Inhaltsverzeichnis

Dokumentation LegLos (Kotlin-Version)

Diese technische Dokumentation gibt einen Überblick über das Zusammenspiel der Klassen in LegLos, sowie zahlreiche Code-Beispiele.

Die zentrale Metapher von LegLos ist eine Bühne (ein Fenster auf dem Bildschirm) mit Akteuren (visuelle Objekte in dem Fenster).

Wenn Entwickler*innen mit der Bibliothek arbeiten, erstellen sie ihren eigenen Code. Dieser Code wird im folgenden Client-Code genannt.

Struktur einer Show

Die Struktur einer Anwendung (einer Show) wird durch Objekte der Klassen `Stage` und `Actor` festgelegt.

Eine `Stage` ist ein visuelles Fenster, in dem `Actor`-Objekte dargestellt werden. In der Regel wird genau eine Stage angelegt:

```
val stage = Stage()
```

Die Stage wird sofort sichtbar. Aufgabe der Stage ist es, `Actor`-Objekte aufzunehmen und diese visuell darzustellen.

```
stage.addActor(actor)
```

`actor` ist ein Exemplar der `Actor`-Klasse. Die Visualisierung eines Actors erfolgt über ein Bild. Die Bilddatei wird beim Erzeugen eines `Actor` angegeben:

```
val actor = Actor('dateipfad/bilddatei.png')
```

Einige nützliche Bilder zur Darstellung von Aktoren sind als Assets Teil der Bibliothek:

```
val kodee = Actor(Assets.KODEE)
```

Eigenschaften von Actor

Ein `Actor` verwaltet den State eines Aktors. Einige der Eigenschaften sind direkt zugreifbar, andere werden von Objekten verwaltet, die stets Teil eines `Actor` sind. Zu den öffentlichen Eigenschaften von `Actor` gehören:

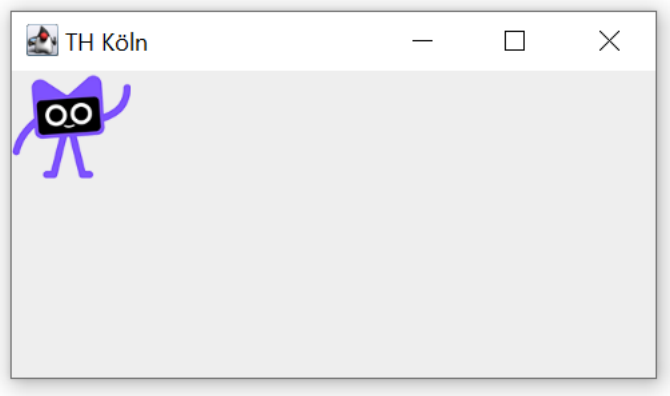
- `location` definiert die Position des `Actor` und ist vom Typ `Location`. Auf die Eigenschaften `x` und `y` kann auch direkt zugegriffen werden. Die Position wird in Pixel gemessen.
- `size` definiert die Größe des `Actor` und ist vom Typ `Size`. Auf die Eigenschaften `width` und `height` kann auch direkt zugegriffen werden. Die Größe wird in Pixel angegeben.
- `rotation` legt die visuelle Rotation des Bildes in Grad fest. Ein `Actor` kann sich im Uhrzeigersinn um 360 Grad drehen. Der Wert von `rotation` kann größer sein, es wird der Modulo-Wert verwendet: 405 und 45 sind also dieselbe Rotation. Negative Rotationswerte sind möglich, die Rotation erfolgt dann entgegen des Uhrzeigersinns.

- `opacity` legt die Deckkraft des Bildes fest. Der maximale Wert ist 100 und legt fest, dass das Bild vollständig sichtbar ist. Ein kleinerer Wert lässt das Bild transparent erscheinen, z.B. kann man bei einem Wert von 60 durch den Aktor durschauen. Der Minimalwert ist 0. In diesem Fall ist der Aktor vollständig transparent (die Deckkraft `opacity` ist 0).
- `drawBorder` ist ein boolsches Flag, ob ein Rahmen um den Aktor gezeichnet werden soll. Dies ist beim Testen einer Show hilfreich.
- `visible` legt fest, ob der Actor angezeigt wird
- `active` gibt an, ob der Actor gerade auf der Stage ist

Beispiele für das Erstellen und Konfigurieren von Aktoren:

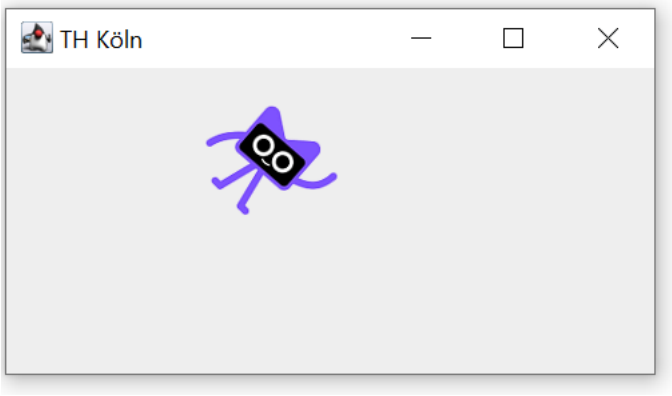
Stage erzeugen, kodee erzeugen, kodee der Stage hinzufügen:

```
val stage = Stage()
val kodee = Actor( Assets.KODEE  )
stage.addActor(kodee)
```



Die Eigenschaften von kodee lassen sich verändern:

```
kodee.x = 100
kodee.y = 20
kodee.rotation = 45
```



Komplexe Eigenschaften von Actor

Neben den atomaren Eigenschaften gibt es als Teil des `Actor` auch komplexe Objekte als Eigenschaften:

- `dragging` ist vom Typ `DragMode` und steuert das Dragging-Verhalten des Actors und legt fest, ob und wie ein 'Actor' mit der Maus verschoben werden kann. Dies geschieht über folgende Eigenschaften:
    - `enabled` schaltet das `dragging` ein
    - `restriction` schränkt das Ziehverhalten ein, wenn es eingeschaltet ist. `restriction` kann die Aufzählungswerte ´DragRestriction.NONE´, ´DragRestriction.HORIZONTAL´ und `DragRestriction.VERTICAL` festlegen.
- `appearance` legt neues Bild für den `Actor` fest. Der Eigenschaft kann ein `UIAppearance`-Objekt zugewiesen werden. Dieses Objekt sorgt automatisch dafür, dass eine Bilddatei geladen wird.
- `text` ist vom Typ `TextRendering` und ermöglicht das Rendern von Text über dem `Actor`-Bild. Es stellt folgende Eigenschaften bereit:
    - `content` legt den Textinhalt, z.B. `"Hallo Welt"`fest.
    - `xOffset` und `yOffset` definieren einen Versatz für die Positionierung des Texts innerhalb des Aktors
    - `textBackground` vom Typ `TextBackground` definiert Bilder für den Texthintergrund, z.B. Button-Layouts oder Sprechblasen. Als Wert für diese Eigenschaft kann auf Konstanten von `Assets.textbackgrounds` zugegriffen werden.
    - `icon` vom Typ `UIIconAppearance` legt ein Icon-Bild fest. Als Wert für diese Eigenschaft kann auf Konstanten von `Assets.textIcons` zugegriffen werden.
- `motion` vom Typ `Motion` steuert die Bewegung eines Actors. Details siehe unten.
- `animation` vom Typ `ActorAnimation` steuert die Animation von Eigenschaften eines Actors. Details siehe unten.
- `canvas` vom Typ `Drawable?` erlaubt es, dass auf dem `Actor`gezeichnet wird, z.B. Linienzüge oder Kreise. Details siehe unten.

Weitere Beispiele für das Setzen von Eigenschaften

```
kodee.dragging.enabled = true
kodee.motion.speed = 1
kodee.appearance = ActorAppearance(Assets.kodee.ELECTRIFIED)
```

Erzeugen eines Actors als Button oder Display

```
val btn = Actor()
btn.text.content = "Action"
btn.text.textBackground = Assets.textBackgrounds.BLUE_BUTTON
btn.location = Location(10,10)
btn.size = Size (200,40)

val display = Actor()
display.text.content = "Punkte: 0"
display.text.textBackground = Assets.textBackgrounds.GREEN_BANNER
display.text.icon = Assets.textIcons.DIAMOND
display.x = 250
display.y = 10
display.width = 200
display.height = 40

// Actors sollten erst der Bühne hinzugefügt werden,
// wenn sie fertigt konfiguiert sind.
stage.addActor(btn)
stage.addActor(display)
```

Dynamik auf der Bühne

`Stage` und `Actor` legen die Struktur statisch fest, d.h. es gibt eine `Stage` und auf dieser befinden sich alle hinzugefügten `Actor`-Objekte.

Das Geschehen auf der Bühne wird hingegen durch **Veränderungen der Eigenschaften** von `Actor` festgelegt.

Es gibt verschiedene Zeitpunkte und Möglichkeiten, wann sich die Eigenschaften von einem `Actor` verändern:

- Zu Beginn des Programmablaufs als initialies Setup
- Während des Programmablaufs:
    - `motion` verändert die `Location` des `Actor` basierend auf der bisherigen `Location`, der Bewegungsrichtung `direction`, der Schrittgeschwindigkeit `speed` und ggf. basierend auf einem Zielpunkt auf der Stage, einem `target`. Wenn `motion.running` gesetzt ist, dann geschehen diese Bewegungen automatisch nach kurzen Zeitabständen (alle 50 ms).
    - `animation` verändert Eigenschaften eines Actors über einen zeitlichen Verlauf und kann mehrere Eigenschaftsanimationen nacheinander abspielen. Eine Eigenschaftsanimation wird durch eine Implementierung eines `PropertyAnimation`-Objekts abgebildet, siehe unten.
    - als Reaktion auf Usereingaben, z.B. kann der Klick auf einen `Actor` den Zustand eines anderen `Actors` (etwa `location` oder `size`) verändern. Diese Reaktionen werden als Code-Blöcke (Lambdas) definiert und einer der `reactionFor...`-Eigenschaften zugewiesen

Reaktionen auf Usereingaben

Eine Reaktion auf eine Usereingabe wird als Lambda-Eigenschaft
festgelegt, d.h. ein `Actor` verfügt über Eigenschaften, in denen Codeblöcke als Reaktion auf eine Usereingabe definiert werden können.
Ein `Actor` hat folgende reaktive Eigenschaften als Basis:

- `reactionForMouseClick`: Reaktion auf Mausklicks (linke Maustaste)
- `reactionForRightMouseClick`: Reaktion auf Mausklicks (rechte Maustaste)
- `reactionForMousePressed`: Reaktion auf Mausdrücken
- `reactionForMouseEntered`: Reaktion auf Mauszeiger fährt in den Actor
- `reactionForMouseExited`: Reaktion auf Mauszeiger fährt aus dem Actor heraus

Eine Reaktion auf das Drücken mit der Maus kann also so festgelegt werden:

```
actor.reationForMousePressed = {
    // ausführbarer Code als Reaktion auf Mouse Pressed, z.B.
    actor.x = 100
    anotherActor.visible = true
}
```

Der Typ dieser Eigenschaften ist `()->Unit`, also der Typ einer Funktion, die keine Parameter erwartet und nichts zurückgibt. Dies entspricht gerade einem einfachen Codeblock.
Es gibt zwei weitere Reaktionen, die im Zusammenhang mit `motion` und `animation` relevant sind:

- `reactionForTargetReached`: Reaktion, wenn die automatische Bewegung von `motion` ein Ziel `target` erreicht hat
- `actor.animation.everyNsteps.reactionForTimePassed`: Reaktion auf das Verstreichen einer bestimmten Zeitspanne. Jeder `Actor` hat ein `animation`-Objekt. Dieses verfügt über ein `everyNSteps`-Objekt. In `KeyboardMotionControl.` lassen sich die `reactionForTimePassed` und `timeSpan` festlegen. `timeSpan` gibt an, wie viele Animationsschritte (jeder Schritt dauert 50ms) verstreichen müssen bis `reactionForTimePassed` ausgeführt wird. `reactionForTimePassed` wird also regelmäßig aufgerufen und kann dazu genutzt werden, den State von `Actor`-Objekten aber auch von allen anderen im Client-Code definierten Objekten in regelmäßigen Zeitabständen zu verändern. Es trägt somit zu einer zyklischen Veränderung von Eigenschaften und somit zur Animation bei.

Automatisches Bewegen

Da es sich bei `Actor`-Objekten konzeptionell um Darstellungen auf der Bühne handelt, ist das Bewegen dieser Objekte eine zentrale Funktion.

Die Bewegung eines `Actor` wird über eine `Motion`-Instanz gesteuert. Jeder `Actor` hat sein eigenes `motion`-Objekt.

Die zentrale Funktion von `Motion` ist die `move()`-Methode. Diese führt genau einen Bewegungsschritt aus.

Wenn die `Motion`-Eigenschaft `running` gesetzt ist, dann wird die `move()`-Funktion automatisch in definierten Zeitintervallen (alle 25ms) aufgerufen und somit ein Bewegungsschritt ausgeführt. Dies ist das häufigste Einsatzszenario. Es ist aber auch möglich, einen einzelnen Bewegungsschritt durch Aufruf von `move()` im Client-Code manuell anzustoßen. Dies kann sinnvoll sein, wenn man als Reaktion auf eine Usereingabe genau einen Schritt ausführen möchte, oder einen einzelnen Schritt in der Animationsliste (siehe unten) einfügen möchte.

`move()` führt einen Bewegungsschritt aus, abhängig von:

- aktueller `Location` des `Actor`
- Schrittgeschwindigkeit `speed`
- Bewegungsrichtung `direction`
- Optional Bewegungseinschränkungen `boundary`
- Optionale Neuausrichtung der Bewegungsrichtung abhängig von einem Bewegungsziel `target`

`speed` legt fest, wie weit sich der `Actor` bewegt, wenn ein Bewegungsschritt in `move()` ausgeführt wird. Wenn `speed==0` ist, dann bewegt sich der `Actor` nicht. Wenn `speed` negativ ist, dann bewegt sich der `Actor` rückwärts. Das Setzen von `speed` kann durch folgende Eigenschaften begrenzt werden:

- `negativeSpeedAllowed`: negative Werte für `speed` sind zugelassen
- `maxSpeed`: maximaler Wert für `speed`

Als Hilfsfunktionen kann mit `accelerate(delta)` und `brake(delta)` die Geschwindigkeit beschleunigt oder gebremst werden, die Veränderung wird in absoluten Werten bezogen auf die Schrittgröße festgelegt.

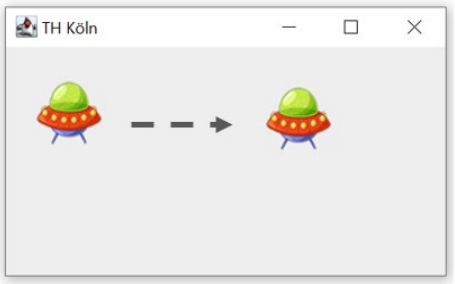
Die `direction` legt die Bewegungsrichtung in Grad fest:

- 0: Actor bewegt sich nach oben (`Direction.NORTH`)
- 90: Actor bewegt sich nach rechts (`Direction.EAST`)
- 180: Actor bewegt sich nach unten (`Direction.SOUTH`)
- 270: Actor bewegt sich nach lenks (`Direction.WEST`).

Die Orientierung von `direction` wird in Gradzahlen zwischen 0 und 360 angegeben. Negative Zahlen sind erlaubt, die Orientierung richtet sich dann entgegen des Uhrzeigersinns aus, -90 bedeutet dann, dass sich der Actor nach links bewegt. Größere Werte als 360 werden per Modulo-Operation normalisiert, d.h. aus 400 Grad werden intern 40 Grad. Die `direction` speichert den tatsächlichen Wert, also z.B. 400.

Einfaches Beispiel für Bewegung

```
val stage = Stage()
val ufo = Actor(Assets.misc.UFO , 20,20 ,50,50 )
ufo.motion.direction = Direction.EAST.orientation
ufo.motion.speed = 1
stage.addActor(ufo)
```



Die Bewegungsrichtung kann direkt als Eigenschaft gesetzt werden. Zudem lässt sich mit `rotate(r)` die Orientierung rotieren. `rotate(45)` rotiert die Richtung um 45 Grad nach rechts, `rotate(-30)` rotiert die Richtung um 30 Grad nach links

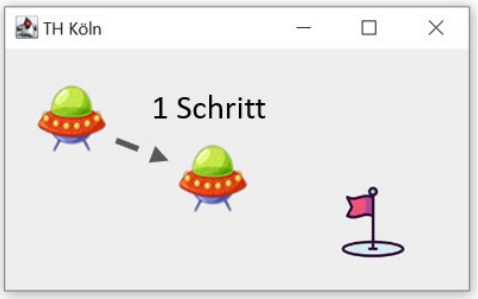
Bewegung in Richtung zu anderen Aktoren

Die Orientierung der Bewegungsrichtung lässt sich an Zielen ausrichten:

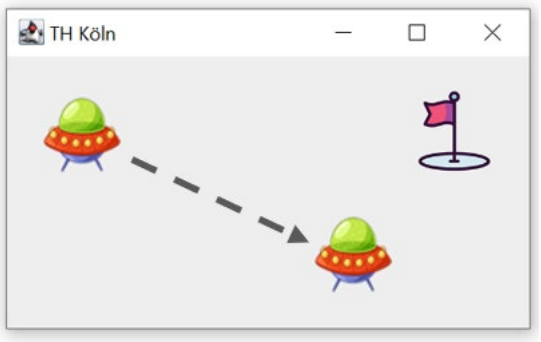
- `reorientTo (target : Actor)`: orient sich an einem anderen Actor aus, d.h. die Bewegungsrichtung ist in Richtung `target`
- `reorientTo (target : Location)`: orientiert sich an einer `Location` aus.

Der Aufruf der `reorientTo(..)`-Funktionen führt die Orientierung einmalig aus. Dies kann sinnvoll sein, wenn z.B. durch eine Reaktion auf eine Usereingabe einmalig die Bewegungsrichtung geändert werden soll.

```
val flag = Actor(Assets.misc.FLAG)
ufo.motion.running = false
ufo.motion.reorientTo(flag)
ufo.motion.move() // 1 step
```

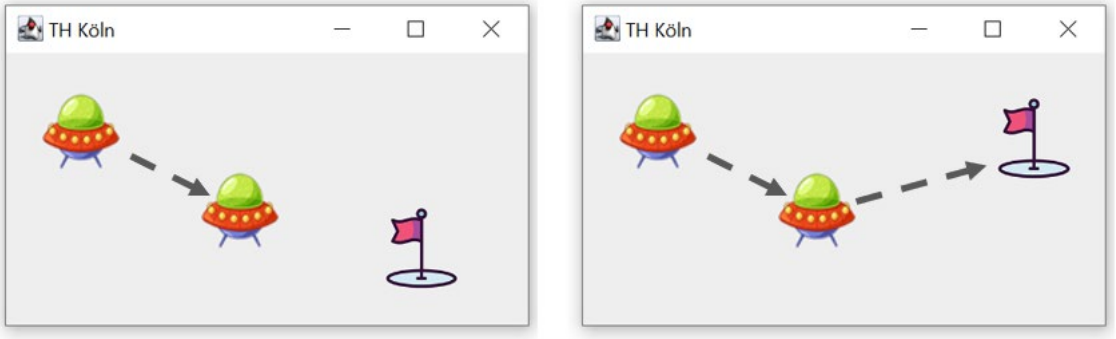


Wenn running == true gilt, dann bewegt sich das Ufo weiter in Richtung der Flagge. Dabei ändert sich die Bewegungsrichtung **nicht**, wenn die Flagge ihre Position verändert:



Wenn sich die Bewegungsrichtung fortlaufend während der Bewegung an einem Bewegungsziel orientiert soll, dann kann die Eigenschaft `target` gesetzt werden. Wenn `target` nicht null ist, dann wird zu Beginn von `move` jeweils eine Neuausrichtung durchgeführt. Zudem setzt `move()` das `target` wieder auf null sobald das Ziel erreicht wurde und löst ein `targetReached`-Event aus, d.h. der `reactionForTargetReached`-Codeblock wird ausgeführt.

```
ufo1.motion.target = ActorTarget(flag)
```



Der Typ der Eigenschaft `target` ist `Target`. `Target` ist ein Interface, das eine `Location` als Ziel zurückliefert. Derzeit gibt es zwei `Target`-Implementierungen:

- `LocationTarget`: richtet sich statisch an einer `Location` aus
- `ActorTarget`: richtet sich dynamisch nach der aktuellen `Location` eines anderen `Actor` aus.

Eine `Target`-Implementierung könnte aber auch komplexere Berechnungen durchführen, um den Zielpunkt festzulegen. Z.B. könnte als Zielpunkt der Mittelpunkt zweier Aktoren verwendet werden. Eigene Implementierungen sind erlaubt.

Motion Boundaries

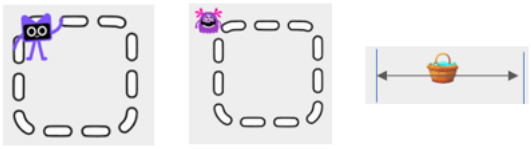
Als weitere optional setzbare Eigenschaft legt `boundary` fest, ob sich die `Location` eines `Actor` nur innerhalb eines festlegten Bereichs befinden darf. Folgende Werte kann `boundary` derzeit haben:

- `null`: es gibt keinerlei Einschränkungen
- `StageBoundary`-Objekt: der Actor muss vollständig auf der Stage sichtbar bleiben, dies ist der Defaultwert.
- `StageBoundaryExtended`: der Actor darf die Stage gerade eben verlassen, d.h. ein Actor kann außerhalb des sichtbaren Bereichs platziert werden. Interessant für Animationen, die außerhalb der Stage starten.
- `OtherActorBoundary(boundingActor)`: Bewegungsraum wird durch einen anderen Actor festgelegt. Der Actor muss innerhalb des `boundingActor` bleiben
- `OtherActorBoundaryExtended(boundingActor)`: Bewegungsraum wird durch einen anderen Actor festgelegt. Der Actor muss den boundingActor noch berühren
- `CustomBoundary`: legt minimale und maximale x,y Werte fest

Wenn `boundary` mit gesetzt ist, dann wird in der `move()`-Methode die Bewegung abgeschnitten, falls sich der `Actor` aus der Boundary heraus bewegen würde. Die `boundary` wird auch beim Ziehen eines Actors mit der Mause berücksichtigt.

Beispielcode:

```
val box1 = Actor(Assets.misc.DROP_ZONE)
val box2 = Actor(Assets.misc.DROP_ZONE)
kodee.motion.boundary = OtherActorBoundary(box1)
monster.motion.boundary = OtherActorBoundaryExtended(box2)
basket.motion.boundary = CustomBoundary (50,300,40,600)
```



```
Links: OtherActorBoundary
Mitte: OtherActorBoundaryExtended
Rechts: CustomBoundary
```

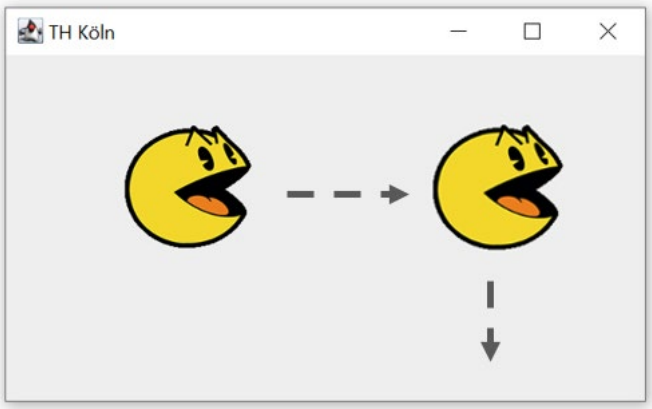
Bewegung mit Tasten steuern

`Motion` verfügt zudem über die Eigenschaft `keyMotionControl`. Mit dieser lässt sich festlegen, dass die `motion`-Eigenschaften über die Pfeiltasten der Tastatur gesteuert werden können. Die möglichen Werte sind:

- `KeyboardMotionControl.NONE`: Keine Steuerung durch die Tastatur
- `KeyboardMotionControl.MOTION_BY_ARROWS_AUTOSTOP`: Actor bewegt sich in Richtung der Pfeiltastasten, solange dies gedrückt werden
- `KeyboardMotionControl.MOTION_BY_ARROWS_CONTINUE`: Actor wechselt die Richtung entsprechend der Pfeiltaste und bewegt sich weiter, auch wenn die Tasten nicht mehr gedrückt sind
- `KeyboardMotionControl.SPEED_AND_ORIENTTATION_BY_ARROWS`: Steuert die Geschwindigkeit und Orientierung der Bewegungsrichtung mit den Pfeiltasten (hoch: beschleunigen, runter: bremsen, links: Drehung nach links, rechts: Drehung nach rechts)

Beispielcode

```
val pacMan = Actor(Assets.misc.PACMAN)
pacMan.motion.keyMotionControl=
   KeyboardMotionControl.MOTION_BY_ARROWS_CONTINUE
```



Animation von Eigenschaften

`motion` sorgt bereits dafür, dass sich Aktoren von alleine bewegen können.

Doch es gibt noch andere Eigenschaften, die animiert werden können:

- Größe
- Opacity
- Rotation

Zudem sollen manchmal komplexere Animationsabläufe festgelegt werden, z.B. im Kreis oder Rechteck laufen, zwischen den Bildschirmrändern hin und her bewegen usw.

Hierfür spielen die Klassen `ActorAnimation`, `AnimationQueue` und das Interface `PropertyAnimation` zusammen.

Jeder `Actor` besitzt die Eigenschaft `animation` vom Typ `ActorAnimation`. Diese besitzt wiederum die Eigenschaft `queue` vom Typ `AnimationQueue`. Dabei handelt es sich um eine Warteschlange für Animationsfolgen (eine Animationsliste, bei der die zuletzt hinzugefügten Animationen als letztes ausgeführt werden). Eine Animationsfolge wird durch Objekte vom Typ `PropertyAnimation` festgelegt.

Der Animationsliste können `PropertyAnimation` hinzugefügt werden:
`actor.animation.queue.add( somePropertyAnimaton )`

Die einfachste, aber sehr wichtige, Implementierung von `PropertyAnimation` ist `PropertyAnimationValueChange`

Sie legt eine Animation für eine **Eigenschaftsveränderung von einem Startwert hin zu einem Endwert** fest und gibt dabei die Anzahl der Animationsschritte an. Die folgende `PropertyAnimation` verändert die Eigenschaft `opacity` von `100` (sichtbar) auf `0` (unsichtbar) in `10` Schritten für `actor`

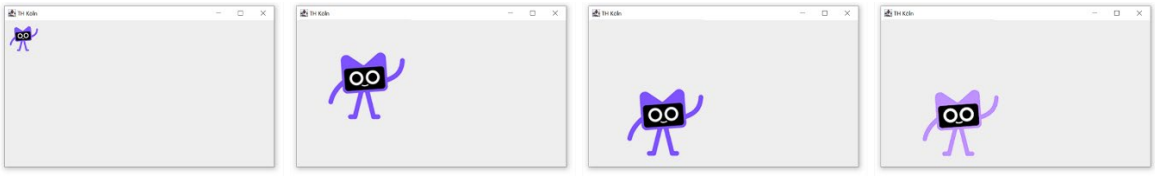
```
val somePropertyAnimaton = PropertyAnimationValueChange (
    _start =  0,
    _end = 100,
    numberOfSteps = 10,
    actor,
    AnimatableProperties.OPACITY
)
```

Folgende Eigenschaften können von `PropertyAnimationValueChange` animiert werden:

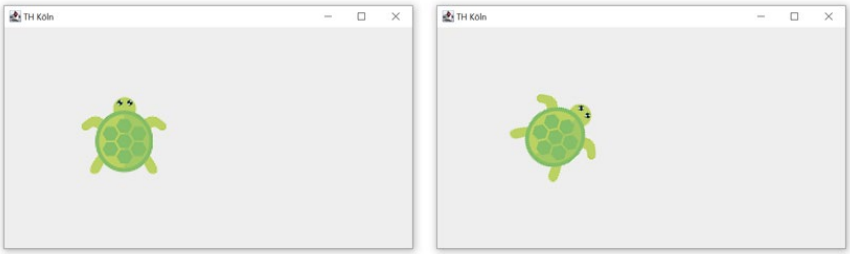
- `AnimatableProperties.ROTATION`: ändert die `rotation` Eigenschaft
- `AnimatableProperties.OPACITY`: ändert die `opacity` Eigenschaft
- `AnimatableProperties.WIDTH`: ändert die `width` Eigenschaft
- `AnimatableProperties.HEIGHT`: ändert die `height` Eigenschaft
- `AnimatableProperties.X`: ändert die `x` Eigenschaft
- `AnimatableProperties.Y`: ändert die `y` Eigenschaft

Animationsbeispiele

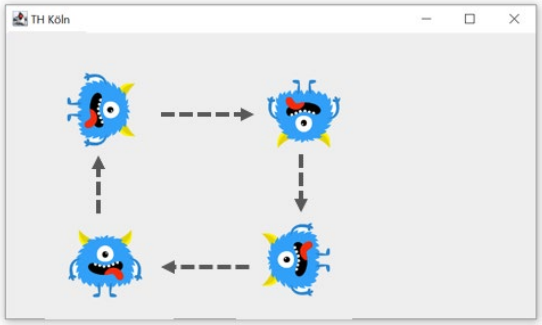
```
fun animateKodee() {
    // animates widh and height property from 50 to 200 in 10 steps for kodee
    val w_a = PropertyAnimationValueChange
         (50,200,30,kodee,AnimatableProperties.WIDTH)
    val h_a = PropertyAnimationValueChange
         (50,200,30,kodee,AnimatableProperties.HEIGHT)

    // let the animation for width and height be executed simultanously:
    val compound = ProperteryAnimationComposite(listOf(w_a,h_a))

    // animate Y thereafter
    val r_a = PropertyAnimationValueChange
         (10,100,30,kodee,AnimatableProperties.Y)

    // animate fade out and fade in by setting opacity:
    val fadeout = PropertyAnimationValueChange
        (100,0,18 ,kodee, AnimatableProperties.OPACITY)
    val fadein = PropertyAnimationValueChange
        (0,100,30 ,kodee, AnimatableProperties.OPACITY)

    // adding each of the animations into a sequence:
    val seq=PropertyAnimationSequence (listOf(compound,r_a,fadeout,fadein))

    // set start location
    kodee.location= Location (10,10)

    // add to kodee's animation queue:
    kodee.animation.queue.addPropertyAnimation(seq)


    kodee.animation.queue.addSingleActionCode {
        kodee.motion.direction = 90
    }

}
```



```
val anim = PropertyAnimationValueChange (rstart,rstart+360,
    100,turtle,AnimatableProperties.ROTATION)
turtle.animation.queue.addPropertyAnimation(anim)
```



```
// Define monster's animation as sequence of Property Animations.
// The sequence is 4x:
//    rotate 90 degrees
//    move to a target
// as a result the monster moves along a rectangle defined by the 4 targets
// PropertyAnimationSingleAction executes a lambda
// PropertyAnimationWaitForTargetReached moves actor until target is
reached
val newMotion = listOf<PropertyAnimation>(
    PropertyAnimationSingleAction({ monster.rotation = 0 } )  ,
    PropertyAnimationWaitForTargetReached(monster, LocationTarget
(200,100)) ,
    PropertyAnimationSingleAction({ monster.rotation = 90 } )  ,
    PropertyAnimationWaitForTargetReached(monster, LocationTarget
(300,100)) ,
    PropertyAnimationSingleAction({ monster.rotation = 180 } )  ,
    PropertyAnimationWaitForTargetReached(monster, LocationTarget
(300,200)) ,
    PropertyAnimationSingleAction({ monster.rotation = 270 } )  ,
    PropertyAnimationWaitForTargetReached(monster, LocationTarget
(200,200))
)

// Use the list of property animations in PropertyAnimationSequence where
// each animation is executed after another
// Wrap it in PropertyAnimationLoop to exectue the animation 4x (infinite)
val pas = PropertyAnimationLoop ( 4, PropertyAnimationSequence(newMotion) )
monster.animation.queue.addPropertyAnimation(pas)
```



Technische Details – Asynchroner Ablauf

Die `PropertyAnimation` ist ein Interface, das dem Iterator-Muster entspricht, es defniert folgende Methoden:

- `hasMoreSteps () : Boolean`: true, wenn noch Animationsschritte ausstehen
- `nextStep()`: führt den nächsten Animationsschritt aus
- `reset()`: setzt die Animationskonfiguration zurück

Die `ActorAnimation` verarbeitet die `PropertyAnimation` der `queue` wie folgt:

- Je Zeitschritt ruft `Actor.step()` die Method `ActorAnimation.step()` auf.
- `ActorAnimation.step()` schaut, ob die aktuelle Animation `currentAnimation` noch weitere Schritte hat.
    - falls ja: der nächste Schritt wird ausgeführt
    - falls nein: die nächste Animation wird aus der `queue` genommen, als `currentAnimation` gesetzt und ausgeführt

Dadurch werden alle `ProperyAnimation`-Objekte der `queue` nacheinander abgearbeitet. Die aktive `currentAnimation` wird solange ausgeführt, wie sie noch weitere Schritte hat.
Es können eigene `PropertyAnimation`-Klassen implementiert werden für komplexe Animationsabläufe.

Einige Implementierungen gibt es bereits:

- `PropertyAnimationSequence`: Führt mehrere `PropertyAnimation` **nacheinander** aus, wobei immer nur ein Schritt getätigt wird. Dies ist eine Möglichkeit, eine Reihe von Animationen hintereinander festzulegen, ohne diese einzeln der `queue` hinzuzufügen.
- `ProperteryAnimationComposite`: Führt mehrere `PropertyAnimation` **gleichzeitig** aus
- `PropertyAnimationLoop`: Führt eine `PropertyAnimation` **mehrfach** nacheinander aus, besonders nützlich für `PropertyAnimationSequence`-Objekte
- `PropertyAnimationWait`: Verzögert die Auführung der nächsten Animation, d.h. es wird `waitTicks` Mal gar nichts ausgeführt
- `PropertyAnimationWaitForTargetReached`: richtet die `motion.direction` einmalig in Richtung eines `Target` aus und wartet solange bis das Ziel erreicht wird (oder ein anderes Ziel gesetzt wird)
- `PropertyAnimationSingleAction`: führt ein Lambda-Funktion als einzigen Schritt aus, so kann können Codeblöcke nacheinander in zeitlichen Abständen ausgeführt werden

Das Ausführen von Codeblöcken in zeitlichen Abständen ist sinnvoll, um Änderungen an Eigenschaften auch sichtbar zu machen. Beispiel:

```
for (i in 1..20) {
   actor.x +=10
}
```

Der Codeblock `{actor.x += 10}` wird 20 Mal ausgeführt, am Ende ist x um 200 größer. Auf der Stage sieht man jedoch nur die Änderung von 200, denn die einzelnen Wertänderungen werden sofort und sehr schnell durchgeführt.

Was der Client-Code eigentlich erreichen wollte, ist die Veränderung von `x` in 20 Animationsschritten. Dazu ist es notwendig, die einzelnen Veränderungen der `queue` hinzuzufügen:

```
for (i in 1..20) {
   actor.animation.queue.add (
     PropertyAnimationSingleAction( {actor.x +=10} )
   )
}
```

Jetzt wird der Codeblock `{actor.x += 10}` in 20 mal in die `queue` eingefügt und zeitlich verzögert ausgeführt, so dass die 20 Schritte auch sichtbar werden.

Da dies ein häufiger Anwendungsfall ist, bietet `AnimationQueue` auch direkt eine Methode `addSingleActionCode`, um einen Codeblock in die Warteschlange einzufügen:

```
 for (i in 1..20) {
   actor.animation.queue.addSingleActionCode (  {actor.x +=10} )
   )
}
```

Auch für `PropertyAnimationWait` gibt es eine solche Hilfsfunktion:

```
actor.animation.queue.addWait(10)
```

Turtle-Steuerung

Eine weitere nützliche Möglichkeit, vorkonfigurierte Codeblöcke in die `queue` zu stellen bietet die `turtleControl` vom Typ `Turtle`.

Sie stellt Methoden bereit, um einzelne Animationschritte in die `queue` zu stellen, z.B. `actor.animation.turtleControl.forward(20)`

Der Aufruf bewegt den `actor` um 20 Schritte vorwärts in die aktuelle Bewegungsrichtung `actor.motion.direction`. `turtleControl` lässt einen `Actor` wie eine Schildkröte bewegen: vorwärts, rückwärts, nach links oder rechts drehen. Wer Scratch oder Logo kennt, ist mit dieser Art der Bewegung von Figuren vertraut.

`Turtle` definiert folgende Methoden, um einen einzelnen Bewegungsschrit in die `queue` einfügen:

- `forward(steps)`: Actor bewegt sich `steps` Schritte vorwärts
- `backward(steps)`: Actor bewegt sich `steps` Schritte rückwärts
- `turnRight(degree)`: Actor dreht sich `degree` Grad im Uhrzeigersinn nach rechts
- `turnLeft(degree)`: Actor dreht sich `degree` Grad entgegen des Uhrzeiersinns nach links
- `turnRightAndMoveForward(degree,steps)`: Actor dreht sich nach rechts und bewegt sich danach vorwärts in einem Schritt
- `turnRightAndMoveBackward(degree,steps)`: Actor dreht sich nach links und bewegt sich danach vorwärts in einem Schritt
- `moveUpBy(steps)`: Actor bewegt sich `steps` Schritte hoch, unabhängig von der aktuellen Bewegungsrichtung
- `moveDownBy(steps)`: Actor bewegt sich `steps` Schritte runter, unabhängig von der aktuellen Bewegungsrichtung
- `moveLeftBy(steps)`: Actor bewegt sich `steps` Schritte nach links, unabhängig von der aktuellen Bewegungsrichtung
- `moveRightBy(steps)`: Actor bewegt sich `steps` Schritte nach rechts, unabhängig von der aktuellen Bewegungsrichtung
- `moveToRandom()`: Actor bewegt sich zu einer zufälligen Positio
- `wait(ticks)`: Es dauert `ticks` Zeiteinheiten bis die nächste Animation ausgeführt wird

`turnRightAndMoveForward` und `turnRightAndMoveBackward` ermöglichen es, dass eine Rotation und eine Bewegung in einem Schritt ausgeführt werden. Der Aufruf von `turnRight(45)` und `forward(20)` würde sonst zwei Animationsschritte verursachen. Dies führt bei aneinander gereihten Schritten zu Ruckeln, da abwechselnd bewegt und rotiert wird.

Komplexe Bewegungsabläufe lassen sich über `turtleControl` festlegen:

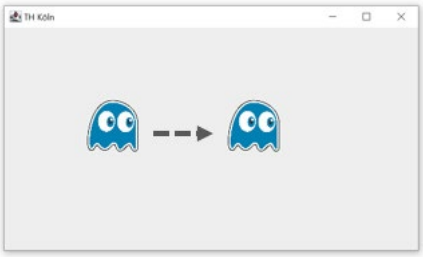
```
actor.animation.turtleControl.moveToRandom()
for (i in 1..10) {
  actor.animation.turtleControl.forward(20)
  actor.animation.turtleControl.turnRight(45)
  actor.animation.turtleControl.moveDownBy(10)
}
```

Etwas eleganter geht es mit dem Kotlin Schlüsselwort with:

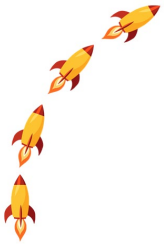
```
with (actor.animation.turtleControl) {
    moveToRandom()
    for (i in 1..10) {
        forward(20)
        turnRight(45)
        moveDownBy(10)
    }
}
```

Beispiele

```
ghost.turtleControl.moveRightBy(30)
```



```
for (i in 1..72) {
   spacecraft.turnRightAndMoveForward(10,5)
}
```



Geometrische Events

Ein geometrisches Event tritt auf, wenn zwei Aktoren sich geometrisch überschneiden, oder sich ein Actor in einem anderen Actor befindet. Es gibt folgende Zustände, die zu Events führen:

- Intersection: Actor 1 und Actor 2 überschneiden sich
- Contained: Actor 1 befindet sich vollständig in Actor 2
- Drop: Actor 1 wurde per Maus gezogen und beim Loslassen befindet sich Actor 1 vollständig in Actor 2

Dabei gibt es ein start-Event, wenn der Zustand beginnt und ein stop-Event, wenn der Zustand endet.

Um auf diese Events zu reagieren, können eigene `EventListener` implementiert werden. Dabei handelt es sich um Schnittstellen, die jeweils eine Methode vorgeben, wenn der Zustand startet und endet. Schauen wir uns dies als Beispiel für Intersection an. Das Interface sieht so aus:

```
interface IntersectionEventListener {
  fun onStartIntersection()
  fun onStopIntersection()
}
```

`onStartIntersection` soll definieren, was passiert, wenn die Überschneidung beginnt. `onStopIntersection` soll definieren, was passiert, wenn die Überschneidung endet.

Eine Implementierung kann so aussehen:

```
    val boomActor = Actor(Assets.BOOM)
    val myIntersectionListener = object : IntersectionEventListener {

       private var intersectionCount  = 0

        override fun onStartIntersection() {
            boomActor.visible = true
            intersectionCount++
        }

        override fun onStopIntersection() {
            boomActor.visible = false
        }
    }
```

`myIntersectionListener` implementiert das Interface wie folgt:

- `onStartIntersection`: macht einen `boomActor` (z.B. eine "Boom"-Sprechblase) sichtbar
- `onStopIntersection`: macht den gleichen `boomActor` wieder unsichtbar

Es handelt sich dabei um ein reguläres Objekt (oder eine reguläre Klassendefinition), welches dem Interface genügt. Das Objekt kann also optional auch einen eigenen Zustand besitzen. Im Beispiel ist dies die

Eigenschaft `intersectionCount`. Sie zählt, wie häufig onStartIntersection() aufgerufen wurde.

Nun müssen wir noch festlegen, für welche Aktoren die Überschneidung zu dem Event führen soll. Jeder Actor hat die Methode `addIntersectionEventListener`. Mit dieser lässt ein `IntersectionListener` für den Actor hinzufügen. Dabei muss man zudem angeben, mit welchem weiteren Actor die Überschneidung zu einem Event führen soll. Folgender Code legt fest, dass die Methoden von `myIntersectionListner` ausgeführt werden, wenn `actor1` und `actor2` sich überschneiden bzw. nicht mehr überschneiden.

```
actor1.addIntersectionEventListener(actor2,myIntersectionListener)
```

Einem Actor können mehrere `IntersectionListener` hinzugefügt werden. Dadurch können unterschiedliche Reaktionen auf die Überschneidung mit anderen Aktoren festgelegt werden. Eine Intersection ist ähnlich wie eine Kollision, mit dem Unterschied, dass ein Actor bei schneller Geschwindigkeit über einen anderen Actor "hinwegspringen" kann, d.h. es kommt gar nicht erst zu keiner Überschneidung.

Für Contained-Events definiert das `ContainedListener`-Interface die Methoden `onStartContainment()` und `onStopContainment()`. Der Listener wird so hinzugefügt:

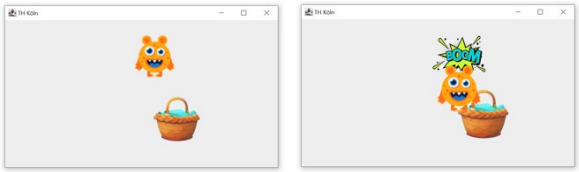
```
actor1.addContainedEventListener(actor2,myIntersectionListener)
```

Das Ereignis wird dann ausgeführt, actor1 sich vollständig in actor2 befindet (bzw. nicht mehr).

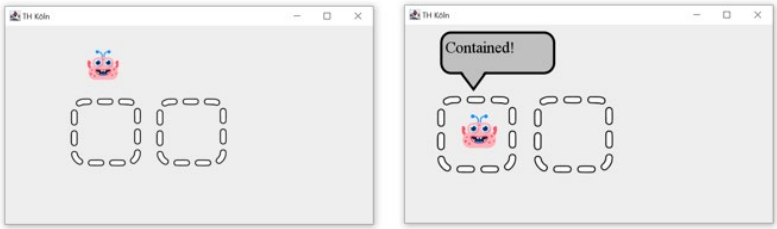
Der `DropEventListener` mit seinen Methoden `onDropIn` und `onDropOut` funktioniert ähnlich wie der `ContainedListener`. Es gibt jedoch zwei Unterschiede. Die Überprüfung, ob `actor1` sich in `actor2` befindet, findet nur statt, wenn `actor1` mit der Maus gezogen und losgelassen wird. Das Event tritt also ein, wenn `actor1` über `actor2` fallen gelassen (drop in) wurde oder wieder heraus gezogen (drop out) wird. Zudem definiert das Interface noch die Eigenschaft `alignWhenDropped`. Diese muss von Implementierungen definiert werden. Sie gibt an, ob beim Loslassen von `actor1` über dem Ziel `actor2` der gezogene `actor1` mittig im Ziel platziert werden soll. Dies ist beim Droppen über einer Box oft gewünscht.

Beispiele

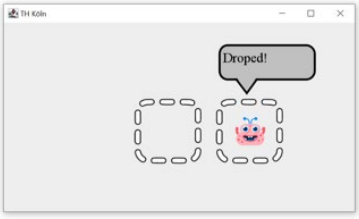
```
val basket = Actor(Assets.misc.BASKET)
val dragMonster = Actor(Assets.monster.MONSTER6)
val boom = Actor(Assets.bubbles.BOOM)


dragMonster.addIntersectionEventListener(
   basket,
   object: IntersectionEventListener {
       override fun onStartIntersection() {
           boom.location = dragMonster.location
           boom.visible  = true
       }

       override fun onStopIntersection() {
           boom.visible = false
       }
   }
)
```



```
val drop1 = Actor(Assets.misc.DROP_ZONE)
val dropMonster = Actor (Assets.monster.MONSTER8)
val bubble = Actor()

drop1.addContainedEventListener(
    dropMonster,

    object : ContainedEventListener  {
        override fun onStartContained() {
            bubble.location = Location(20,170)
            bubble.text.content = "Contained!"
            bubble.visible = true
        }

        override fun onStopContained() {
            bubble.visible = false
        }
    }
)
```



```
val drop2 = Actor(Assets.misc.DROP_ZONE)
val dropMonster = Actor (Assets.monster.MONSTER8)
val bubble = Actor()
dropMonster.addDropEventListener(
    drop2 ,
    object : DropEventListener {
        override fun onDropIn() {
            bubble.location = Location(140,170)
            bubble.text.content = "Droped!"
            bubble.visible = true

        }

        override fun onDropOut() {
            bubble.visible = false
        }

        override val alignWhenDropped: Boolean = true

    }
)
```



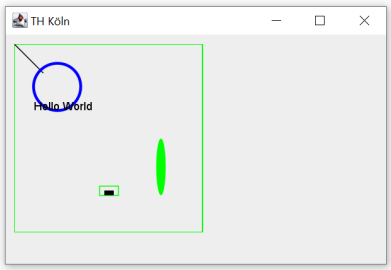
Zeichnen

Auf einem `Actor` kann gezeichnet werden. Hierzu wird die `canvas`-Eigenschaft mit einem `Drawable`-Objekt gesetzt. `Drawable` ist ein Interface. Es wird implementiert von

- Klassen oder Objekten, deren primäre Aufgabe es ist, etwas zu zeichnen (z.B. Diagramme)
- Klassen, die ihren Zustand ähnlich wie mit `toString()` noch als Zeichnung darstellen möchten

`Drawable` definiert eine Methode `draw(pen:Pen)`. Sie wird beim Rendern des Actors als letzes aufgerufen und erhält ein `Pen`-Objekt als Parameter.
Implementierungen von `Drawable`-können das `Pen`-Objekt nutzen, um zu zeichen:

```
actor.canvas = object : Drawable {
    override fun draw(pen: Pen) {
        pen.drawLine(0,0,30,30)
        pen.color= PenColor.BLUE
        pen.stroke = PenThickness.MEDIUM
        pen.drawOval(20,20,50,50)
        pen.stroke = PenThickness.THIN
        pen.color = PenColor.GREEN
        pen.fillOval(150,100,10,60)
        pen.drawRect(90,150,20,10)
        pen.color = PenColor.BLACK
        pen.fillRect(95,155,10,5)
        pen.drawString("Hello World",20,70,)
    }
}
```



`Pen` besitzt folgende Methoden zum Zeichnen:

- `drawLine(x1,y1,x2,y2)`: zeichnet eine Linie von `x1,y1` nach `x2,y2`
- `drawOval(x,y,width,height)`: zeichnet ein Oval an der Stelle `x,y` mit der Breite `width` und der Höhe `height`
- `fillOval(x,y,width,height)`: zeichnet gefülltes Oval an der Stelle `x,y` mit der Breite `width` und der Höhe `height`
- `drawRect(x,y,width,height)`: zeichnet ein Rechteck an der Stelle `x,y` mit der Breite `width` und der Höhe `height`
- `fillRect(x,y,width,height)`: zeichnet gefülltes Rechteck an der Stelle `x,y` mit der Breite `width` und der Höhe `height`
- `pen.drawString(str,x,y)`: zeichnet die Zeichenkette str an der Stelle `x,y`

`Pen` besitzt folgende Eigenschaften, mit denen die nachfolgenden Zeichnungen verändert werden:

- `color`: Farbe, z.B. `PenColor.BLUE` oder `PenColor.BLACK`
- `stroke`: Strichstärke,
  entweder `PenThickness.THIN`, `PenThickness.MEDIUM` oder `PenThickness.THICK`

Packages im Überblick

Packages mit Klassen, mit denen Client-Code primär in Berührung kommt:

- `motion`: Alles zur automatichen Bewegung des `Actor`
- `animation`: Alles für die Animation von Eigenschaften
- `ensembles`: Vorgefertigte Sets mit `Actor`-Objekten zum Testen und für Demos
- `coordinatesystem`: `Location` `Size` und globale Konstante
  in `WorldConstants` (Stage-Größe)
- `drawing`: Zeichnen auf dem Actor
- `geoevents`: Listener und Handler für geometrische Events
- `imageprovider`: Verwaltet und lädt Bilder, Assets bereitstellen

Packages, die vor allem Framework (hier: Swing) spefische Klassen und Adapter bereitstelln

- `controller`: MouseEvents, Keystroke-Events. Swing-Klassen!
- `uicomponents`: Repräsentation der visuellen Elemente. Swing-Klassen!