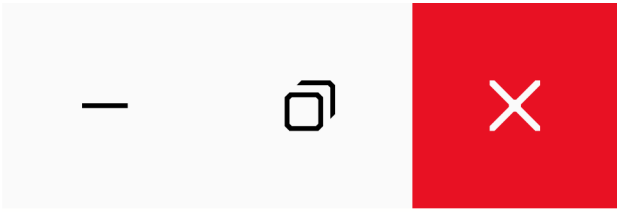


Animation

What is Animation
Basic Animation
Smooth Animation
Keyframe Animation

June 14

What is Animation

Animation

Animation is the simulation of movement using a series of images (or drawings, models, etc.).

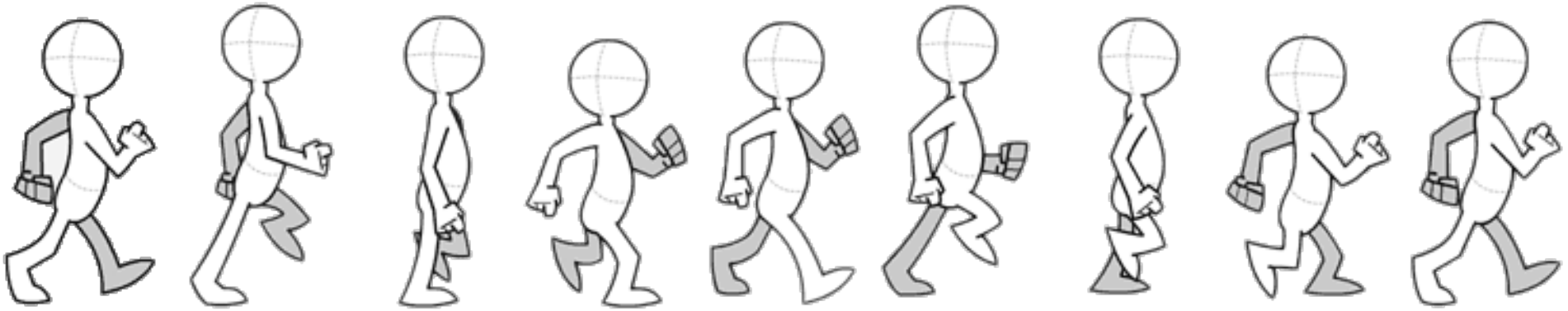


IMAGE SOURCE: https://www.angryanimator.com/word/2010/11/26/tutorial-2-walk-cycle/

Animation Terminology

Frame: each image (or state) of an animation sequence

Frame rate: number of frames to display per second

Key Frame: defines the beginning and ending points of a transition

- **Tweening**: interpolation of frames between two key frames
- **Easing**: a function that controls how tweening is calculated

Animation Terminology

In user interface programming, we use animation to draw attention to UI elements, or to visually indicate change.

To do this, we typically animate numerical parameters that affect how graphics are drawn over a period of time.

- parameters are often related to transformations
  (e.g. translate X and Y position to animate drawing position)
- parameters can be anything numeric: fill, stroke weight, etc.
- animating non-numeric values (e.g. a String or Image) is possible, but custom tweening methods are needed

Frame Rate

Measured in **frames-per-second (fps)**. Can be expressed as Hertz (Hz): International System of Units (SI) measure defined as one cycle per second (e.g., 60 FPS = 60 Hz)

Common device and media frame rates:

- Hand-drawn animation: as low as 12 FPS, usually 24 fps
- GIFs: usually 15 to 24 fps
- Film: standard 24 fps, high framerate 60 fps
- Legacy Broadcast Television: NTSC: 30 fps*, PAL 25 fps*
- Computer displays: 60 fps or more
- Computer games: 60 fps or more
- Virtual Reality displays: 90 fps, 120 fps, or more

e.g. Loving Vincent. 12 fps. https://youtu.be/CGzKnyhYDQI

Basic Animation

Animation Using `java.util.Timer`

A **timer** triggers an event after some time period

1. Set time period to time interval for desired frame rate, e.g., 50 FPS (i.e., frequency of $f = {}^{1}/_{50}\ Hz$, new frame every ${}^{1}/_{50}\ s = 20\ ms$
2. In the timer event handler
   a) update parameters you want to animate
   b) redraw an updated image for the frame
3. Restart the timer for the next interval

```
val animation = Timer().apply {
    scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            myCanvas.graphicsContext2D.apply {
                clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                myDrawable.y += 2.0
                myDrawable.draw(myCanvas.graphicsContext2D)
            }
        }
    }, 0L, 20L)
}
```

Animation Using `java.util.Timer`

```
private val animation = Timer()

override fun start(stage: Stage) {
    val myDrawable = FillCirc(0.0, 0.0, 50.0, Color.GREEN, "Green Circle")
    val myCanvas = Canvas(480.0, 320.0)
    animation.apply {
        scheduleAtFixedRate(object : TimerTask() {
            override fun run() {
                myCanvas.graphicsContext2D.apply {
                    clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                    myDrawable.y += 2.0
                    myDrawable.draw(myCanvas.graphicsContext2D)
                }
            }
        }, 0L, 20L)
    }
    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(myCanvas), myCanvas.width, myCanvas.height)
    stage.show()
}

override fun stop() {
    super.stop()
    animation.cancel()
}
```

Timers and the UI Thread

Many UI frameworks are single-threaded (including JavaFX)

- the event dispatch queue is one thread to avoid deadlocks and race conditions due to unpredictable user-generated events

These UI frameworks are typically not thread-safe

- to reduce execution burden, complexity, etc.

Most modifications to the scene graph (and nodes it contains) must be performed on the UI execution thread

- otherwise, an exception is thrown

This has implications for animation timers.

Animation Using `java.util.Timer`

```
val animation = Timer().apply {
    scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            myCanvas.graphicsContext2D.apply {
                clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                myDrawable.y += 2.0
                myDrawable.draw(this)
            }
        }
    }, 0L, 20L)
}
```

`java.util.Timer` runs on a separate thread from the JavaFX application thread!

This means that it may cause an exception if modifications to the scene graph are attempted in the event handler.

Demo: Animation/Timers

Animation Using `java.util.Timer` & `Platform.runLater`

```
val animation = Timer().apply {
    scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            Platform.runLater {
                myCanvas.graphicsContext2D.apply {
                    clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                    myDrawable.y += 2.0
                    myDrawable.draw(this)
                }
            }
        }
    }, 0L, 50L)
}
```

How do we update the UI from a different thread? Pass it as a `Runnable` to the UI thread.

`Platform.runLater` runs the specified `Runnable` on the JavaFX application thread (at the next available time, not specified).

Animation Using javafx.animation.AnimationTimer

```
val animation = object : AnimationTimer() {
    override fun handle(now: Long) {
        myCanvas.graphicsContext2D.apply {
            myDrawable.y += 2.0
            clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
            myDrawable.draw(this)
        }
    }
}.start()
```

`AnimationTimer` runs the specified `Runnable` on the JavaFX UI thread at 60 fps.

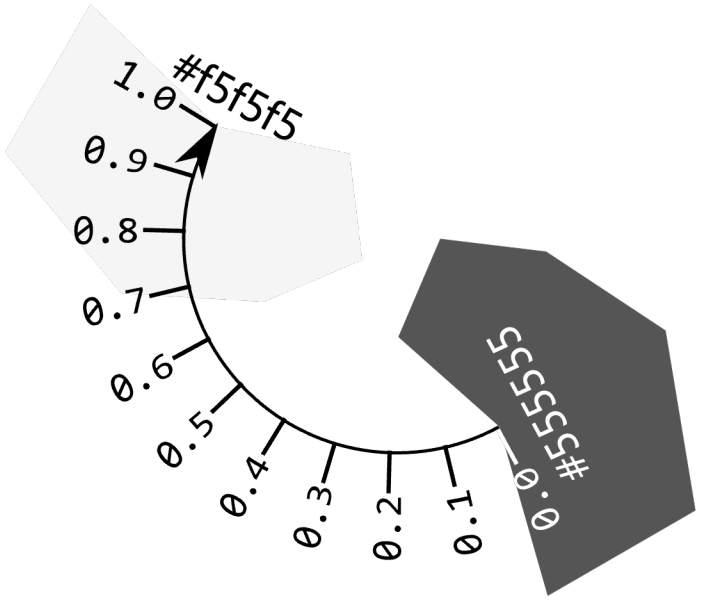
`Platform.runLater` is not needed in this case, because this particular timer (`javafx` package) is designed to safely run on the UI thread.

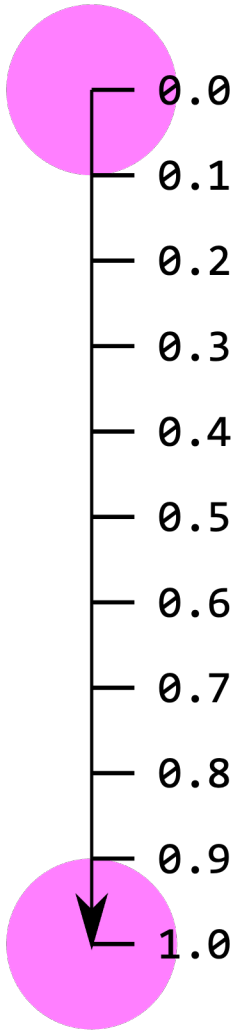
Demo: Animation/Timers

Animation Control by the Drawable

Animation can be thought of as moving through a range from `0.0`, which represents the start state, to `1.0`, which represents the end state.

The animation value is mapped to one or more visual or other property (e.g., location, colour, text), each with a definition of start and end states.

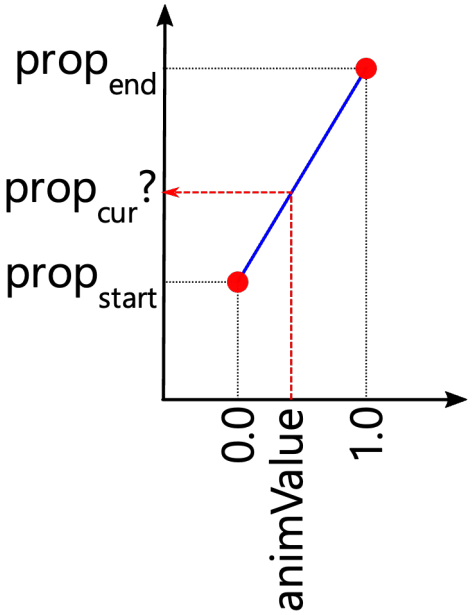




Animation Control by the Drawable

While start and end states are known, intermediate states must be calculated.

A basic approach for this is **linear interpolation**:



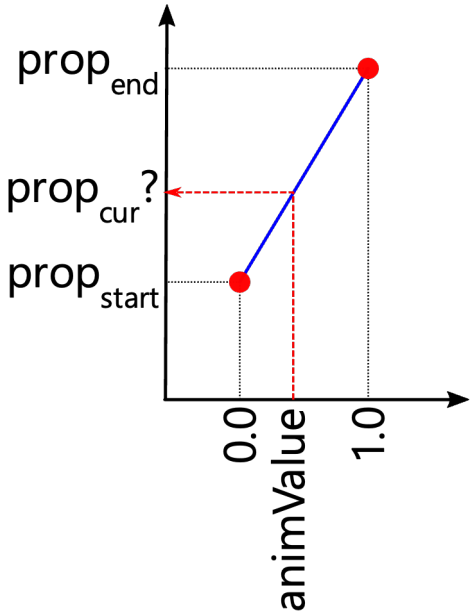
[$prop_{start}$, $prop_{end}$] is the range for the property we wish to animate. e.g. x coordinate, or color.

`animValue` is the value [0,1] that we use to track progress through the animation

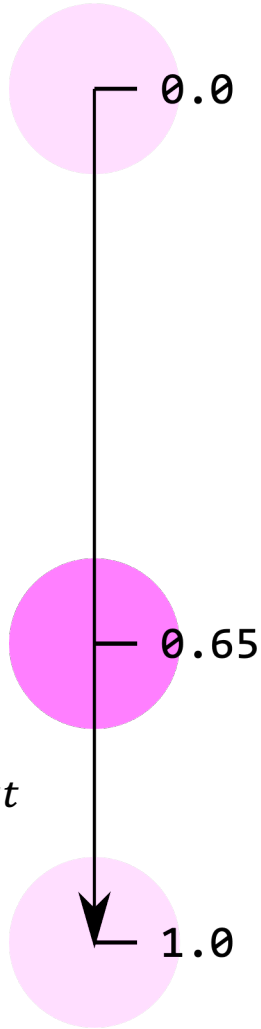
$prop_{cur}$ is the derived value at any point in time

Animation Control by the Drawable

The start and end states of the property are associated with values 0.0 and 1.0 respectively, and intermediate values of the property $prop_{cur}$ must be interpolated from the current value of the animation $animValue_{cur}$ (“’tweening”).



$$prop(animValue) = animValue * (prop_{end} - prop_{start}) + prop_{start}$$



Animation Control by the Drawable

Animation can be thought of as moving through a range from `0.0`, which represents the start state, to `1.0`, which represents the end state.

```
private var animValue = 0.0    // [0.0 ... 1.0]
```

The animation value is mapped to one or more visual or other property (e.g., location, colour, text), each with a definition of start and end states.

```
private var animPropStart = 0.0 // start state
private var animPropEnd = 200.0 // end state
```

This happens over a fixed time period for the animation (e.g. 1000 ms).

The value may also have a change rate per frame / call to `animate`.

```
private var animSpeed = 0.005
```

Animation Control by the Drawable

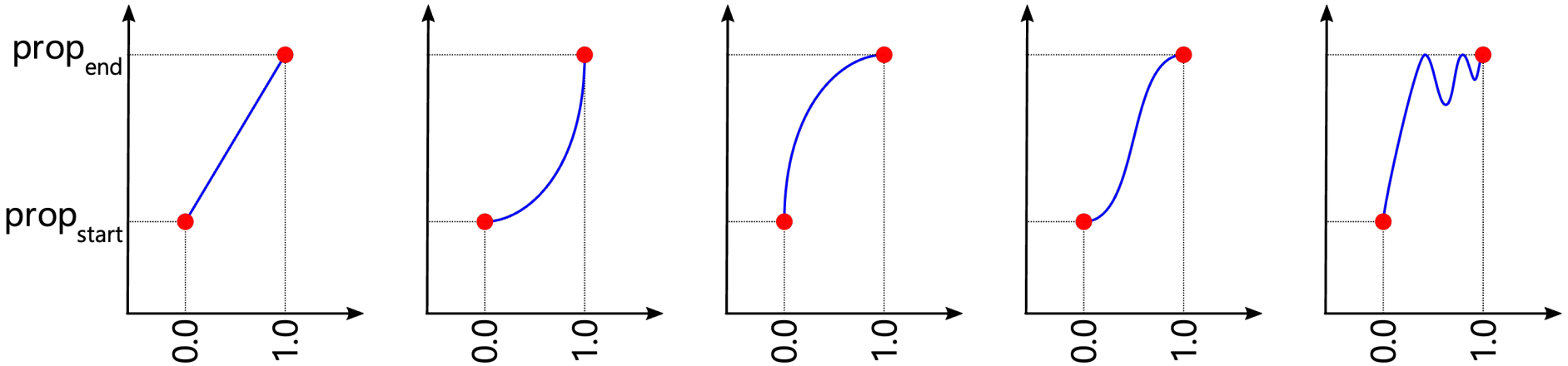
```
fun Double.lerp(min: Double, max: Double) : Double {
    return this * (max - min) + min
}

enum class Direction(val value: Double) { UP(-1.0), DOWN(1.0) }

private var animValue = 0.0          // [0.0 ... 1.0]
private var animSpeed = 0.005        // change to animValue per call
private var animDirection = Direction.DOWN
private var animPropertyStart = 50.0 // property start state
private var animPropertyEnd = 250.0  // property end state

override fun animate() {
    animValue += animSpeed * animDirection.value
    y = animValue.lerp(animPropertyStart, animPropertyEnd)
    when {
        animValue + animSpeed > 1.0 -> animDirection = Direction.UP
        animValue - animSpeed < 0.0 -> animDirection = Direction.DOWN
    }
}
```

Demo: Animation/Linear

Smooth Animation

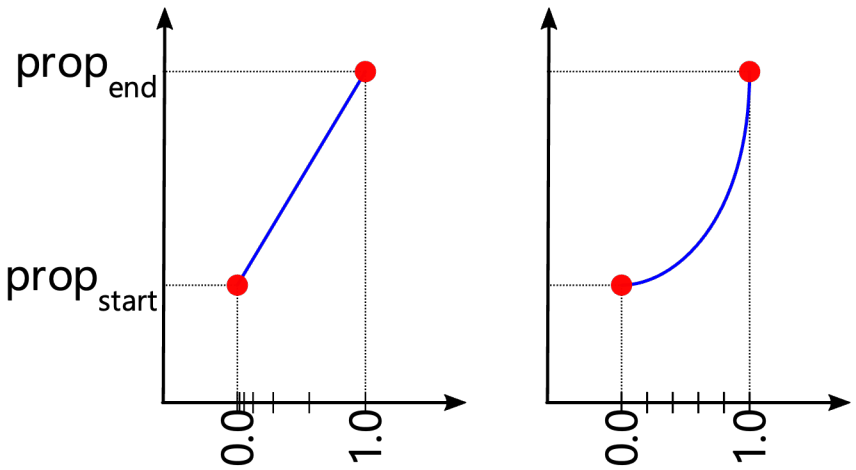
Easing

Oftentimes, linear interpolation is not good enough (“feels unnatural”), and other types of interpolations would be preferred.



Easing

This can be achieved by “easing” the current animation value.

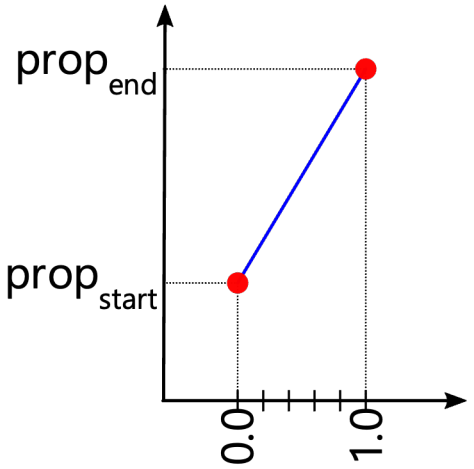


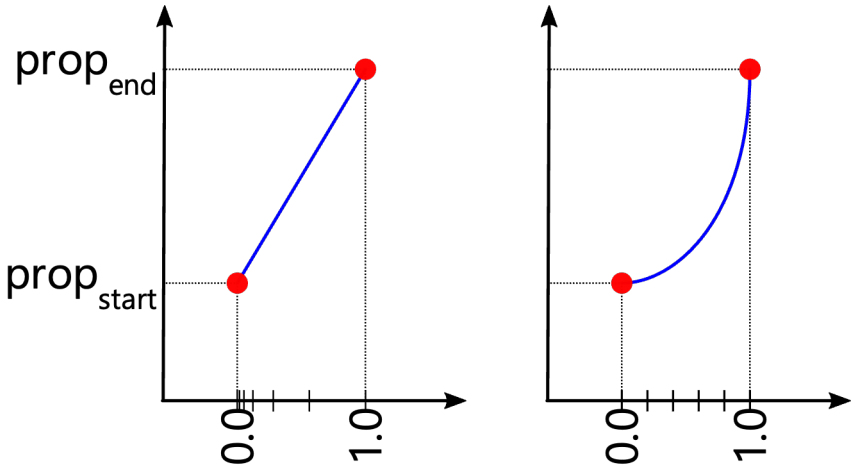
$$prop(animValue) = ease(animValue) * (prop_{end} - prop_{start}) + prop_{start}$$

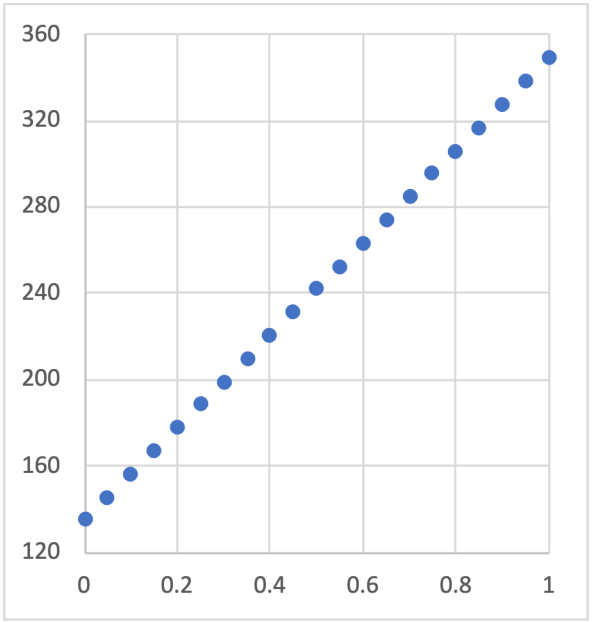
```
fun Double.lerp(min: Double, max: Double) : Double {
    return this * (max - min) + min
}
fun easeIn(x: Double) : Double {
    return x.pow(2)
}
curProp = easeIn(animValue).lerp(animPropStart, animPropEnd)
```

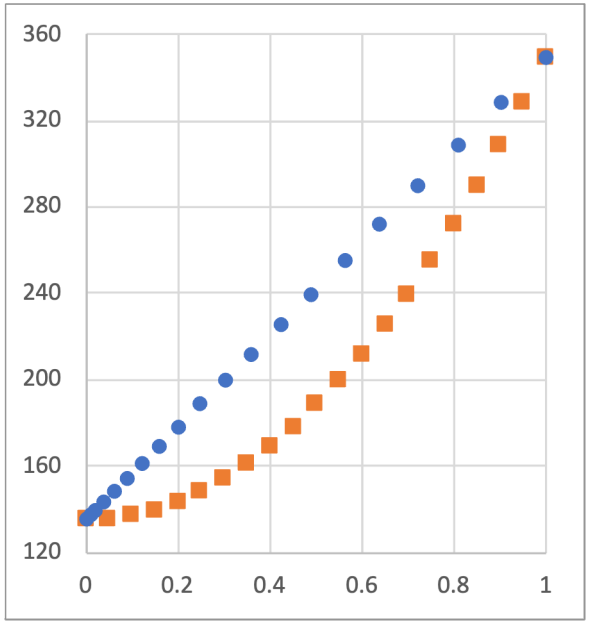
Easing

Easing the animation value results in altering the x-axis.



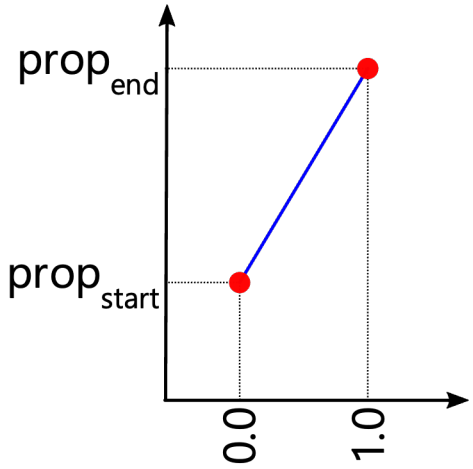






Easing Functions

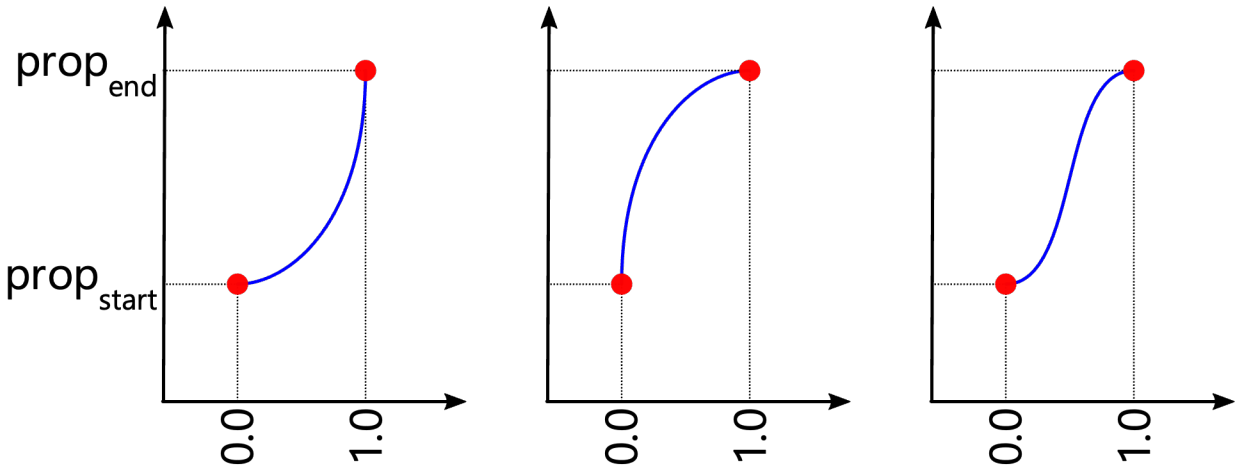
```
val flip = { x: Double -> 1.0 - x }
val easeIn = { x: Double -> x.pow(2) }
val easeOut = { x: Double -> flip(easeIn(flip(x))) }
val easeInOut = { x: Double -> x.lerp(easeIn(x), easeOut(x)) }

// interpolate value with no easing (i.e., linear)
curProp = animValue.lerp(animPropStart, animPropEnd)
```



Demo: Animation/Tweening

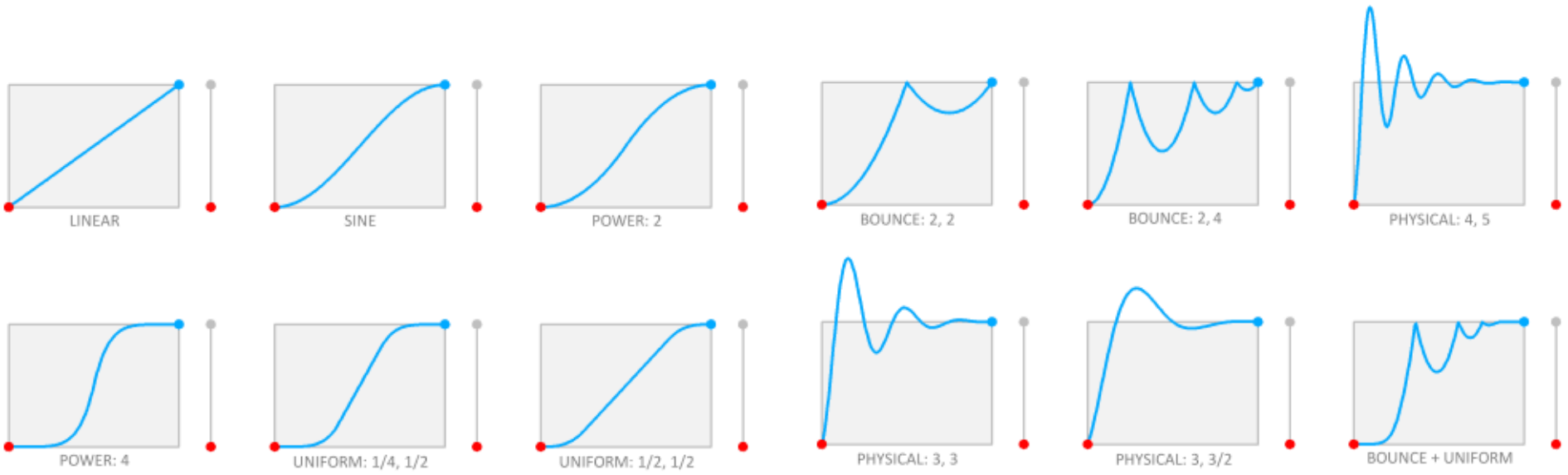
Easing Functions

```
// interpolate value with easeIn (i.e., quadratic)
curProp = easeIn(animValue).lerp(animPropStart, animPropEnd)

// interpolate value with easeOut (i.e., flipped quadratic)
curProp = easeOut(animValue).lerp(animPropStart, animPropEnd)

//interpolate value with easeInOut
curProp = easeInOut(animValue).lerp(animPropStart, animPropEnd)
```



Easing Function Resources

- http://robertpenner.com/easing/
- https://greensock.com/docs/v3/Eases
- https://www.febucci.com/2018/08/easing-functions/



[VIDEO SOURCE] https://www.alanzucconi.com/2021/01/24/piecewise-interpolation/easing-curves//

Animation Using javafx.animation.Transition

Basic ‘tweening animations include:

- `TranslateTransition`, `RotateTransition`, `ScaleTransition`
- `FillTransition`, `StrokeTransition`
- `FadeTransiton`: dissolve node visibility in or out
- `SequentialTransition`: run multiple transitions in a sequence
- `ParallelTransition`: run multiple transitions at the same time

Available interpolations include:

- `Interpolator.`*`LINEAR`*
- `Interpolator.`*`DISCRETE`*
- `Interpolator.`*`EASE_IN`*
- `Interpolator.`*`EASE_OUT`*
- `Interpolator.`*`EASE_BOTH`*
- custom splines

Demo: Animation/Transitions

Animation Using `javafx.animation.Transition`

```
override fun start(stage: Stage) {
    val drawable = Circle(20.0, 20.0, 20.0, Color.BLUE)
    val animation = TranslateTransition(Duration.millis(4000.0),
                                        drawable).apply {
        byY = 200.0
        interpolator = Interpolator.EASE_BOTH
        isAutoReverse = true
        cycleCount = Transition.INDEFINITE
    }
    animation.play()
    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(drawable), 320.0, 240.0)
    stage.show()
}
```

Demo: Animation/Transitions

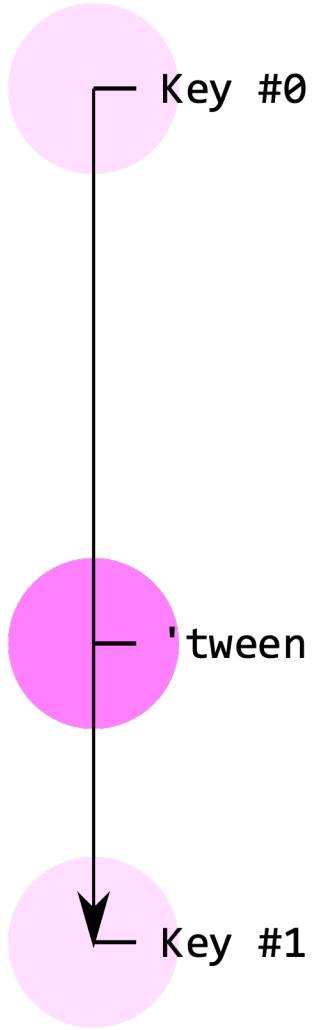
Keyframe Animation

Key Frames and Timeline

A Timeline is a sequence of two (or more) Key Frames.

A Key Frame defines an end point of a transition of one or more Key Values.

A Key Value represents an object's Property, its desired value, and the method of interpolation.



Animation Using `javafx.animation.Timeline`

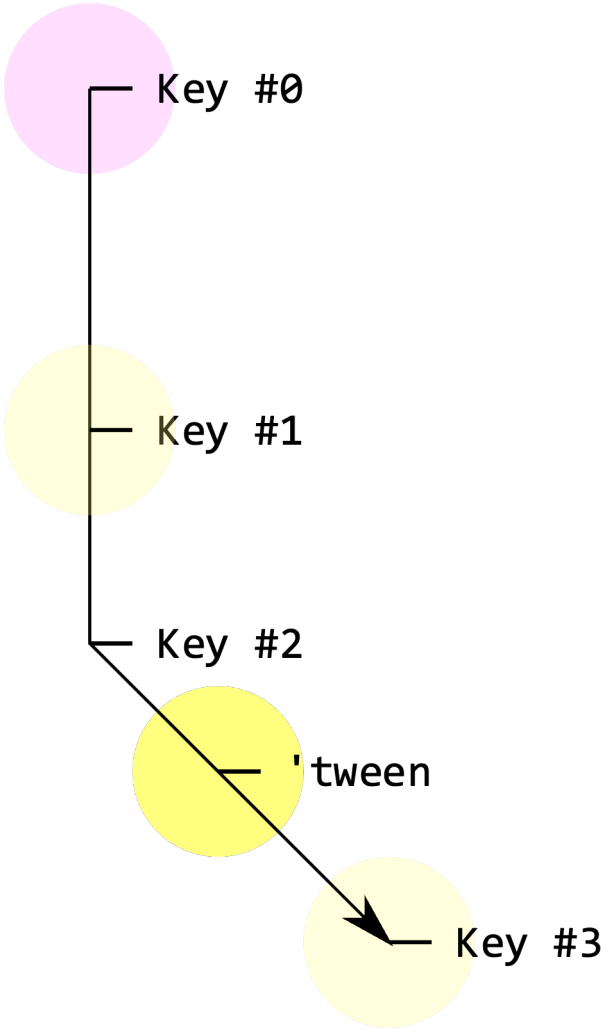
```
override fun start(stage: Stage) {
    val drawable = Circle(20.0, 20.0, 20.0, Color.BLUE)
    val animation = Timeline(KeyFrame(Duration.millis(4000.0),
                             KeyValue(drawable.translateYProperty(),
                                      200.0,
                                      Interpolator.EASE_BOTH))).apply {
        cycleCount = Animation.INDEFINITE
        isAutoReverse = true
    }
    animation.play()

    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(drawable), 320.0, 240.0)
    stage.show()
}
```

Demo: Animation/Timeline

Key Frames and Timeline

A Timeline can have many keyframes to create more complex transitions:

- Each key frame serves as a “snap shot” of one (or more) properties at a certain time.
- Timeline calculates ‘tweens, i.e., the values of each affected property for every frame.



- Key #0: `0.0s` start state (auto-generated)
- Key #1: `1.8s` `fillProperty` is `#FFFF7F`
- Key #2: `2.4s` `translateY`-property is 0.0
- Key #3: `4.0s` `translateX`-property is 200.0
  `translateY`-property is 50.0

Animation Using `javafx.animation.Timeline`

```
override fun start(stage: Stage) {
    val drawable = Circle(20.0, 20.0, 20.0, Color.BLUE)
    val animation = Timeline(                // Key #0 is auto-generated
        KeyFrame(Duration.millis(1800.0), // Key #1
                 KeyValue(drawable.fillProperty(), Color.GREEN, ...)),
        KeyFrame(Duration.millis(2400.0), // Key #2
                 KeyValue(drawable.translateXProperty(), 0.0, ...)),
        KeyFrame(Duration.millis(4000.0), // Key #3
                 KeyValue(drawables[0].translateYProperty(), 200.0, ...),
                 KeyValue(drawables[0].translateXProperty(), 50.0, ...))

        ).apply {
            cycleCount = Animation.INDEFINITE
            isAutoReverse = true
        }
    animation.play()

    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(drawable), 320.0, 240.0)
    stage.show()
}
```

Demo: Animation/Timeline

End of Chapter

Any further questions?