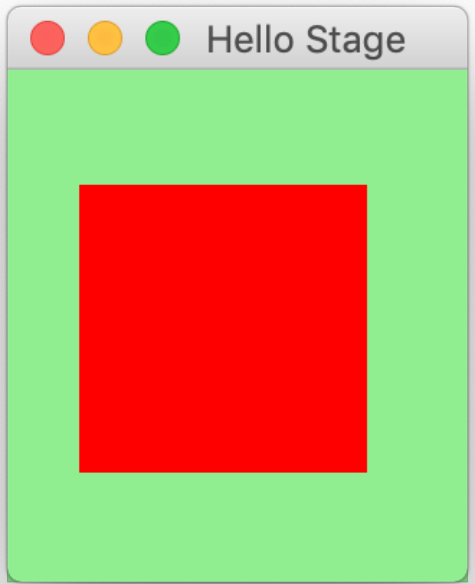
ScalaFX

```
object HelloStageDemo extends JFXApp {
  stage = new PrimaryStage {
    title = "Hello Stage"
    width = 160
    height = 200
    scene = new Scene {
      fill = LightGreen
      content = new Rectangle {
        x = 25
        y = 40
        width = 100
        height = 100
        fill = Red
      }
    }
  }
}
```



- `build.sbt`-sse lisada:

```
libraryDependencies += "org.scalafx" %% "scalafx" % "8.0.144-R12"
```

- Programm laiendab `JFXApp` trait-i.
- Kood kirjutada konstruktorisse, mitte `main`-i v.m.s.
- Kasutab `x_=(...)` meetodeid, näiteks

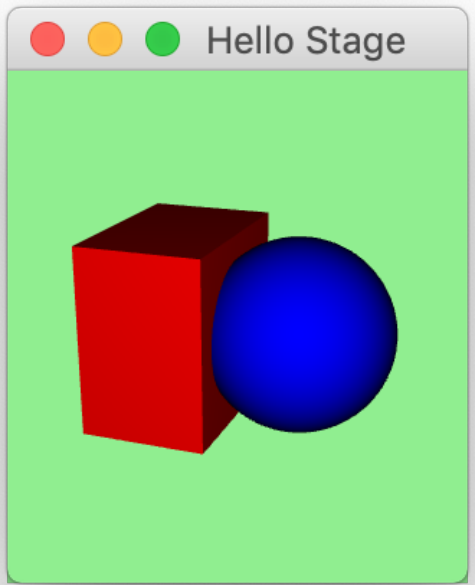
```
title = "Hello Stage"   ==>   title_=("Hello Stage")
```

Arhitektuur

- Stage – Aken ise
- Scene – Akna sisu
- Node – Akna element
  - Text
  - Shape (Line, Rectangle, . . .)
  - Chart (Bar, Pie, Line, Area, Bubble, Scatter)
  - Pane (VBox, HBox, StackPane, TilePane,GridPane . . .)
  - Control (Button, Lable, TreeView, TextArea . . .)
  - Canvas
  - ImageView
  - WebView

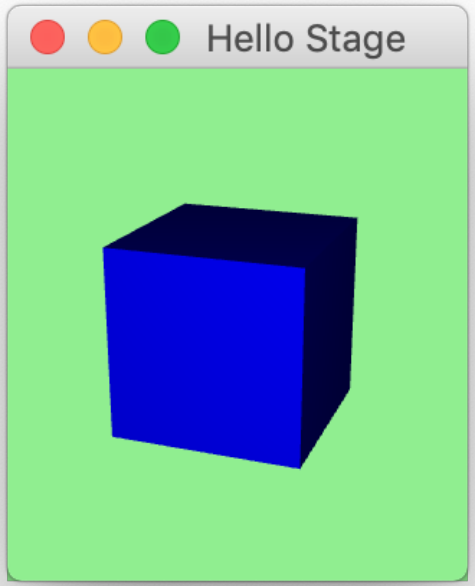
Shape3d

```
content = Seq(
  new Box() {
    transforms = Seq(new Translate(-1, 0, 0))
    material = new PhongMaterial(Color.Red)
    height = 3; width = 2; depth = 3
  },
  new Sphere() {
    radius = 1.5
    transforms =
      Seq(new Translate(1, 0, 0))
    material =
      new PhongMaterial(Color.Blue)
  }
)

camera = new PerspectiveCamera(true) {
  transforms =
    Seq(new Rotate(-20, Rotate.YAxis), new Rotate(-20, Rotate.XAxis),
      new Translate(0, 0, -15))
}
```

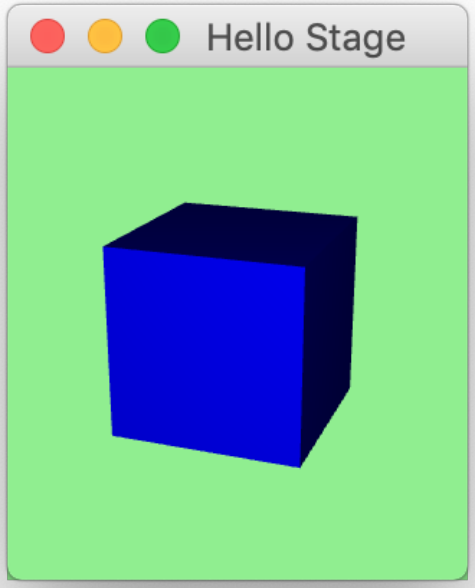


Events

```
content =
  new Box() {
    material =
      new PhongMaterial(Color.Red)
    onMouseClicked = { _ =>
      material =
        new PhongMaterial(Color.Blue)
    }
    height = 3
    width  = 3
    depth  = 3
  }
```

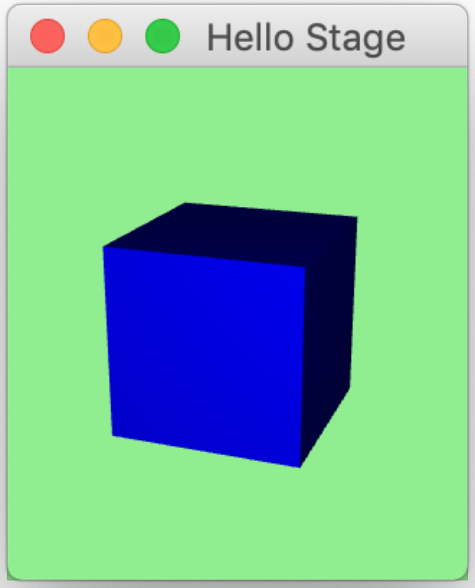


Property



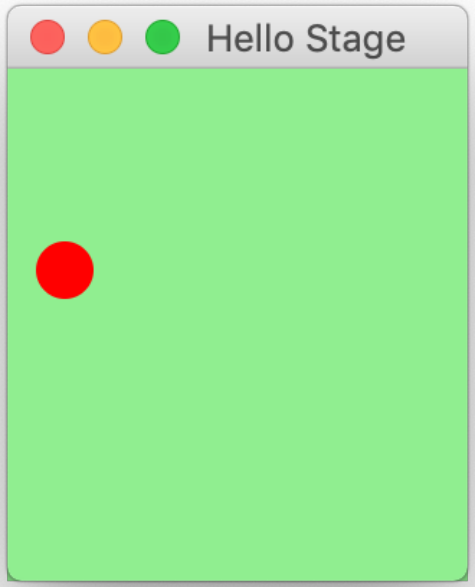

```
content =
  new Box() {
    material <==
      (when(hover)
        choose new PhongMaterial(Color.Red)
        otherwise new PhongMaterial(Color.Blue))

    height = 3; width = 3; depth = 3
  }
```

Animatsioonid




```
content =
  new Sphere() {
    material = new PhongMaterial(Color.Red)
    radius = 3
    onMouseClicked = { _ =>
      Timeline(at(3 s){radius -> 1}).play()
    }
  }
```

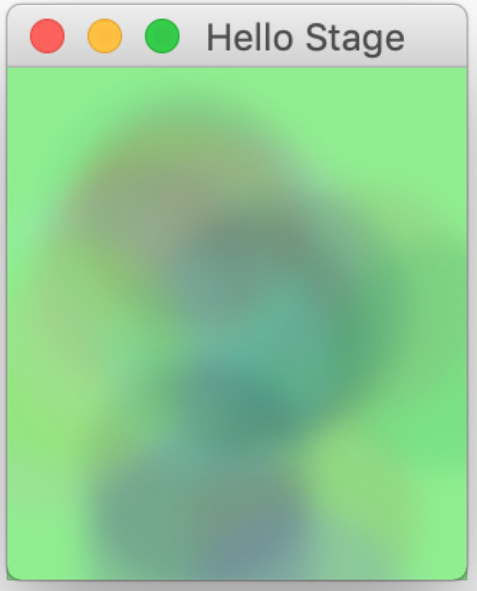
Animatsioonid 2




```
scene = new Scene {
  fill = LightGreen

  private val c = new Circle {
      centerX = 20
      centerY = 70
      radius = 10
      fill = Red
    }

  content = c

  onMouseClicked = { _ =>
    Timeline(at(0.5 s){c.centerY -> (height.value - 5)}).play()
  }
}
```

Animatsioonid 4

```
val cs = for (i <- 0 to 20) yield new Circle {
  centerX = random * 160
  centerY = random * 200
  radius = 40
  fill = color(random, random, random, 0.2)
  effect = new BoxBlur(10,10,3)
  onMouseClicked = handle {
    Timeline(at(3 s) {radius -> 0}).play()
  }
}
new Timeline{
  cycleCount = Timeline.Indefinite
  autoReverse = true
  keyFrames = for (c <- cs) yield at(20 s) {
    Set[KeyValue[_, _ <: Object]](
      c.centerX -> random * 160,
      c.centerY -> random * 200)}
}.play()
```



DelayedInit

Klassid ja objektid, mis pärivad `DelayedInit`, muudetakse nii: `code` $\Longrightarrow$ `delayedInit(code)`. S.t

```
trait C1 extends DelayedInit {
  println("C1 initsialiseerimine")
  def delayedInit(body: => Unit): Unit = {
    println("enne C2 initsialiseerimist")
    body       // C2 initsialiseerimine
    println("peale C2 initsialiseerimist")
  }
}

class C2 extends C1 {
  println("C2 initsialiseerimine")
}

object Test {
  def main(args: Array[String]): Unit = {
    val c = new C2
  }
}
```

App

`DelayedInit` kasutatakse ka trait-i `App` poolt.

```
trait App extends DelayedInit { // mõned detailid eemaldatud
  private val initCode = new ListBuffer[() => Unit]

  override def delayedInit(body: => Unit) {
    initCode += (() => body)
  }

  def main(args: Array[String]) = {
    for (proc <- initCode) proc()
  }
}

object Test extends App {
  val c = new C
}
```

Property

- Erinevad tüüpi omadused: `BooleanProperty`, `DoubleProperty`, `FloatProperty`, `IntegerProperty`, `LongProperty`, `StringProperty` ja `ObjectProperty`.
- Saab ka ise teha:

```
val speed1 = DoubleProperty(55)
val speed2 = new DoubleProperty(this, "speed2", 55)
```

- Seotud sündmuste käsitlemisega:

```
prop.onChange { (source, oldValue, newValue) => doSomething() }
```

- Saab omavahel ühendada: `a <== b`, `a <==> b`
- ... veel näiteid:

```
area <== base * height / 2
prop <== when(cond) choose(value1) otherwise(value2)
```

Property

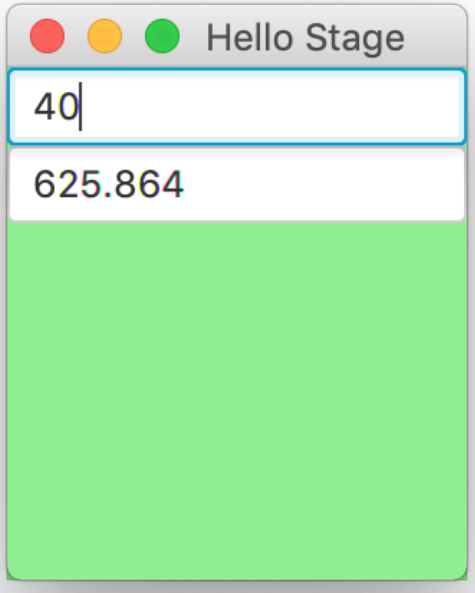
- *Functional Reactive Programming* ehk FRP

- *Observer pattern*: registreerin vaatleja muutuva väärtuse A juurde, mis muudab väärtust B.
- FRP: registreerin B juurde seose, et ta võtaks väärtuse A-st.

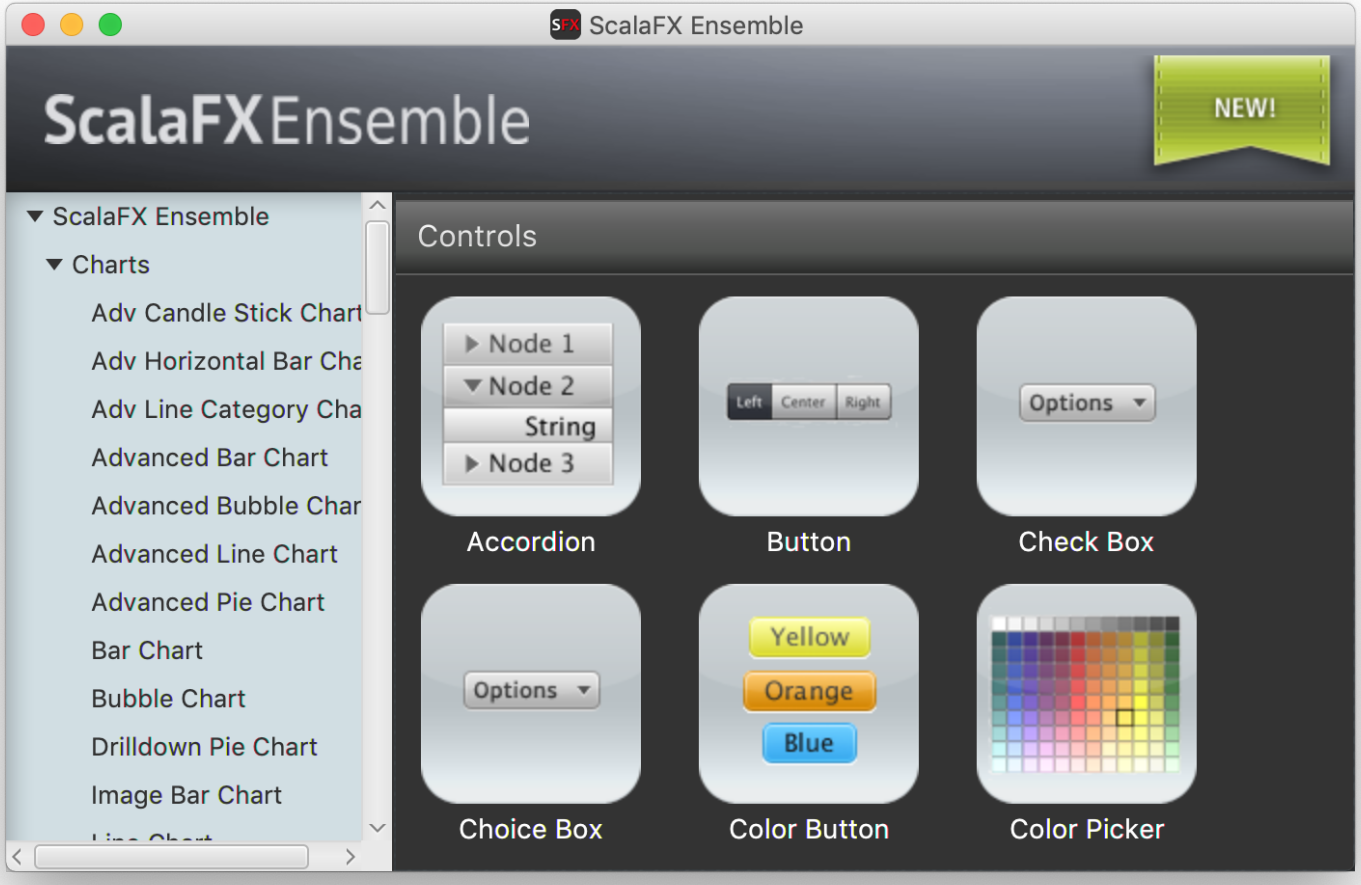
- Sobib lihtsate omaduste sidumiseks.
- Ei saa sõltuda iseenda eelmisest väärtusest.

Omadused 2

```
scene = new Scene {
  val iF = new TextField(){
    maxWidth = 160
    text = "1"
  }

  val input1 = createIntegerBinding(
    () => Try(iF.text.value.toInt).getOrElse(0),
    iF.text)

  val outputText = new TextField(){
    maxWidth = 160
    text <== (input1 * 15.6466).asString()
  }

  content = new VBox(iF, outputText)
}
```



ScalaFX - Ensemble



Mitmelõimelisus

- Igal klassil (monitoril) on järgnevad meetodid:

```
def synchronized[A] (e: => A): A
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

- Synchronized argument täidetakse teisi lõimi välistavalt: kaks lõime ei saa samaaegselt täita sama monitori *synchronized* koodi.

- Enamasti oodatakse mingi tingimuse C täitumist:

```
while (!C) wait()
```

Näide: BoundedBuffer

```
class BoundedBuffer[A](N: Int) {
  var in = 0, out = 0, n = 0
  val elems = new Array[A](N)

  def put(x: A) = synchronized {
    while (n >= N) wait()
    elems(in) = x ; in = (in + 1) % N ; n = n + 1
    if (n == 1) notifyAll()
  }

  def get: A = synchronized {
    while (n == 0) wait()
    val x = elems(out) ; out = (out + 1) % N ; n = n - 1
    if (n == N - 1) notifyAll()
    x
  }
}
```

Näide: BoundedBuffer

- BoundedBuffer kasutamine

```
import scala.concurrent.ops._
...
val buf = new BoundedBuffer[String](10)
spawn { while (true) { val s = produceString ; buf.put(s) } }
spawn { while (true) { val s = buf.get ; consumeString(s) } } }
```

- Spawn definitsioon

```
def spawn(p: => Unit) {
  val t = new Thread() { override def run() = p } t.start()
}
```

Sünkroniseeritud muutujad: SyncVar

```
class SyncVar[A] {
  var isDefined: Boolean = false
  var value: A = _

  def get = synchronized {
    while (!isDefined) wait()
    value
  }

  def set(x: A) = synchronized {
    value = x; isDefined = true; notifyAll()
  }

  def isSet: Boolean = synchronized {
    isDefined
  }
  def unset = synchronized {
    isDefined = false
  }
}
```

Tulevikuväärtused: `future`

- Väärtus, mida arvutatakse teises lõimes.

```
import scala.concurrent.ops._
...
val x = future(pikk_arvutus)
teine_pikk_arvutus
val y = f(x()) + g(x())
```

- `future` on defineeritud nii:

```
def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  fork { result.set(p) }
  (() => result.get)
}
```

Mitmelõimelisus

- Paketis `scala.concurrent` veel võimalusi: `Promise`, `Channel`, `Lock` jne.
- Keerukust aitab vähendada sobivama arvutusmudeli valimine: `Actorid`.
  - Sõnumite edastamisel põhinev mudel.
  - Aktorid reageerivad sissetulevale sõnumile, seejärel võivad teha kohalikke arvutusi ja omakorda saata sõnumeid.
  - Hea implementatsioon: Akka
  - Aktoreid saab viia ka teise arvutisse.