DISTILLING JAVA LIBRARIES

Zach Tellman
@ztellman

impedance mismatches

- mutable state
- objects conflate data and actions

design patterns are missing language features

- first-class functions
- closures
- simple data literals

distillation

distillation

- doesn’t mean everything is lower-case and hyphenated
- .camelCase isn’t a code smell

distillation

- getting at the idea behind the code
- aligning **structure** with **intent**

intent is subjective

what matters is **your** intent

to distill code, you must write more code

make sure it’s worth it

reasons to distill

- reducing incidental complexity
- reducing scope
- creating a gestalt

understanding intent is a process

- intuition guides creation
- creation hones intuition
- lather, rinse, recur

libraries are a vocabulary

“The difference between the right word and the almost right word is the difference between lightning and a lightning bug.”

- Mark Twain

Java2D

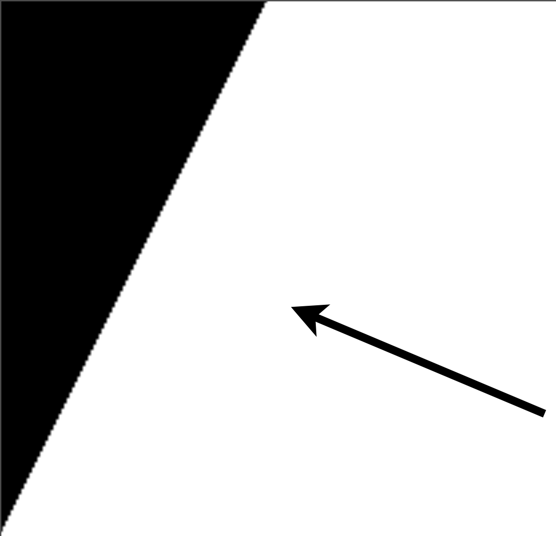
- unavoidable side-effects
- well designed
- not boring

play along at home

http://github.com/ztellman/scrawl

```
(defn draw-triangle [graphics]
  (.fillPolygon graphics
    (int-array [-1 0  1])
    (int-array [-1 1 -1])
    3))

(defn create-panel []
   (proxy [JPanel] []
     (paint [graphics]
       (draw-triangle graphics)))))
```

```
(defn draw-triangle [graphics]
  (.fillPolygon graphics
    (int-array [-1 0  1])
    (int-array [-1 1 -1])
    3))

(defn create-panel []
   (proxy [JPanel] []
     (paint [graphics]
       (draw-triangle graphics))))
```



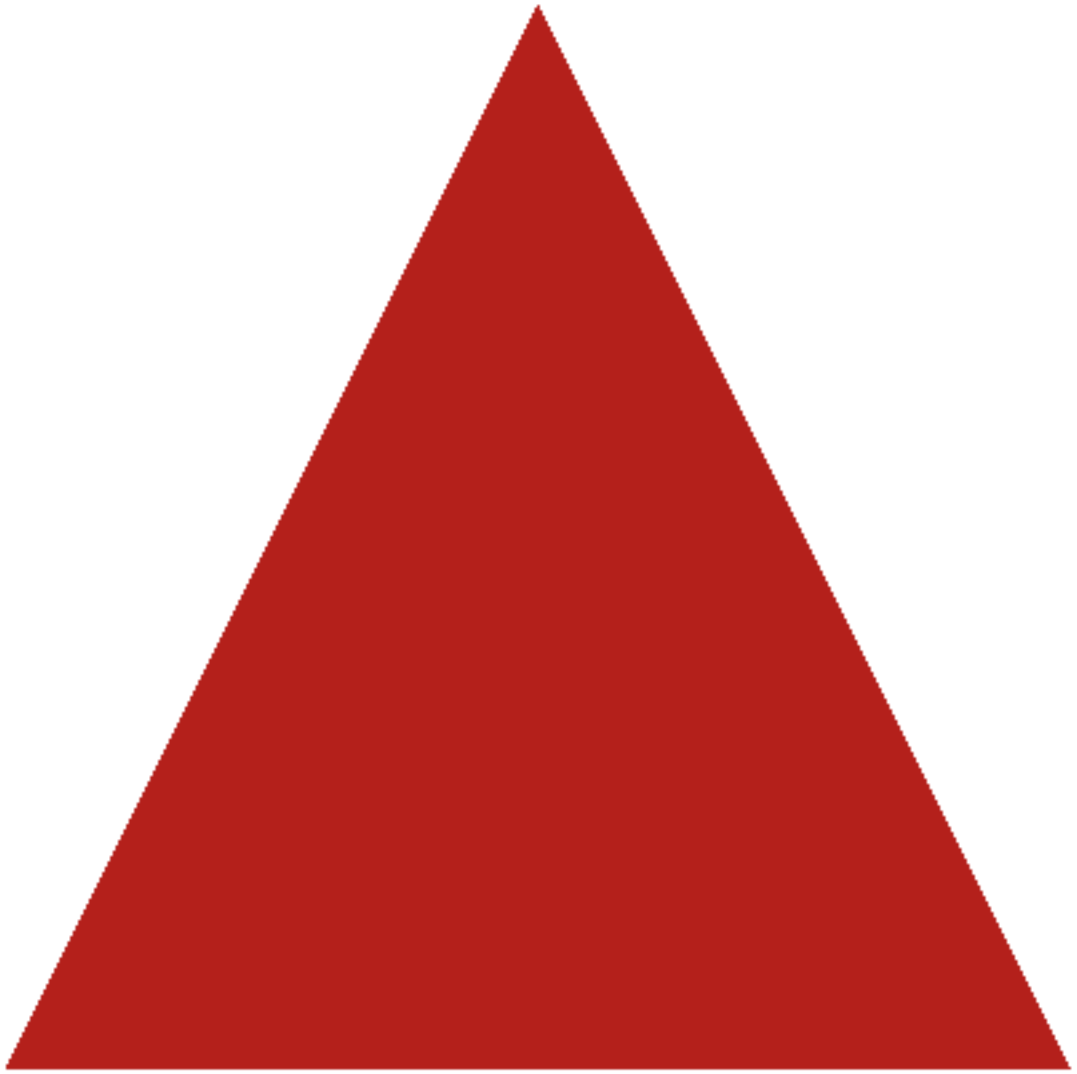
not to scale

```
(defn create-panel []
   (proxy [JPanel] []
     (paint [graphics]

       (.translate graphics
         (/ (.getWidth this) 2)
         (/ (.getHeight this) 2))

       (.scale graphics 200 -200)

       (.setColor graphics Color/RED)

       (draw-triangle graphics))))
```

```
(defn create-panel []
   (proxy [JPanel] []
     (paint [graphics]

       (.translate graphics
         (/ (.getWidth this) 2)
         (/ (.getHeight this) 2))

       (.scale graphics 200 -200)

       (.setColor graphics Color/RED)

       (draw-triangle graphics))))
```

```
(defn create-panel []
   (proxy [JPanel] []
     (paint [graphics]

       (.translate graphics
         (/ (.getWidth this) 2)
         (/ (.getHeight this) 2))

       (.scale graphics 200 -200)

       (.setColor graphics Color/RED)

       (draw-triangle graphics))))
```

```
(defn create-panel []
   (proxy [JPanel] []
     (paint [graphics]

       (.translate graphics
         (/ (.getWidth this) 2)
         (/ (.getHeight this) 2))

       (.scale graphics 200 -200)

       (.setColor graphics Color/RED)

       (draw-triangle graphics))))
```



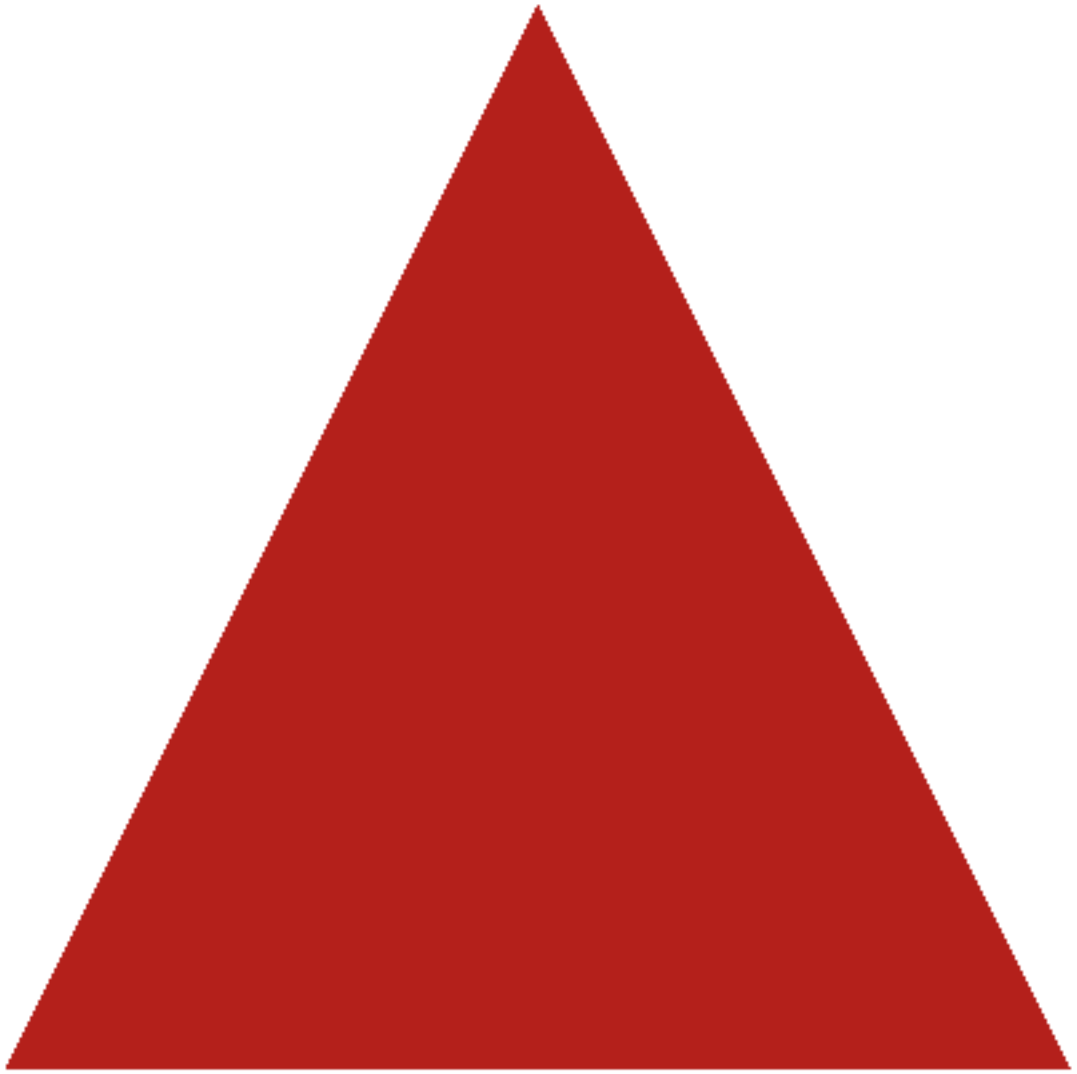
```
(def callback (atom nil))

(def ^:dynamic ^Graphics2D *graphics*)

(defn create-panel []
  (proxy [JPanel] []
    (paint [graphics]

      ;; center, magnify, etc.
      (init-graphics this graphics)

      (when-let [callback @callback]
        (binding [*graphics* graphics]
          (callback))))))
```

```
(def callback (atom nil))

(def ^:dynamic ^Graphics2D *graphics*)

(defn create-panel []
  (proxy [JPanel] []
    (paint [graphics]

      ;; center, magnify, etc.
      (init-graphics this graphics)

      (when-let [callback @callback]
        (binding [*graphics* graphics]
          (callback))))))
```

```
(def callback (atom nil))

(def ^:dynamic ^Graphics2D *graphics*)

(defn create-panel []
  (proxy [JPanel] []
    (paint [graphics]

      ;; center, magnify, etc.
      (init-graphics this graphics)

      (when-let [callback @callback]
        (binding [*graphics* graphics]
          (callback)))))))
```

```
(def callback (atom nil))

(def ^:dynamic ^Graphics2D *graphics*)

(defn create-panel []
  (proxy [JPanel] []
    (paint [graphics]

      ;; center, magnify, etc.
      (init-graphics this graphics)

      (when-let [callback @callback]
        (binding [*graphics* graphics]
          (callback)))))))
```

```
(defn draw-triangle []
  (.fillPolygon *graphics*
    (int-array [-1 0 1])
    (int-array [-1 1 -1])
    3))

(reset! callback draw-triangle)
```



```
(def colors
  {:red       Color/RED
   :black     Color/BLACK
   :green     Color/GREEN
   :blue      Color/BLUE
   :firebrick (Color. 178 34 34)}})
```

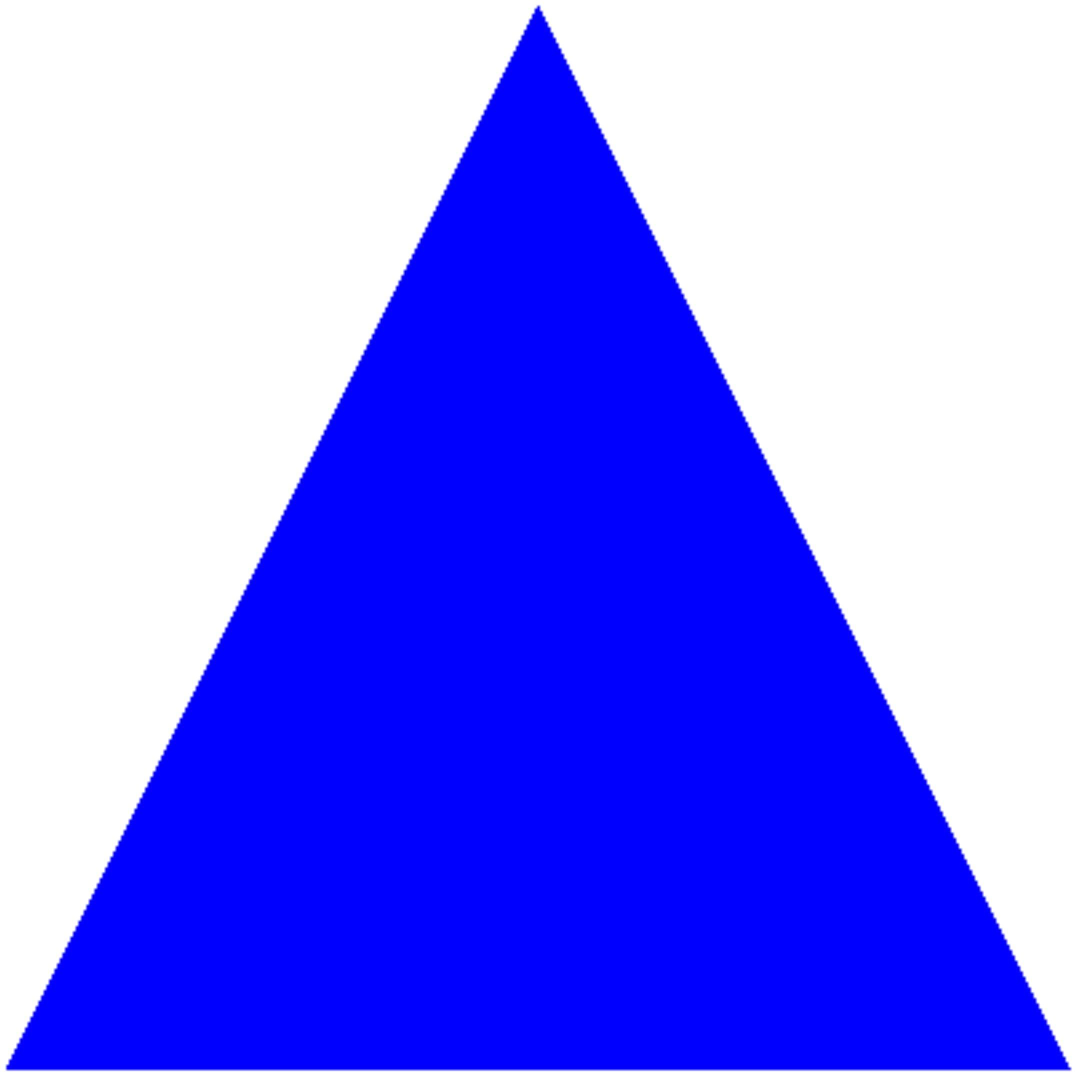
```
(defmacro with-color
  [color & body]
  `(let [original-color# (.getColor *graphics*)]
     (.setColor *graphics* (colors ~color))
     (try
       ~@body
       (finally
         (.setColor *graphics* original-color#)))))
```

```
(defmacro with-color
  [color & body]
  `(let [original-color# (.getColor *graphics*)]
     (.setColor *graphics* (colors ~color))
     (try
       ~@body
       (finally
         (.setColor *graphics* original-color#)))))
```

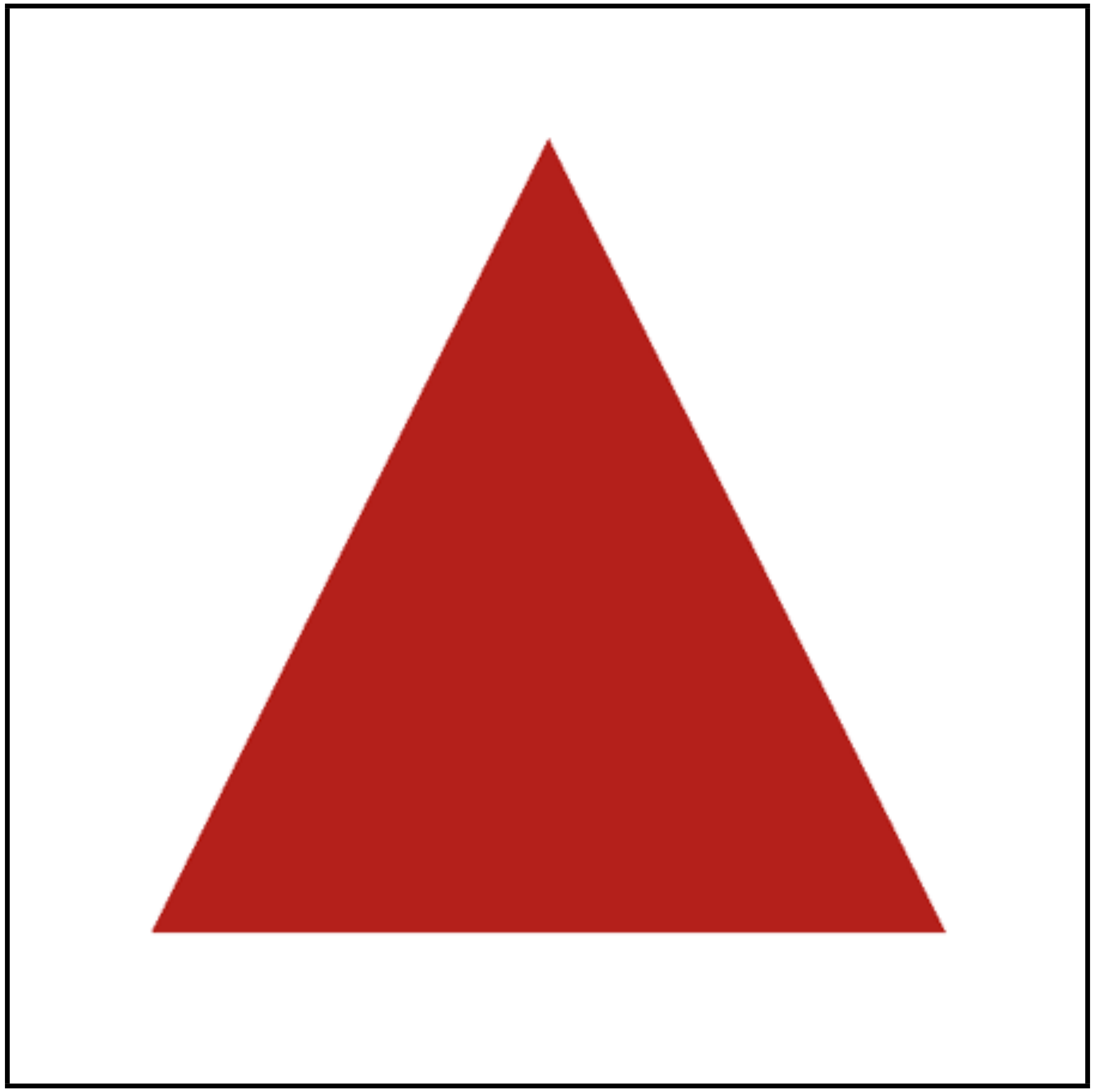
```
(defmacro with-color
  [color & body]
  `(let [original-color# (.getColor *graphics*)]
     (.setColor *graphics* (colors ~color))
     (try
       ~@body
       (finally
         (.setColor *graphics* original-color#)))))
```

```clojure
(defmacro with-color
  [color & body]
  `(let [original-color# (.getColor *graphics*)]
     (.setColor *graphics* (colors ~color))
     (try
       ~@body
       (finally
         (.setColor *graphics* original-color#)))))
```

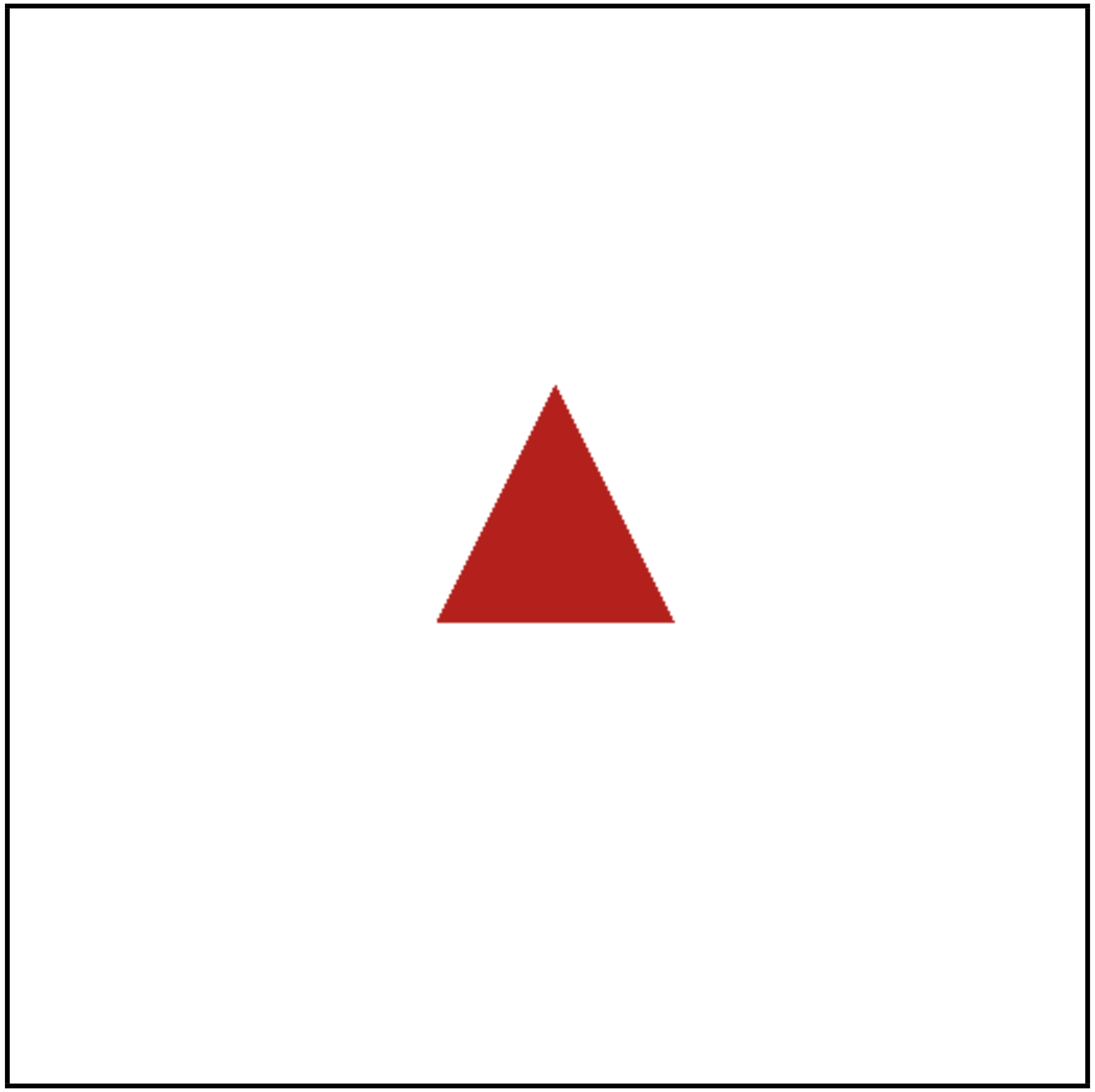
```
(with-color :blue (draw-triangle))
```



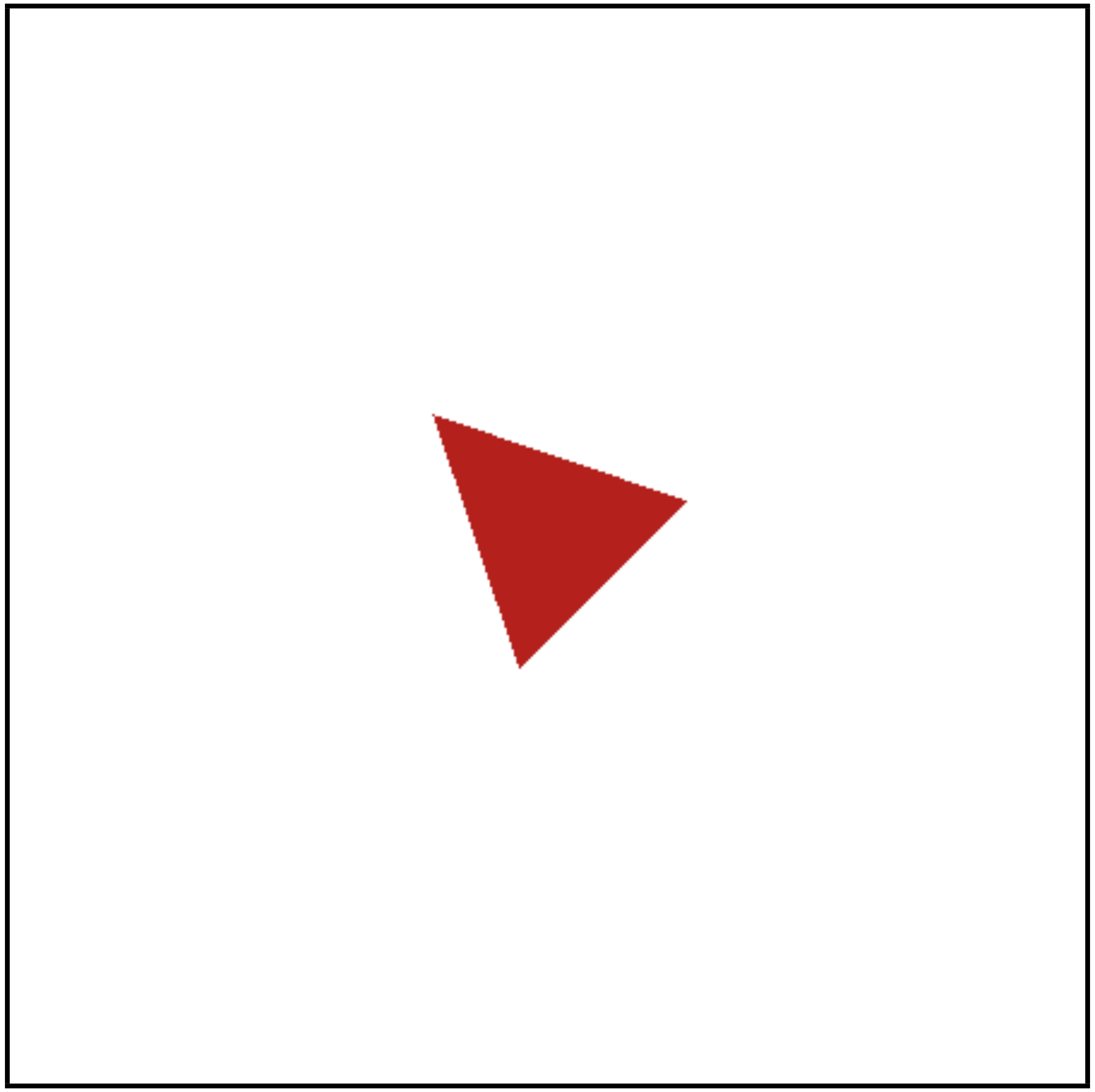
```
(scale 0.25 0.25)

(rotate 45)

(translate 0 2)

(rotate 180)
```



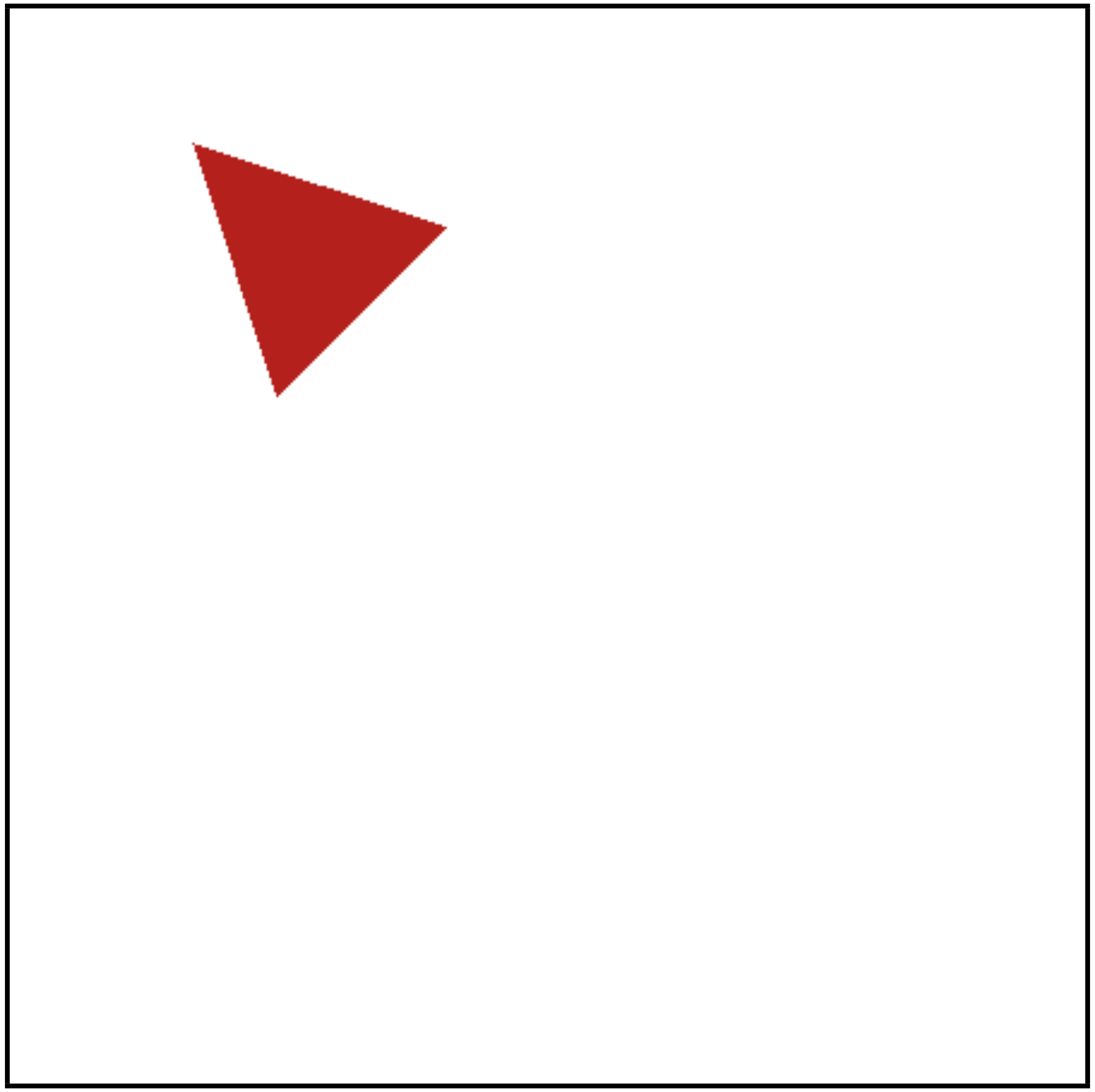
```
(scale 0.25 0.25)

(rotate 45)

(translate 0 2)

(rotate 180)
```



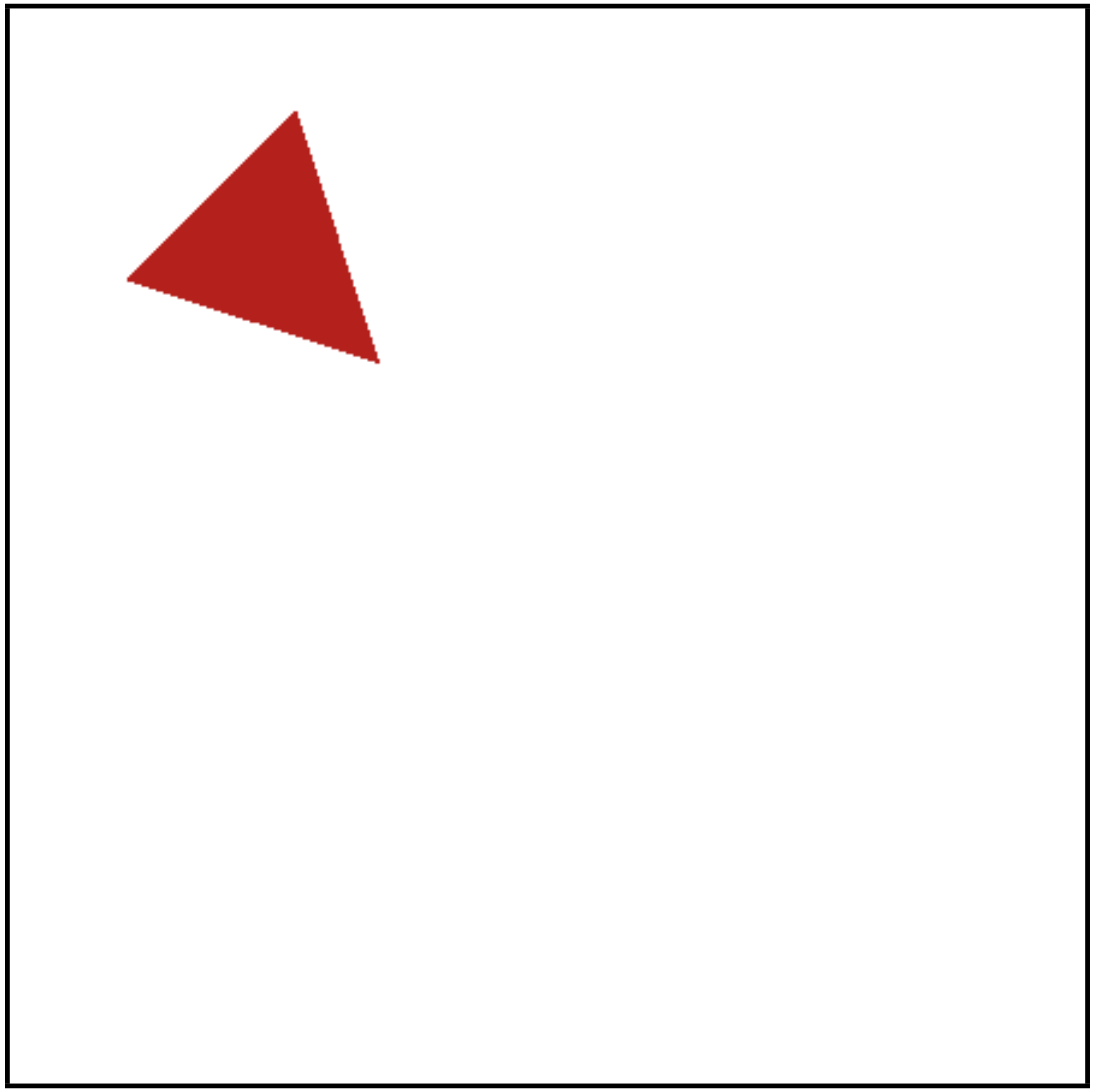
```
(scale 0.25 0.25)

(rotate 45)

(translate 0 2)

(rotate 180)
```



```
(scale 0.25 0.25)

(rotate 45)

(translate 0 2)

(rotate 180)
```



```
(scale 0.25 0.25)

(rotate 45)

(translate 0 2)

(rotate 180)
```



```
(defmacro with-scoped-transforms
  [& body]
  `(let [original-transform# (.getTransform *graphics*)]
     (try
       ~@body
       (finally
         (.setTransform *graphics* original-transform#)))))
```

```
(defmacro with-color
  [color & body]
  `(let [original-color# (.getColor *graphics*)]
     (.setColor *graphics* (colors ~color))
     (try
       ~@body
       (finally
         (.setColor *graphics* original-color#)))))
```

≈

```
(defmacro with-scoped-transforms
  [& body]
  `(let [original-transform# (.getTransform *graphics*)]
     (try
       ~@body
       (finally
         (.setTransform *graphics* original-transform#)))))
```

code

that writes

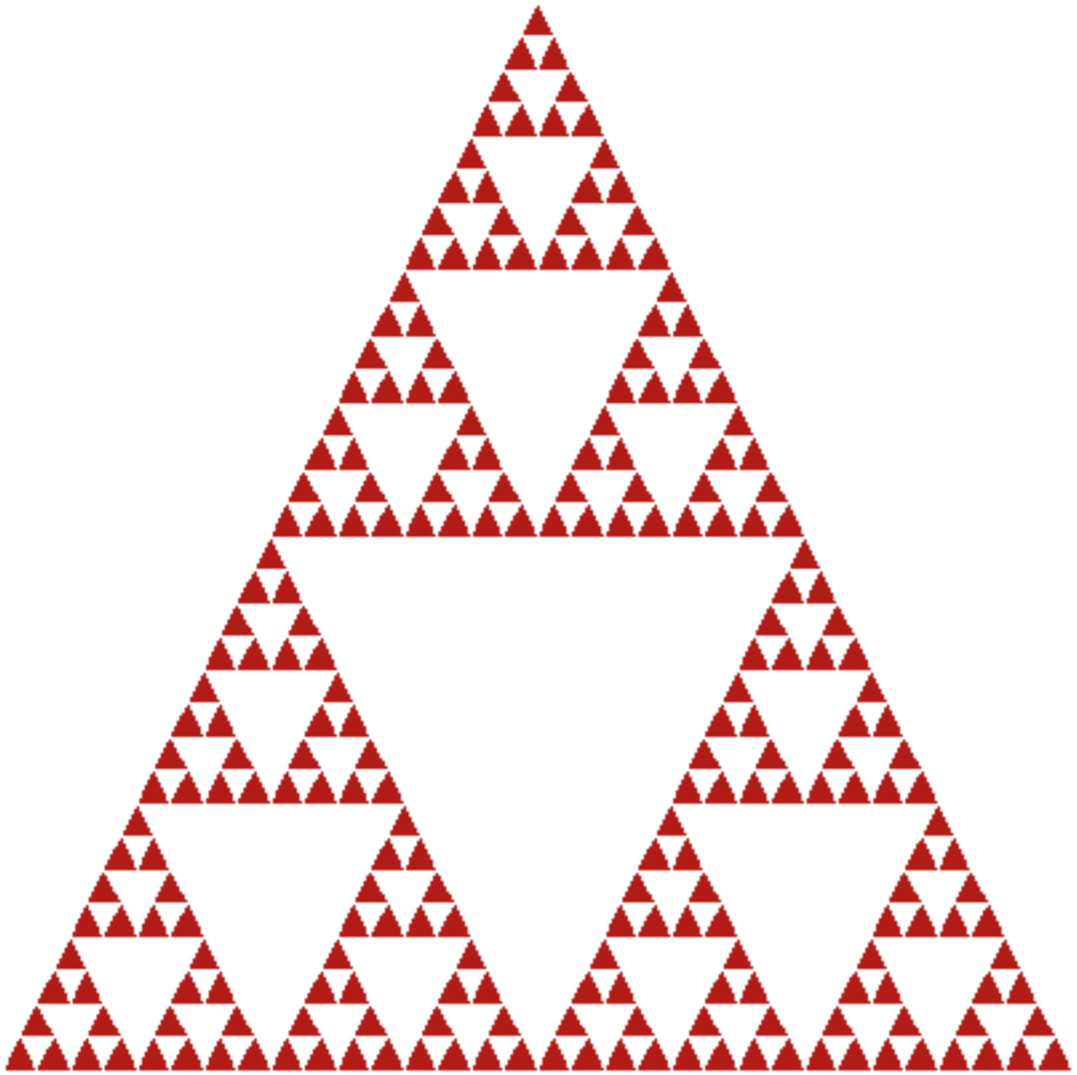
code

that writes

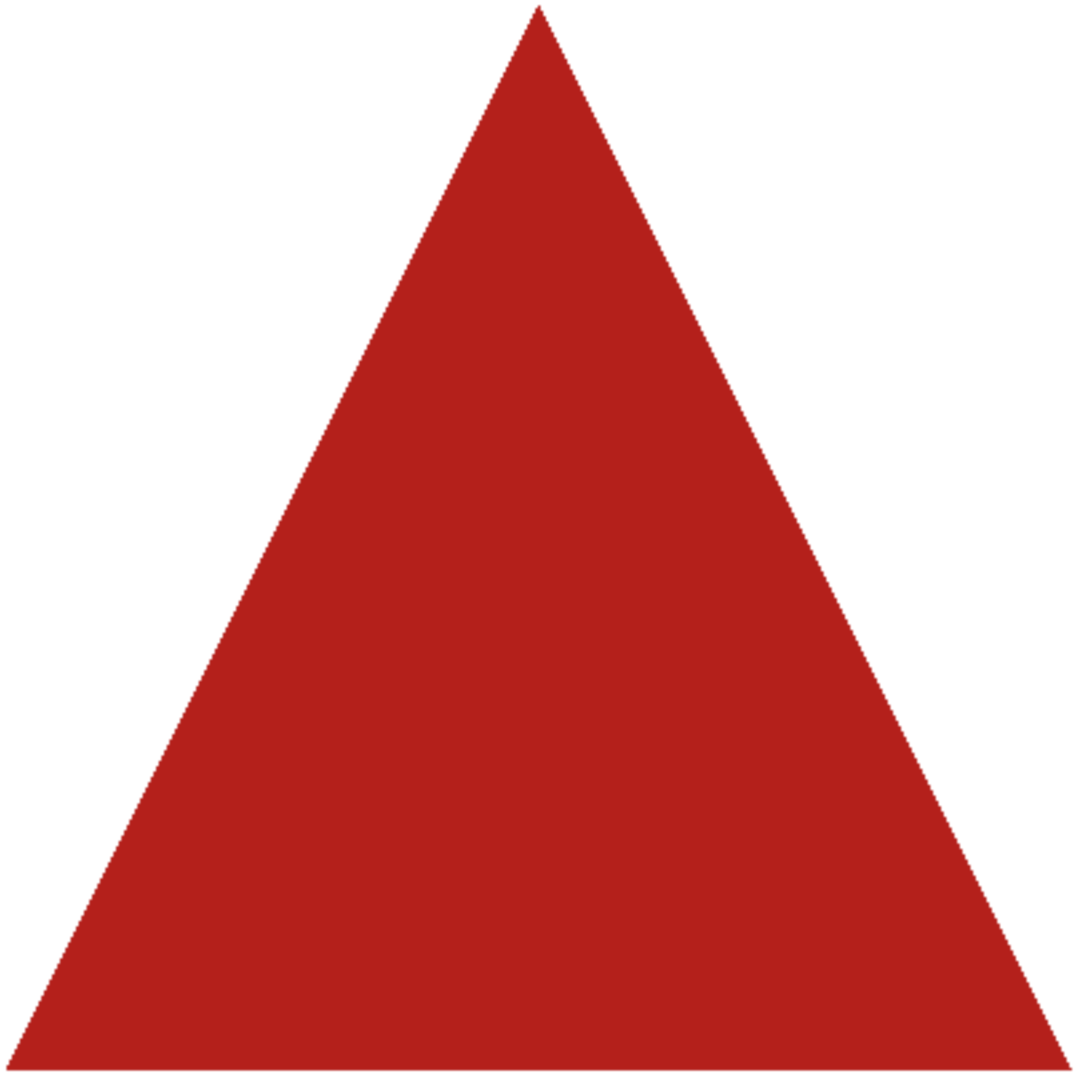
code

} danger zone

sierpinski triangle

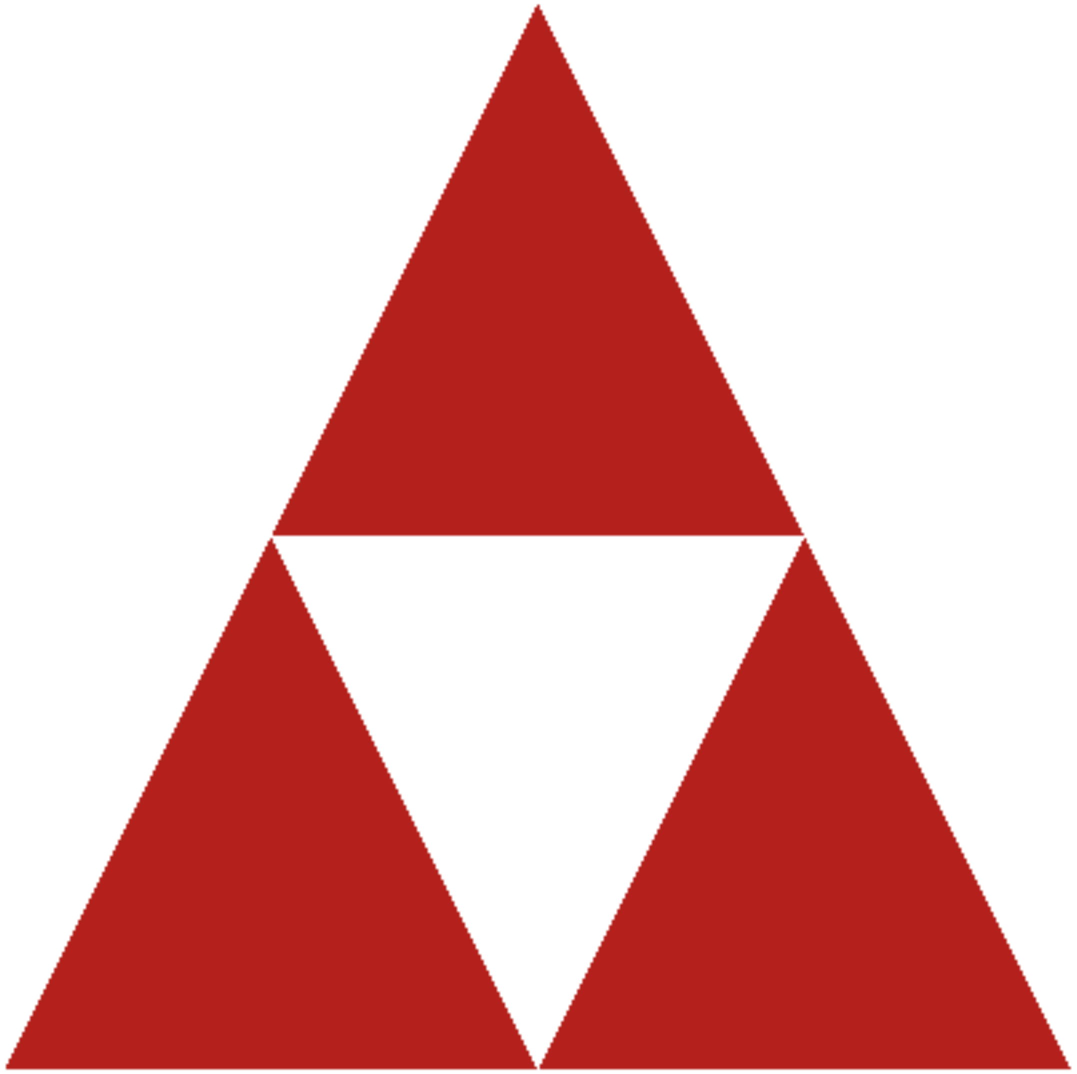


sierpinski triangle

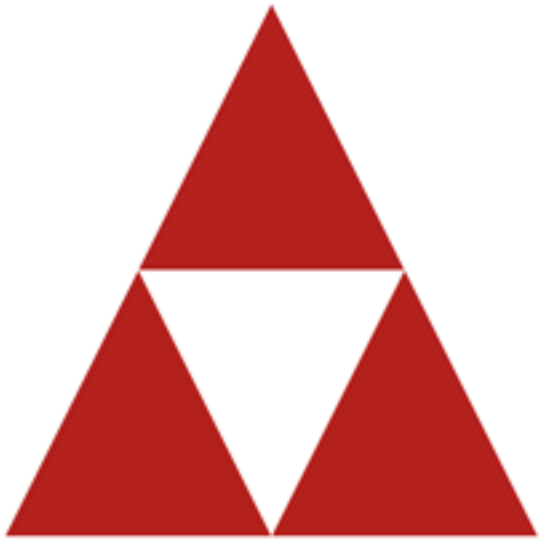


sierpinski triangle

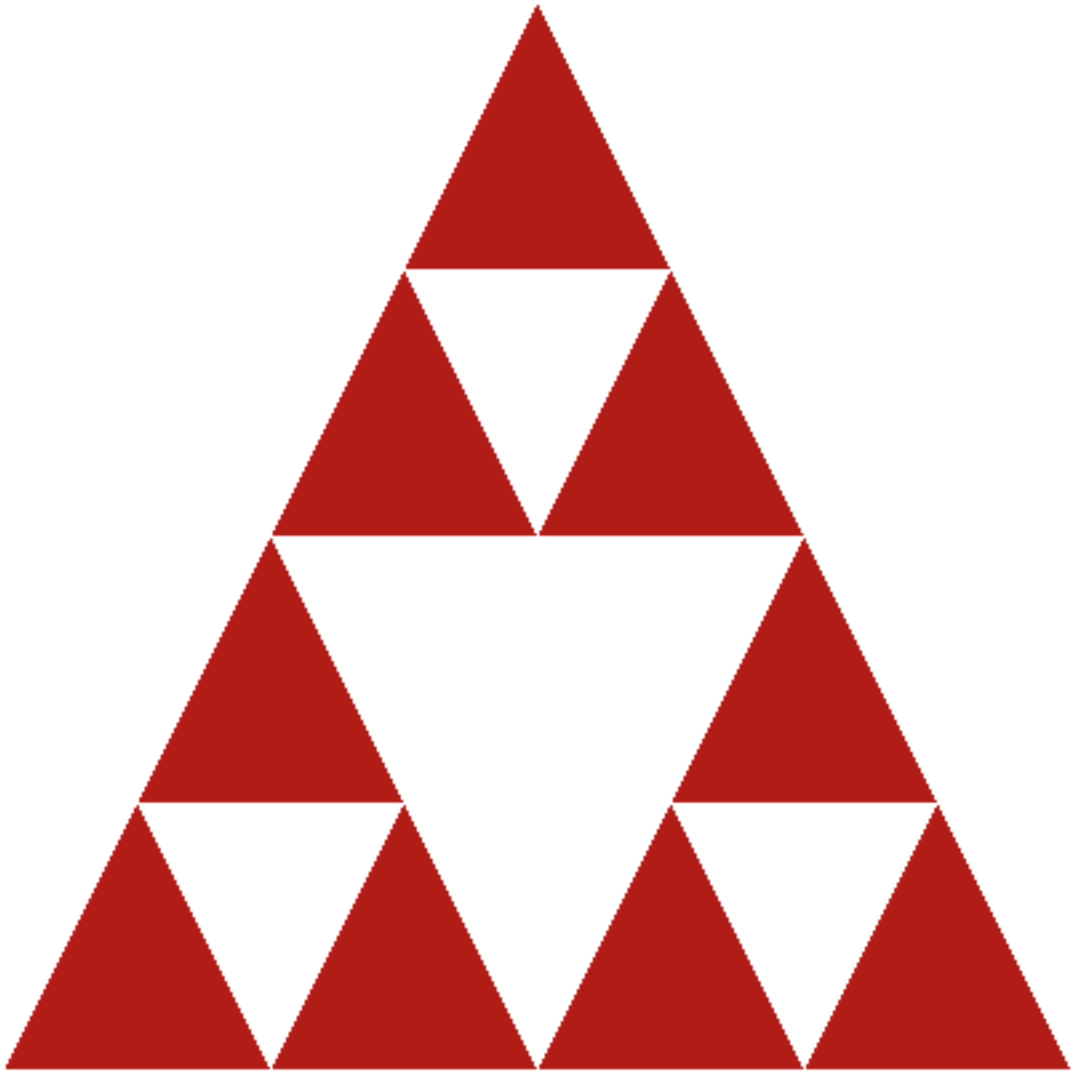
sierpinski triangle



sierpinski triangle



sierpinski triangle

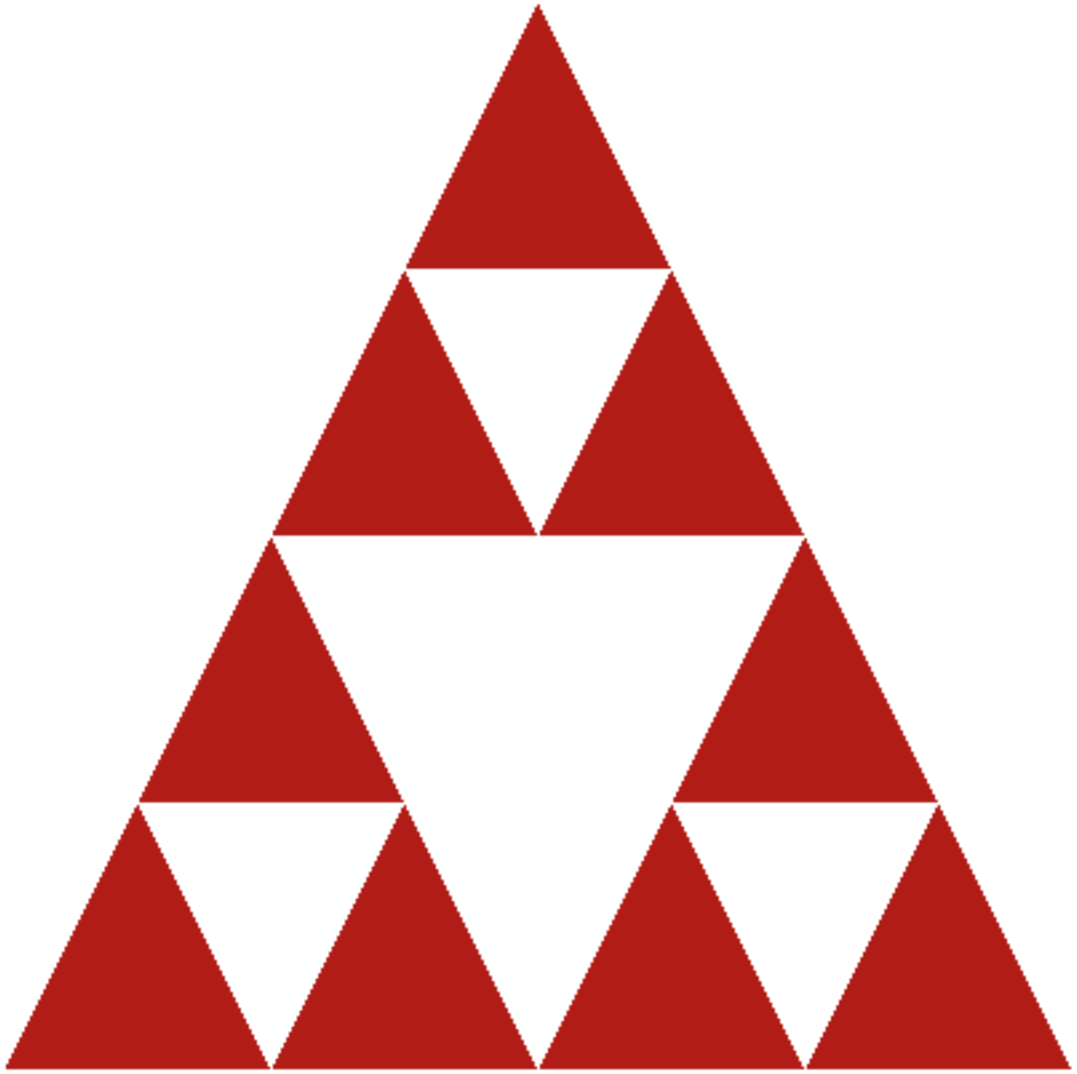


```
(defmacro sierpinski
  [& body]
  `(let [body-fn# (fn [] ~@body)]
     (with-scoped-transforms
       (scale 0.5 0.5)

       ;; bottom-left
       (with-scoped-transforms
         (translate -1 -1)
         (body-fn#))

       ;; bottom-right
       (with-scoped-transforms
         (translate 1 -1)
         (body-fn#))

       ;; top
       (with-scoped-transforms
         (translate 0 1)
         (body-fn#)))))
```

```clojure
(defmacro sierpinski
  [& body]
  `(let [body-fn# (fn [] ~@body)]
     (with-scoped-transforms
       (scale 0.5 0.5)

       ;; bottom-left
       (with-scoped-transforms
         (translate -1 -1)
         (body-fn#))

       ;; bottom-right
       (with-scoped-transforms
         (translate 1 -1)
         (body-fn#))

       ;; top
       (with-scoped-transforms
         (translate 0 1)
         (body-fn#)))))
```

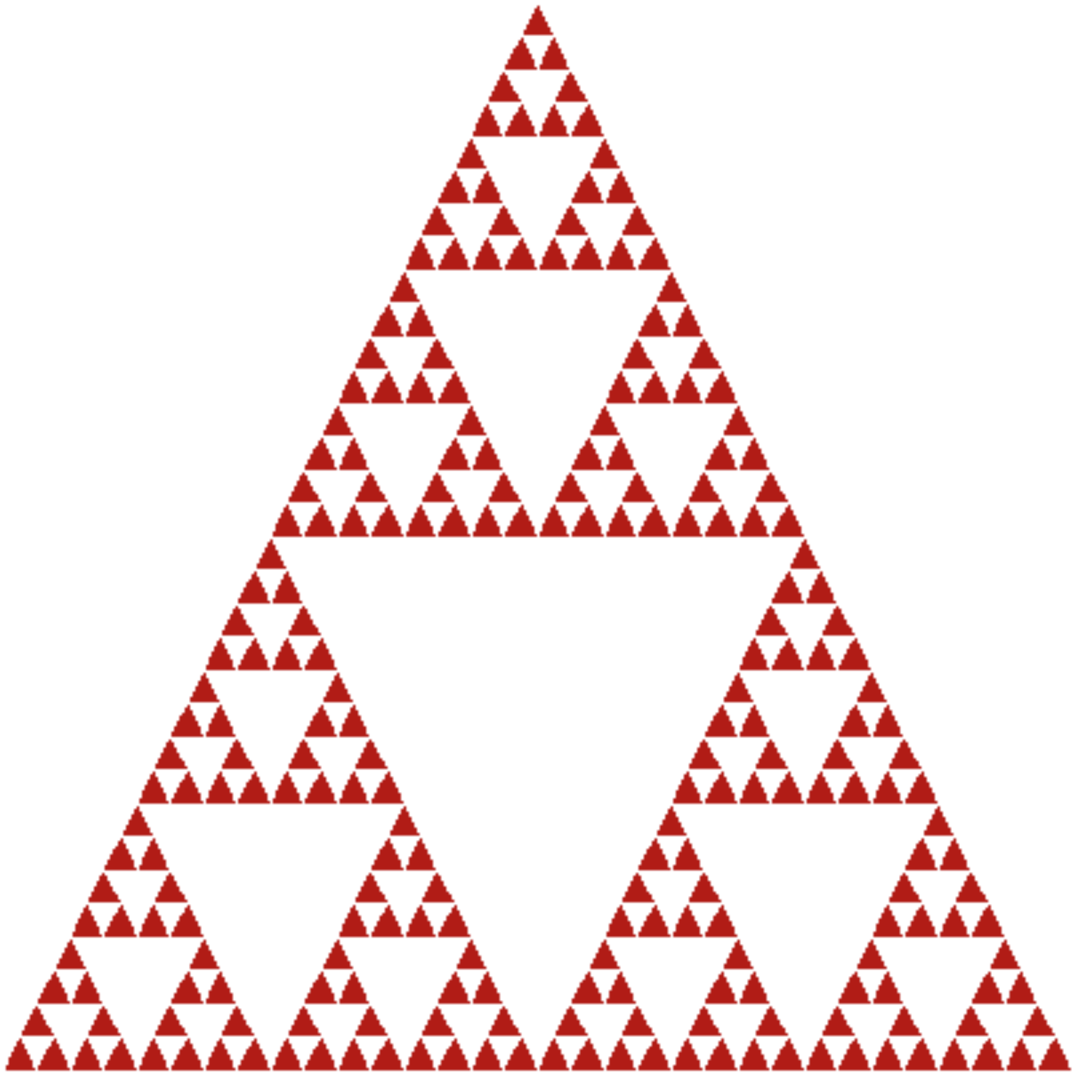
```
(defmacro sierpinski
  [& body]
  `(let [body-fn# (fn [] ~@body)]
     (with-scoped-transforms
       (scale 0.5 0.5)

       ;; bottom-left
       (with-scoped-transforms
         (translate -1 -1)
         (body-fn#))

       ;; bottom-right
       (with-scoped-transforms
         (translate 1 -1)
         (body-fn#))

       ;; top
       (with-scoped-transforms
         (translate 0 1)
         (body-fn#)))))
```

```
(sierpinski (sierpinski (draw-triangle)))
```



```
(defmacro sierpinskis
  [n & body]
  `(-> (do ~@body) ~@(repeat n 'sierpinski)))
```

```
(sierpinskis 5 (draw-triangle))
```



the good

- it works
- it doesn’t deviate too much from the existing API

the bad

- it’s completely imperative
- macros don’t compose

```clojure
(def triangle-renderer draw-triangle)


(defn render [renderer]
  (renderer))
```

```
(defn color
  [f color]
  #(with-color color
     (f)))
```

```
(defn translate*
  [f x y]
  #(with-scoped-transforms
     (translate x y)
     (f)))
```

```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]])

(defn sierpinski
  [f]
  (let [transform (fn [[x y]]
                    (-> f
                      (translate* x y)
                      (scale* 0.5 0.5)))
        [a b c] (map transform offsets)]
    (fn []
      (a)
      (b)
      (c))))
```

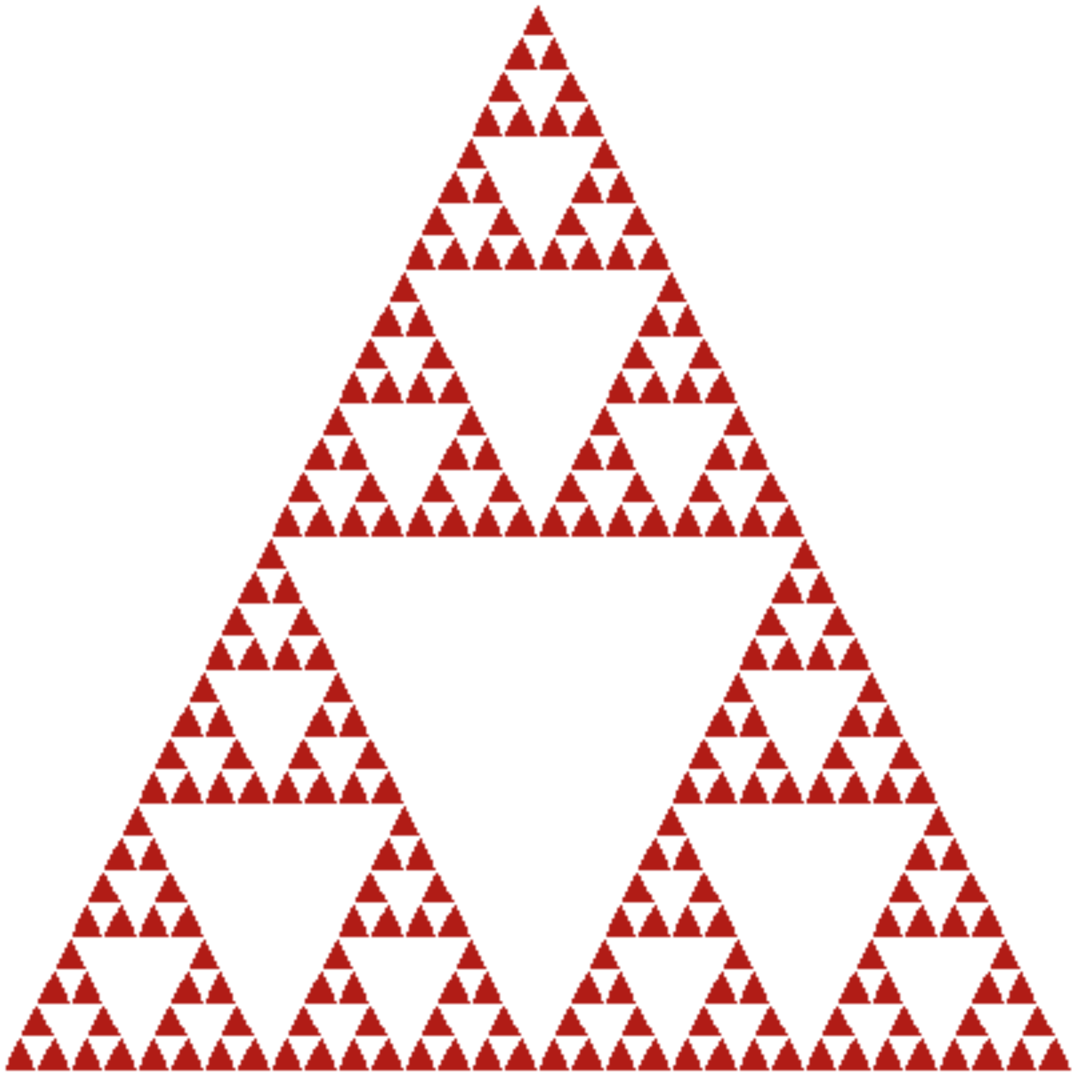
```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]]])

(defn sierpinski
  [f]
  (let [transform (fn [[x y]]
                    (-> f
                      (translate* x y)
                      (scale* 0.5 0.5)))
        [a b c] (map transform offsets)]
    (fn []
      (a)
      (b)
      (c))))
```

```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]]])

(defn sierpinski
  [f]
  (let [transform (fn [[x y]]
                    (-> f
                      (translate* x y)
                      (scale* 0.5 0.5)))
        [a b c] (map transform offsets)]
    (fn []
      (a)
      (b)
      (c))))
```

```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]]])

(defn sierpinski
  [f]
  (let [transform (fn [[x y]]
                    (-> f
                      (translate* x y)
                      (scale* 0.5 0.5)))
        [a b c] (map transform offsets)]
    (fn []
      (a)
      (b)
      (c))))
```

```
(def sierpinskis
  (iterate sierpinski triangle-renderer))
```

```
(render (nth sierpinskis 5))
```



the good

- it’s more idiomatic
- it places fewer constraints on code that uses it

the bad

- it’s completely opaque
- it’s mostly untestable

```
(defprotocol Shape
  (transform [_ transformation])
  (render [_]))

(defrecord Polygon
  [^java.awt.Polygon polygon color]
  Shape
  (render [_]
    (with-color color
      (.fill *graphics* polygon)))
  (transform [this transformation]
    (assoc this
      :polygon (.createTransformedShape
                  ^AffineTransform transformation
                  polygon))))
```

```
(defprotocol Shape
  (transform [_ transformation])
  (render [_]))

(defrecord Polygon
  [^java.awt.Polygon polygon color]
  Shape
  (render [_]
    (with-color color
      (.fill *graphics* polygon)))
  (transform [this transformation]
    (assoc this
      :polygon (.createTransformedShape
                  ^AffineTransform transformation
                  polygon))))
```

```
(defprotocol Shape
  (transform [_ transformation])
  (render [_]))

(defrecord Polygon
  [^java.awt.Polygon polygon color]
  Shape
  (render [_]
    (with-color color
      (.fill *graphics* polygon)))
  (transform [this transformation]
    (assoc this
      :polygon (.createTransformedShape
                  ^AffineTransform transformation
                  polygon))))
```

```
(defprotocol Shape
  (transform [_ transformation])
  (render [_]))

(defrecord Polygon
  [^java.awt.Polygon polygon color]
  Shape
  (render [_]
    (with-color color
      (.fill *graphics* polygon)))
  (transform [this transformation]
    (assoc this
      :polygon (.createTransformedShape
                  ^AffineTransform transformation
                  polygon))))
```

```
(defn triangle
  ([]
     (triangle :red))
  ([color]
     (Polygon.
       (java.awt.Polygon.
          (int-array [-1 0  1])
          (int-array [-1 1 -1])
          3)
       color)))
```

```
(defn transformation []
  (AffineTransform.))

(defn scale [transformation x y]
   (let [transformation (AffineTransform. transformation)]
      (.concatenate
        transformation
        (AffineTransform/getScaleInstance x y))
      transformation)))
```

```
(defn transformation []
  (AffineTransform.))

(defn scale [transformation x y]
   (let [transformation (AffineTransform. transformation)]
      (.concatenate
        transformation
        (AffineTransform/getScaleInstance x y))
      transformation)))
```

```
(defn transformation []
  (AffineTransform.))

(defn scale [transformation x y]
   (let [transformation (AffineTransform. transformation)]
      (.concatenate
        transformation
        (AffineTransform/getScaleInstance x y))
      transformation)))
```

```
(defn transformation []
  (AffineTransform.))

(defn scale [transformation x y]
   (let [transformation (AffineTransform. transformation)]
      (.concatenate
        transformation
        (AffineTransform/getScaleInstance x y))
      transformation)))
```

```
(defn transformation []
  (AffineTransform.))

(defn scale [transformation x y]
   (let [transformation (AffineTransform. transformation)]
      (.concatenate
        transformation
        (AffineTransform/getScaleInstance x y))
      transformation)))
```

```
(defmacro defn-transform [name args affine-transform]
  `(defn ~name ~args
     (let [^AffineTransform original-transformation# ~(first args)
           transformation# (AffineTransform. original-transformation#)]
       (.concatenate transformation# ~affine-transform)
       transformation#)))

(defn-transform scale [transformation x y]
  (AffineTransform/getScaleInstance x y))
```

```
(defmacro defn-transform [name args affine-transform]
  `(defn ~name ~args
     (let [^AffineTransform original-transformation# ~(first args)
           transformation# (AffineTransform. original-transformation#)]
       (.concatenate transformation# ~affine-transform)
       transformation#)))

(defn-transform scale [transformation x y]
  (AffineTransform/getScaleInstance x y))
```

```
(defmacro defn-transform [name args affine-transform]
  `(defn ~name ~args
     (let [^AffineTransform original-transformation# ~(first args)
           transformation# (AffineTransform. original-transformation#)]
       (.concatenate transformation# ~affine-transform)
       transformation#)))

(defn-transform scale [transformation x y]
  (AffineTransform/getScaleInstance x y))
```

```
(defmacro defn-transform [name args affine-transform]
  `(defn ~name ~args
     (let [^AffineTransform original-transformation# ~(first args)
           transformation# (AffineTransform. original-transformation#)]
       (.concatenate transformation# ~affine-transform)
       transformation#)))

(defn-transform scale [transformation x y]
  (AffineTransform/getScaleInstance x y))
```

```
(defmacro defn-transform [name args affine-transform]
  `(defn ~name ~args
     (let [^AffineTransform original-transformation# ~(first args)
           transformation# (AffineTransform. original-transformation#)]
       (.concatenate transformation# ~affine-transform)
       transformation#)))

(defn-transform scale [transformation x y]
  (AffineTransform/getScaleInstance x y))
```

```
(defmacro defn-transform [name args affine-transform]
  `(defn ~name ~args
     (let [^AffineTransform original-transformation# ~(first args)
           transformation# (AffineTransform. original-transformation#)]
       (.concatenate transformation# ~affine-transform)
       transformation#)))

(defn-transform scale [transformation x y]
  (AffineTransform/getScaleInstance x y))
```

```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]]])

(def transformations
  (map
    (fn [[x y]]
      (-> (transformation)
        (scale 0.5 0.5)
        (translate x y)))
    offsets))

(defn sierpinski
  [shapes]
  (mapcat
    (fn [transformation]
      (map #(transform % transformation) shapes))
    transformations))
```

```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]])

(def transformations
  (map
    (fn [[x y]]
      (-> (transformation)
        (scale 0.5 0.5)
        (translate x y)))
    offsets))

(defn sierpinski
  [shapes]
  (mapcat
    (fn [transformation]
      (map #(transform % transformation) shapes))
    transformations))
```

```
(def offsets
  [[-1 -1]
   [1  -1]
   [0   1]])

(def transformations
  (map
    (fn [[x y]]
      (-> (transformation)
        (scale 0.5 0.5)
        (translate x y)))
    offsets))

(defn sierpinski
  [shapes]
  (mapcat
    (fn [transformation]
      (map #(transform % transformation) shapes))
    transformations))
```

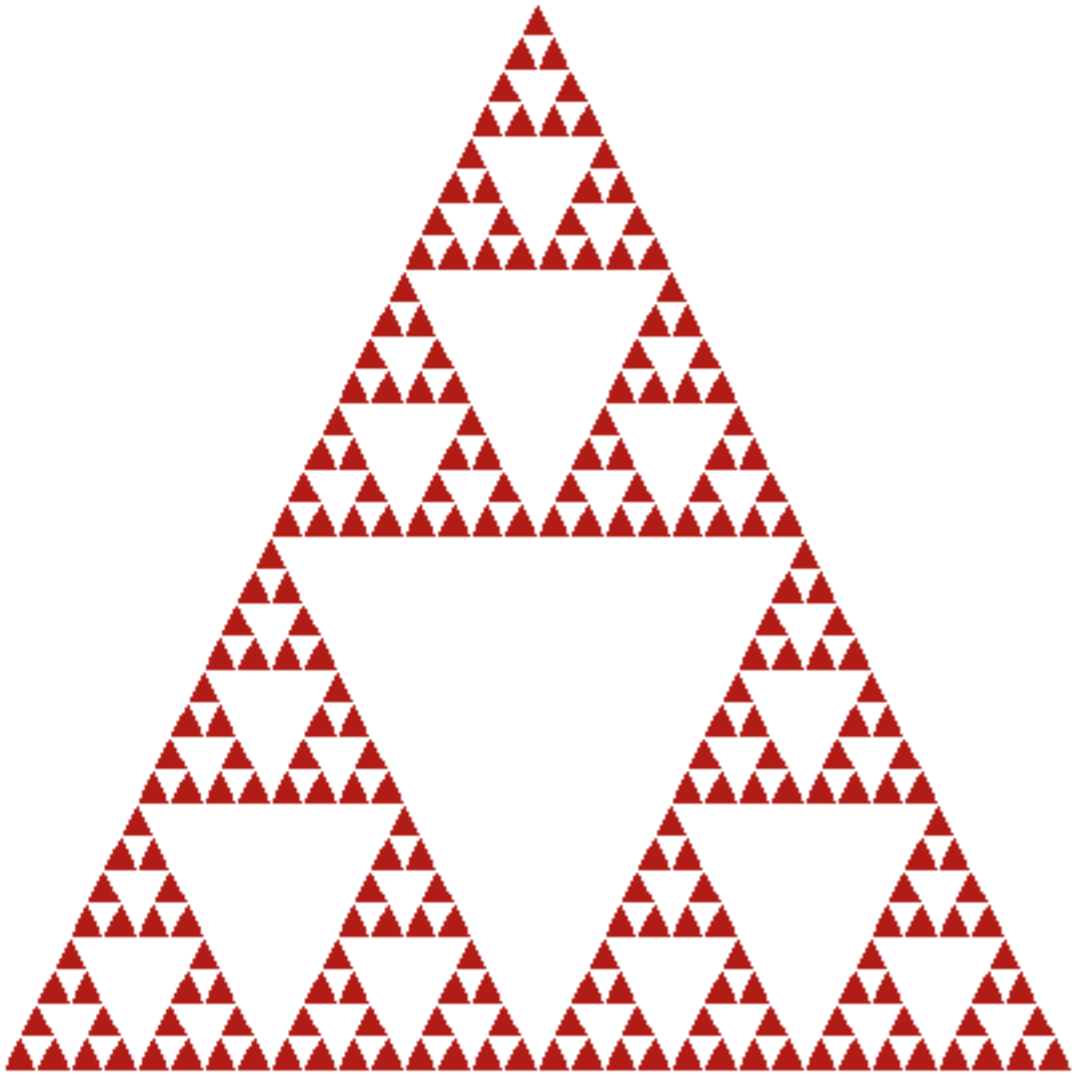
```
(def sierpinskis
  (iterate sierpinski [(triangle)]))
```

```
(defn render-all [shapes]
  (doseq [s shapes]
    (render s)))
```

```
(def sierpinskis
  (iterate sierpinski [(triangle)]))
```

```
(defn render-all [shapes]
  (doseq [s shapes]
    (render s)))
```

```
(render-all (nth sierpinskis 5))
```



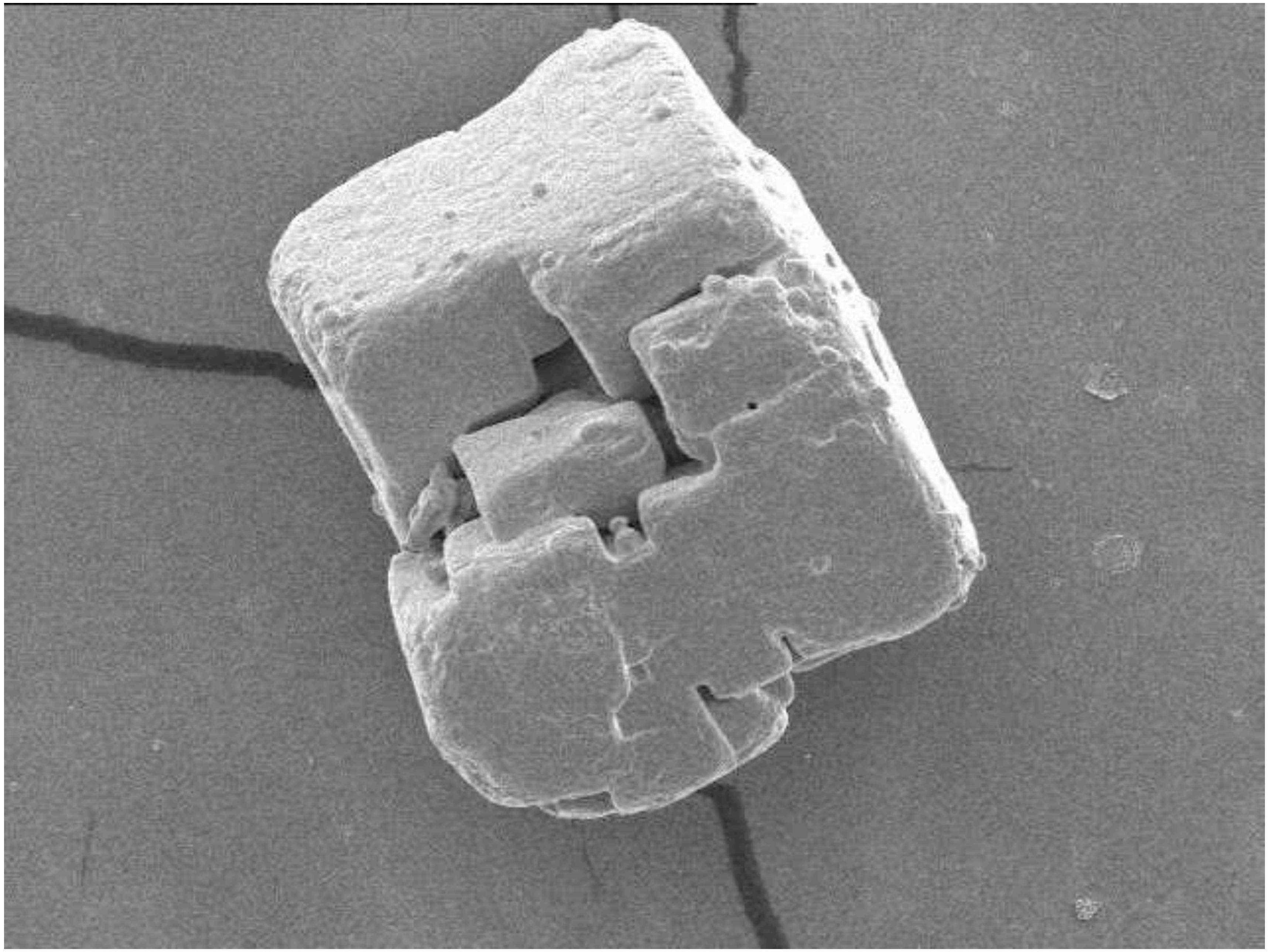
the good

- first-class data
- fully idiomatic

the bad

- may not be memory efficient
- we’ve completely reinvented the API
- other features of Java2D may not cleanly fit this new idiom

a grain of salt



questions?