo: 'images/flippers-alpha.png',
                      width: 100.0,
                      onTap: () {
                        Navigator.of(context).pop();
                      },
                    ),
                  ),
                );
              }
            ));
          },
        ),
      ),
    );
  }
}
```

Key information:

1. When the user taps the InkWell containing the source hero, the code creates the destination route using MaterialPageRoute. Pushing the destination route to the Navigator's stack triggers the animation.
2. The Container positions the PhotoHero in the destination route's top-left corner, below the AppBar.
3. The onTap() method for the destination PhotoHero pops the Navigator's stack, triggering the animation that flies the Hero back to the original route.
4. Use the timeDilation property to slow the transition while debugging.

13. Radial hero animations

What's the point?

1. A radial transformation animates a circular shape into a square shape.
2. A radial hero animation performs a radial transformation while flying the hero from the source route to the destination route.
3. MaterialRectCenter-Arc-Tween defines the tween animation.
4. Build the destination route using PageRouteBuilder.
5. Flying a hero from one route to another as it transforms from a circular shape to a rectangular shape is a slick effect that you can implement using Hero widgets. To accomplish this, the code animates the intersection of two clip shapes: a circle and a square. Throughout the animation, the circle clip (and the image) scales from minRadius to maxRadius, while the square clip maintains constant size. At the same time, the image flies from its position in the source route to its position in the destination route. For visual examples of this transition, see Radial transformation in the Material motion spec.

This animation might seem complex (and it is), but you can customize the provided example to your needs. The heavy lifting is done for you.

Radial hero animation code
Each of the following examples demonstrates a radial hero animation. This lab describes the first example.

radial_hero_animation
A radial hero animation as described in the Material motion spec.

basic_radial_hero_animation
The simplest example of a radial hero animation. The destination route has no Scaffold, Card, Column, or Text. This basic example, provided for your reference, isn't described in this lab.

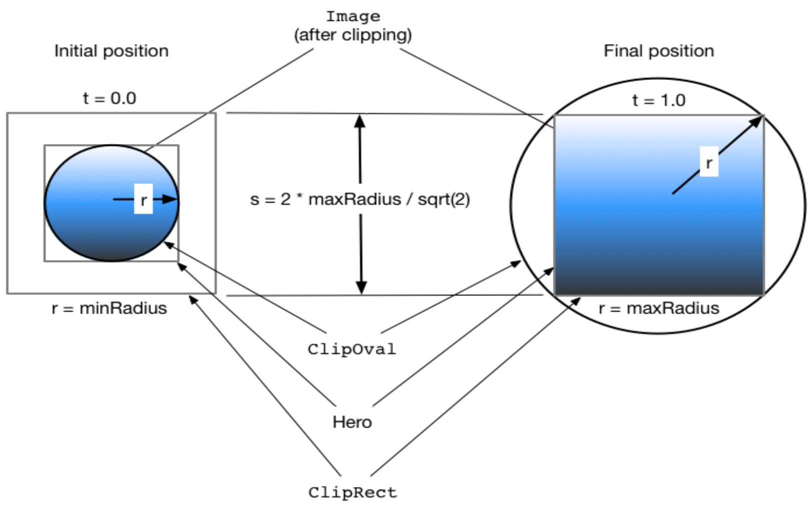
radial_hero_animation_animate_rectclip
Extends radial_hero_animation by also animating the size of the rectangular clip. This more advanced example, provided for your reference, isn't described in this lab.

Pro tip
The radial hero animation involves intersecting a round shape with a square shape. This can be hard to see, even when slowing the animation with timeDilation, so you might consider enabling the debugPaintSizeEnabled flag during development.

What's going on?

The following diagram shows the clipped image at the beginning (t = 0.0), and the end (t = 1.0) of the animation.

Radial transformation from beginning to end



The blue gradient (representing the image), indicates where the clip shapes intersect. At the beginning of the transition, the result of the intersection is a circular clip (ClipOval). During the transformation, the ClipOval scales from minRadius to maxRadius while the ClipRect maintains a constant size. At the end of the transition the intersection of the circular and rectangular clips yield a rectangle that's the same size as the hero widget. In other words, at the end of the transition the image is no longer clipped.

To run the example:

Tap on one of the three circular thumbnails to animate the image to a larger square positioned in the middle of a new route that obscures the original route.
Return to the previous route by tapping the image, or by using the device's back-to-the-previous-route gesture.
You can slow the transition further using the timeDilation property.

Photo class

The Photo class builds the widget tree that holds the image:

```
class Photo extends StatelessWidget {
  const Photo({super.key, required this.photo, this.color, this.onTap});

  final String photo;
  final Color? color;
  final VoidCallback onTap;

  Widget build(BuildContext context) {
    return Material(
      // Slightly opaque color appears where the image has transparency.
      color: Theme.of(context).primaryColor.withOpacity(0.25),
      child: InkWell(
        onTap: onTap,
        child: Image.asset(
          photo,
          fit: BoxFit.contain,
        ),
      ),
    );
  }
}
```

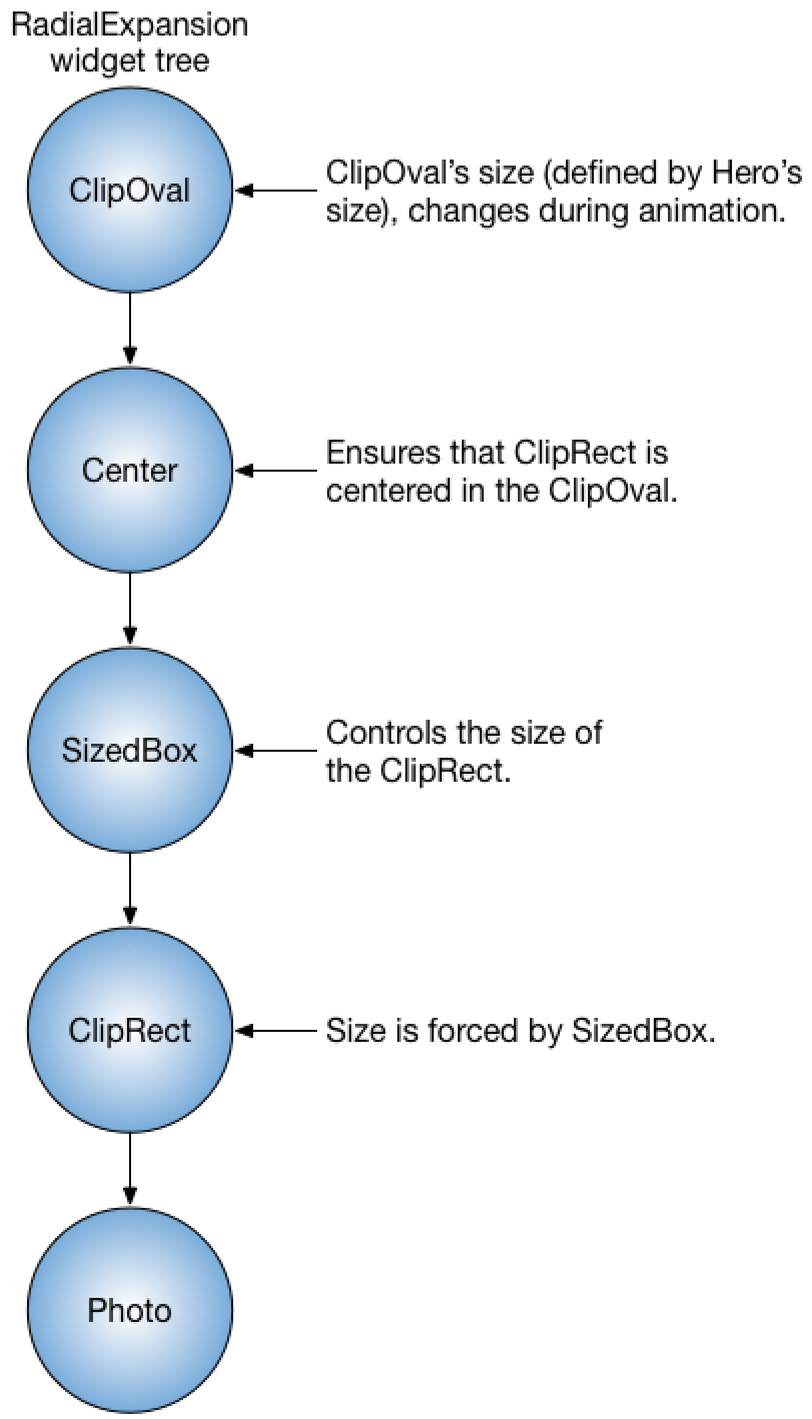
Key information:

1. The InkWell captures the tap gesture. The calling function passes the onTap() function to the Photo's constructor.
2. During flight, the InkWell draws its splash on its first Material ancestor.
3. The Material widget has a slightly opaque color, so the transparent portions of the image are rendered with color. This ensures that the circle-to-square transition is easy to see, even for images with transparency.
4. The Photo class does not include the Hero in its widget tree. For the animation to work, the hero wraps the RadialExpansion widget.

RadialExpansion class

The RadialExpansion widget, the core of the demo, builds the widget tree that clips the image during the transition. The clipped shape results from the intersection of a circular clip (that grows during flight), with a rectangular clip (that remains a constant size throughout).

To do this, it builds the following widget tree:


RadialExpansion widget tree
ClipOval
ClipOval's size (defined by Hero's size), changes during animation.
Center
Ensures that ClipRect is centered in the ClipOval.
SizedBox
Controls the size of the ClipRect.
ClipRect
Size is forced by SizedBox.
Photo

RadialExpansion widget tree

Here's the code:

```
class RadialExpansion extends StatelessWidget {
  const RadialExpansion({
    super.key,
    required this.maxRadius,
    this.child,
  }) : clipRectSize = 2.0 * (maxRadius / math.sqrt2);
  final double maxRadius;
  final clipRectSize;
  final Widget child;

  @override
  Widget build(BuildContext context) {
    return ClipOval(
      child: Center(
        child: SizedBox(
          width: clipRectSize,
          height: clipRectSize,
          child: ClipRect(
            child: child, // Photo
          ),
        ),
      ),
    );
  }
}
```

Key information:

The hero wraps the RadialExpansion widget.

As the hero flies, its size changes and, because it constrains its child's size, the RadialExpansion widget changes size to match.

The RadialExpansion animation is created by two overlapping clips.

The example defines the tweening interpolation using MaterialRectCenterArcTween. The default flight path for a hero animation interpolates the tweens using the corners of the heroes. This approach affects the hero's aspect ratio during the radial transformation, so the new flight path uses MaterialRectCenterArcTween to interpolate the tweens using the center point of each hero.

Here's the code:

```
static RectTween _createRectTween(Rect? begin, Rect? end) {
  return MaterialRectCenterArcTween(begin: begin, end: end);
}
```

The hero's flight path still follows an arc, but the image's aspect ratio remains constant.

Task 1: Implement a Standard Hero Animation

Objective: Create a simple standard Hero animation where an image "flies" between two screens when tapped.

Instructions:

1. **Create a Flutter app** with a `MaterialApp` widget.
2. **Create two screens (routes)**:
   - A **home screen** displaying a photo (using `PhotoHero` widget).
   - A **detail screen** that shows the same photo in a different size and position.
3. Use the `Hero` widget with a matching **tag** to animate the image between the two routes. When the image is tapped, it should animate to the detail screen, and tapping it again on the detail screen should return to the home screen.
4. **Optional:** Slow the animation speed by modifying `timeDilation` during testing (set `timeDilation = 5.0;`).

Expected Outcome: When the user taps on the photo on the home screen, it should smoothly transition to the detail screen, scaling and moving in a curved path. Tapping the photo on the detail screen should return it to the original position.

Task 2: Create a Radial Hero Animation

Objective: Implement a radial hero animation where an image flies from one screen to another while changing its shape from a circle to a square.

Instructions:

1. **Create a Flutter app** with two screens (routes):
   - The **first screen** displays three circular images aligned horizontally.
   - The **second screen** should display the tapped image in a square shape.
2. When any of the images on the first screen is tapped, the image should fly to the second screen, changing from circular to square.
3. Use the `RadialExpansion` widget to clip the image in the first screen and animate it into a square shape as it moves to the second screen.
4. **Optional:** Use `timeDilation` to slow the animation and make it more visible during development.

Expected Outcome: Tapping one of the circular images should animate the image to a square shape on the second screen. The transition should include the change in shape and smooth movement.

Task 3: Customize the Hero Animation Transition Duration

Objective: Modify the default behavior of the Hero animation by customizing the **duration** and **curve** of the transition.

Instructions:

1. Use the `Hero` widget to animate an image between two screens (similar to Task 1).
2. Add a custom `Tween` to the `Hero` widget's `createRectTween` property to change the duration of the transition and use a custom curve (e.g., `Curves.easeInOut` or `Curves.elasticOut`).
3. Experiment with different durations and curves to see how the animation timing changes.

Expected Outcome: The image should still "fly" from one screen to another, but the animation should feel different based on the chosen duration and curve. Test with different curve types (e.g., `Curves.easeInOut`, `Curves.elasticOut`, etc.) and durations.

Bonus Task: Add a Custom Radial Hero Animation with a Color Change

Objective: Extend the radial hero animation by making the image's background color change as the image transforms from circular to square.

Instructions:

1. **Extend the radial hero animation** from Task 2, but add an additional effect: as the image transitions from circular to square, change the background color of the image.
2. You can use an `AnimatedContainer` to animate the background color based on the transition.
3. Experiment with both the **shape transformation** and the **color change** so that it feels like the color is smoothly blending into the new shape.

Expected Outcome: The image should transform from circular to square, with the background color transitioning in sync with the shape transformation. The color should change smoothly as part of the radial transition.

Deliverables:

For each task, the students should:

- Provide a well-commented Flutter codebase with the implemented Hero animations.
- Submit a brief explanation of how the transition works, including the reasoning behind any customizations made (e.g., custom duration, curve, or background color change).