Team: Ilan Biala, Nick Roberts, Varun Sharma
API: Processing

Issues List

Relating to the relationship between a drawable entity and its settings.

Properties of a shape (color, outline weight, and outline color) **should be modeled separately from the shape's identity** (its dimensions, position, and whether it is a circle/square/etc.)
Pros:
- Promotes code reuse by allowing client to easily draw the same shape with multiple properties.

Cons:
- Requires the library to be opinionated about what properties are intrinsic to a shape. Some users might find it confusing that varying color and varying dimension between multiple drawings of a shape requires different code patterns.

There should be <u>no</u> provision for setting the default settings for drawing a shape. Consequently, the APIs for constructing instances of *ShapeSettings* should require users to specify a lot of information up front (i.e. there's no way to specify just the stroke weight without also specifying the stroke color) to avoid the pitfall of having the API choose "reasonable" defaults that in fact are very rarely used.
Pros:
- Allowing the user to set a globally-default settings value could result in users frequently setting this value, relying on mutable state to draw shapes as they want. This encourages similar usage patterns as with Processing. We want to avoid this.
- The pitfall with the API trying to choose "reasonable" defaults is this: when, if ever, is there a natural way to assign a color and outline to a square that makes sense in many contexts? We claim that the user will always have some opinion as to a square's color and outline, and so the API should make it natural for them to provide this information.

Cons:
- It might be convenient to set a globally-default settings value if many shapes are indeed drawn with those same settings.

Relating to class hierarchies

***Shape* should act as a common supertype of rectangle, circle, and ellipse.**
Pros:
- A method can return *Shape*, and the client can use draw the result of this method.
- You can store Rectangles and Ellipses in the same data structure and still draw the contents of the data structure on the screen.
- We can control the subtypes of *Shape* by creating a package-private constructor, so we don't have to worry about the client adding more subtypes that we don't know how to draw.

Cons:
- Not idiomatic in Java, since we need to maintain a runtime representation of a shape's type internally to the API. (However, the uses of *instanceof* and casting are quarantined completely to the API. They never are exposed to the user.)
- *Shape* does not have any useful methods. It is only a marker type. However, only our wrapper controls subtypes of *Shape*, so this isn't a problem. (I.e. we don't allow the client to implement their own shapes.)

***Shape* should not be a supertype of canvases and images, but we should provide a type *Drawable* that packages together things ready to be drawn (a shape and its settings, an image and its settings, or a canvas and its settings).**
Pros:
- It's useful to decouple a shape from the settings (color, stroke) used to draw that shape. Then you can draw the same shape multiple times with different settings.
- Image/canvas settings are different than shape settings, and there is still common behavior between images and canvases (you can draw them on the screen!)
- Now a method can return a Drawable, which is either a shape plus settings, an image plus settings, or a canvas plus settings. A canvas can take a Drawable and will know how to draw it.

Cons:
- Requires an inelegant implementation in Java (where we case on the runtime type of objects), but this is just something we have to do in the internal API; it's not externally visible.

***Circle* should be a separate type than (in fact, a subtype of) *Ellipse*.**

Pros:

- It is arguably counterintuitive to create a circle by calling *Ellipse.of(2*radius, 2*radius)*. We could add another static factory for creating a circle, but it is strange for this factory to be located inside of *Ellipse*.
- Once you create a circle, it's more likely that you'll want to get the radius out of the circle than it is you'll want to get the width/height. Adding a separate *Circle* type allows the client to easily access this information once a circle is created.

Cons:

- Forces more types into the API.
- Complicates equality semantics if we want circles and ellipses to test as equal when they will be drawn the same. We can still uphold the behavioral contracts of *Object#equals* since neither *Circle* nor *Ellipse* can be overridden further by the client.
- It seems that this would entail *Square* class for consistency. However, the kind of information you want to access about a square (length/height) is the same information already accessible on the *Rectangle* class. Furthermore, ellipse is a more complex concept than a rectangle, so it's nice to give an inexperienced client the opportunity to just think about circles.

Relating to the representation of data.

The construction of shapes, images, and settings should be immutable, but the act of drawing on a canvas should be mutable.

Pros:

- Making shapes and settings (and the relationship therebetween) as immutable fixes many of the problems with processing where lingering color and position-mode settings persist to future calls.
- The user draws on a canvas by making a series of calls, superimposing newly-drawn images on top of previously-drawn ones. Having this API be immutable would complicate usage patterns with little to no benefit, and arguably violating the normal idioms of Java (e.g. adding to a List is a mutable action; so too is drawing on a canvas).

Cons

- Inconsistent to have mutability some places and immutability others, but we think the taxonomy is understandable enough (only drawing on a canvas is mutable!) for this to make sense to clients.

Our API should deal in doubles where Processing dealt in floats, which is to say "almost everywhere."

Pros:

- Processing apparently supports floating point inputs for some functions, so *double* seems to be a better choice than *int*.
- Floats rack up imprecision in client arithmetic, so doubles promote better client code.

Cons:

- It's confusing that pixels can be specified as floating point data.
- We do lose precision when converting from double to float, but at least we haven't accumulated error from multiple lossy arithmetic steps.

Relating to the presence/absence of convenience methods

Although they are redundant with the *Canvas#draw(Drawable, Position)* method, we should include such overloadings as *Canvas#draw(Shape, ShapeSettings, Position*) and *Canvas#draw(Image, ImageSettings, Position).*

Pros:

- The *Drawable* overloading is clearly desirable, since it allows the client to draw the return value of a method that's just declared to return *Drawable*.
- While the *Shape/ShapeSettings* overloading is superfluous (since you could just call *Drawable.ofShape(shape, settings)*), including the overloading allows for a simpler way of drawing when the client doesn't want to go through the second layer of indirection.

Cons

- Slightly more complex API

Anywhere that we provide a method that takes a pair *(Image, ImageSettings)*, we should also provide an overriding that takes just an *Image*. However, we should not make the same allowance for *(Shape, ShapeSettings)*; the *ShapeSettings* parameter should always be mandatory.

Pros:

- It makes sense to draw an image with its settings as originally imported from file: we can use the dimensions from the file with no tint. This can be the fallback rendering option when no *ImageSettings* are specified.
- There is arguably no reasonable default value for *ShapeSettings*. You could argue that it might make sense to draw the shape with transparent fill, a stroke weight of 1, and a stroke color of block (therefore just drawing the outline), but this is almost never what the client actually wants to draw. The client should always be required to specify properties of the shape; and, if they're tired of specifying the same *ShapeSettings* together with the same *Shape*, they can package the values together with *Drawable*.

Cons

- Small inconsistency between *ImageSettings* and *ShapeSettings*.