Photo by Marvin Meyer on Unsplash

Lab 2

Flags · Target · Expand Your Toolbelt · Heart · Clover · Rounded Rectangle · Isosceles Trapezoid · Swiss Railway Clock

Copy Your Lab 1 Toolbelt

In Lab 1 you wrote quite a few methods for your `Toolbelt` class. The `Toolbelt` class included in the Lab 2 starter repository is empty. Please **copy the contents of your Lab 1 `Toolbelt` class** into the Lab 2 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future).

Base Colors

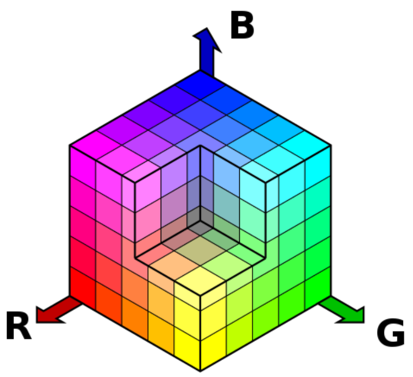
We saw that JTamaro provides some ***names*** for various ***values*** of ***type*** `Color`:

```
RED   GREEN   BLUE   CYAN   MAGENTA   YELLOW   BLACK   WHITE
```

Additionally, it also provides a ***method*** with which we can produce any color (the 8 mentioned above, and any other color your computer could show on the screen):

```
Color rgb(int red, int green, int blue)
```

The `rgb` method expects three integer numbers (values of type `int`) as arguments. Each number must be a value from 0 to 255, inclusive. You can imagine the method as controlling three dimmable lamps: one red, one blue, and one green. The parameters determine the strength of each lamp. If you pass a 0 as an argument, the lamp is off. If you pass a 255, the lamp is turned on at full strength.



The RGB color space can be seen as a cube in 3-dimensional space. The three axes correspond to the three base colors: red, green, and blue.

The cube has 8 corners, each of the corners corresponds to one of the named colors in JTamaro.

If you call the `rgb` method as follows, you will get a pure red color. The red lamp is fully on, the green and blue lamps are off. The call produces a `Color` value that is equivalent to the value bound to the name `RED`:

```
rgb(255, 0, 0)
```

This means, to create a red circle, you can call...

```
Toolbelt.circle(200, rgb(255, 0, 0))
```

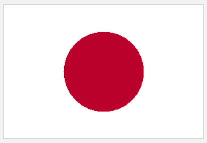
...or you can call...

```
Toolbelt.circle(200, RED)
```

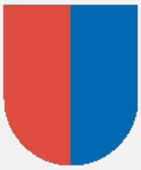
From now on, tasks in labs might require you to create colors, e.g., using `rgb`.

A. Flags

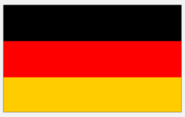
Task A1

Class:	`JapaneseFlag`
Task:	Implement the `flag` method. **Assert** that the value of the `height` parameter is acceptable. The Japanese flag has a 3:2 aspect ratio. The red disk is 3/5 of the height. Define a parameter-less method `japaneseRed` that produces the official red color for the Japanese flag (using values 188, 0, 45 for the red, green, and blue components, respectively). The background is pure `WHITE`. Use your toolbelt.
Run in JShell:	`show(JapaneseFlag.flag(100))`
Output:	

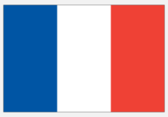
Task A2

Class:	`TicinoCoatOfArms`
Task:	Implement the `coatOfArms` method. **Assert** that the value of the `radius` parameter is acceptable. With a radius of 50, the overall width of the graphic is 100, and the overall height is 240% of the radius. Define and use two parameter-less methods with appropriate names for the official red (RGB: 221, 77, 62) and blue (RGB: 0, 106, 177) colors.
Run in JShell:	`show(TicinoCoatOfArms.coatOfArms(50))`
Output:	

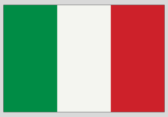
Task A3

Class:	`GermanFlag`
Task:	Implement a `flag` method with a `height` parameter of type `double`, which determines the overall height of the flag. **Assert** that the value of the `height` parameter is acceptable. The German flag has a 5:3 aspect ratio. A method that produces the official German flag "gold" color has RGB values 255, 204, 0. The red color is a pure `RED`, and the black color is a pure `BLACK`.
Run in JShell:	`show(GermanFlag.flag(80))`
Output:	

Task A4

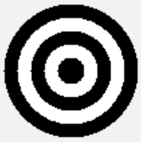
Class:	`FrenchFlag`
Task:	Implement a `flag` method with a `height` parameter of type `double`, which determines the overall height of the flag. **Assert** that the value of the `height` parameter is acceptable. Unlike the German flag, the French flag has a 3:2 aspect ratio. Use the official French flag red (RGB: 239, 65, 53) and blue (RGB: 0, 85, 164) colors. The white color is a pure `WHITE`.
Run in JShell:	`show(FrenchFlag.flag(80))`
Output:	

Task A5

Class:	`ItalianFlag`
Task:	Implement a `flag` method with a `height` parameter of type `double`, which determines the overall height of the flag. **Assert** that the value of the `height` parameter is acceptable. The Italian flag has a 3:2 aspect ratio. Methods that produce the official Italian flag green (RGB: 0, 140, 69), white (RGB: 244, 245, 240) and red (RGB: 205, 33, 42) colors.
Run in JShell:	`show(ItalianFlag.flag(80))`
Output:	

B. Target

Task B1

Class:	`Target`
Task:	Implement a `target` method with a `diameter` parameter of type `double`, which determines the overall diameter of the target. **Assert** that the value of the parameter is acceptable.
Run in JShell:	`show(Target.target(100))`
Output:	

C. Expand your Toolbelt

You had to combine three stripes for the flags, and five circles for the target. You probably used multiple calls to the `above` method (for the German flag), and multiple calls to the `beside` method (for the French and Italian flag), and multiple calls to the `overlay` method (for the target).

This might be quite common, so let's add some combination methods to your toolbelt.

Task C1

Class:	`Toolbelt`
Task:	Implement a `beside3` method that takes three parameters of type `Graphic`, and produces a `Graphic` that arranges all three besides each other. Implement a `above3` method that takes three parameters of type `Graphic`, and produces a `Graphic` that arranges all three above each other. Implement a `overlay5` method that takes five parameters of type `Graphic`, and produces a `Graphic` that arranges all five on top of each other. **Update** your `GermanFlag`, `FrenchFlag`, `ItalianFlag`, and `Target` classes, so they call your new `Toolbelt` methods.

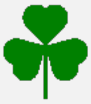
D. Heart

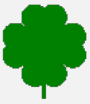
Task D1

Class:	`Heart`
Task:	Implement the `heart` method. The `radius` parameter is the radius of the circular part of the heart. The graphic should have a **pin** on the tip at the bottom. **Assert** that the value of the `radius` parameter is acceptable. Also implement the `purpleColor` method, so it produces a … purple color.
Run in JShell:	`show(Heart.heart(40, Heart.purpleColor()))`
Output:	
Run in JShell:	`show(Heart.loveHeart(20))`
Output:	

E. Clover

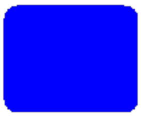
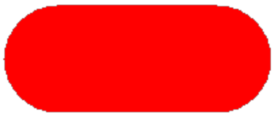
Task E1

Class:	`Clover`
Task:	Looking at the outputs below, you see that a clover consists of hearts. You want to reuse your heart code, now and in the future. So, extract the `heart` method into the toolbelt. Now implement the incomplete methods. The `clover` method should produce a three-leaf clover or a four-leaf clover depending on the `fourLeaves` parameter. The `height` of the stem should be four times the `leafRadius` and its `width` one fourth of the `leafRadius`. **Assert** that the value of the parameters is acceptable.
Run in JShell:	`show(Clover.clover(10, false))`
Output:	

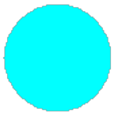
Run in JShell:	`show(Clover.clover(10, true))`
Output:	

F. Rounded Rectangle

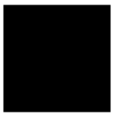
Task F1

Class:	`RoundedRectangle`
Task:	Implement the `roundedRectangle` method. Most probably you want to create additional methods to decompose the problem into smaller ones. Note: It’s ok for a `Graphic` to have width 0 and/or height 0. **Assert** that the value of the parameters is acceptable!
Run in JShell:	`show(RoundedRectangle.roundedRectangle(100, 80, 10, BLUE))`
Output:	
Run in JShell:	`show(RoundedRectangle.roundedRectangle(200, 80, 40, RED))`
Output:	

Run in JShell:	`// what arguments do you need here?`
	`show(RoundedRectangle.roundedRectangle(80,    ,    , CYAN))`

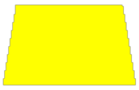
Output:	

Run in JShell:	`// what arguments do you need here?`
	`show(RoundedRectangle.roundedRectangle(80,    ,    , BLACK))`

Output:	

G. Isosceles Trapezoid

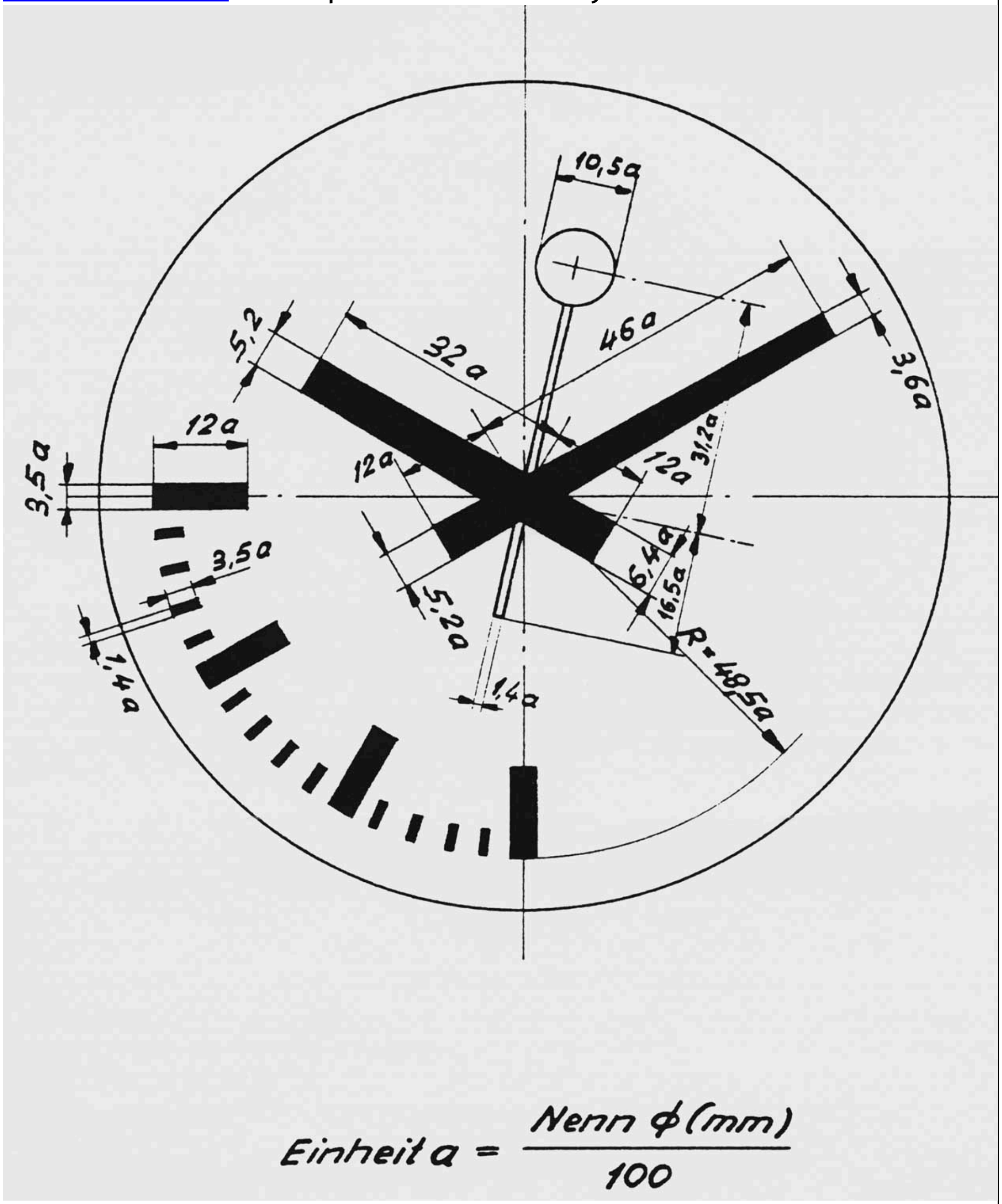
Task G1

Class:	`Trapezoid`
Task:	Implement the `isoscelesTrapezoid` method. Note: It's ok for a `Graphic` to have width 0 and/or height 0. **Assert** that the value of the parameters is acceptable, and that the larger basis is greater than the smaller basis!
Run in JShell:	`show(Trapezoid.isoscelesTrapezoid(100, 80, 60, YELLOW))`
Output:	

H. Swiss Railway Clock

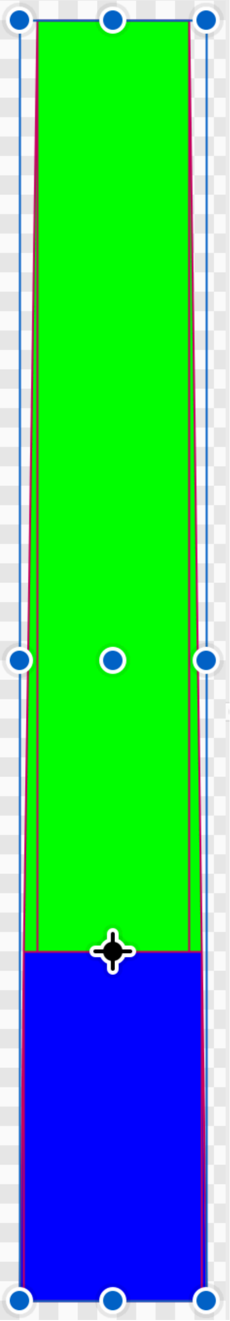
Move the `isoscelesTrapezoid` method from the `Trapezoid` class to the `Toolbelt` class so that you may re-use it to build the clock.

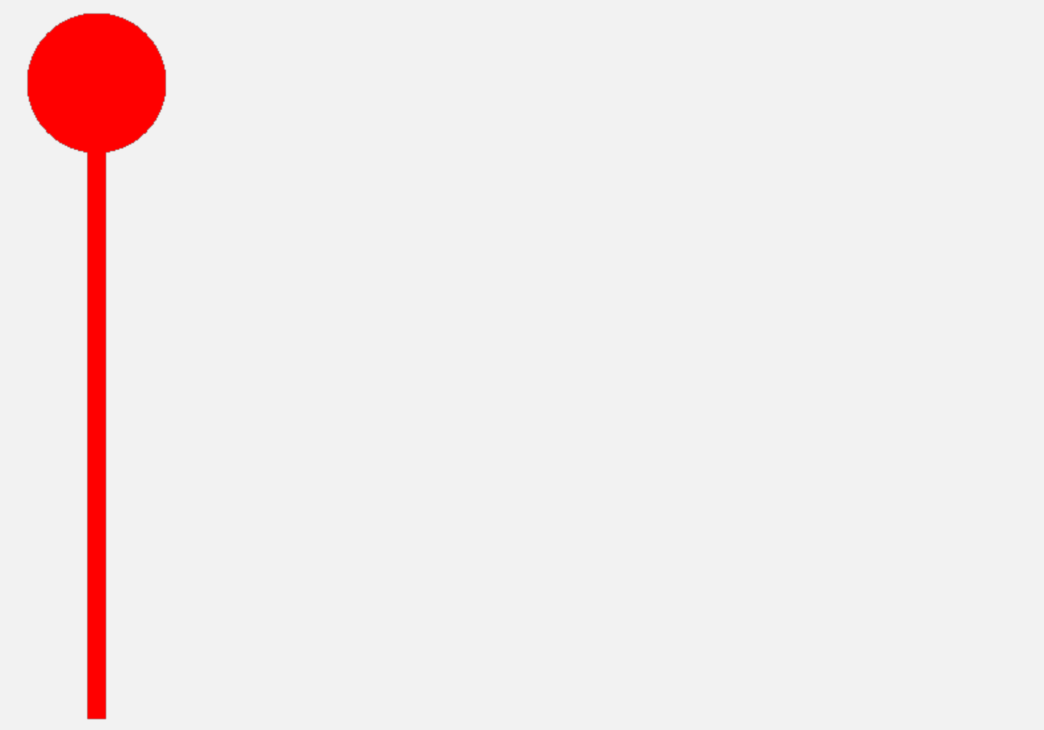
Task H1

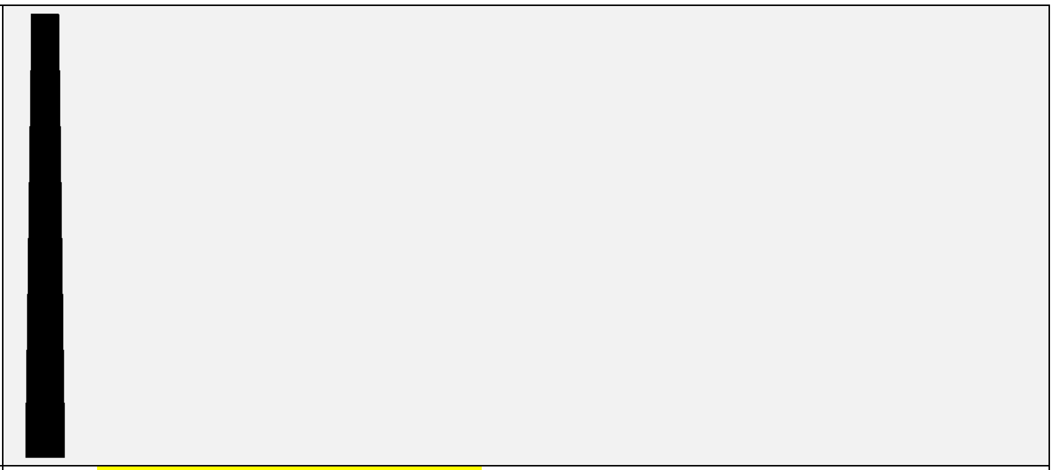
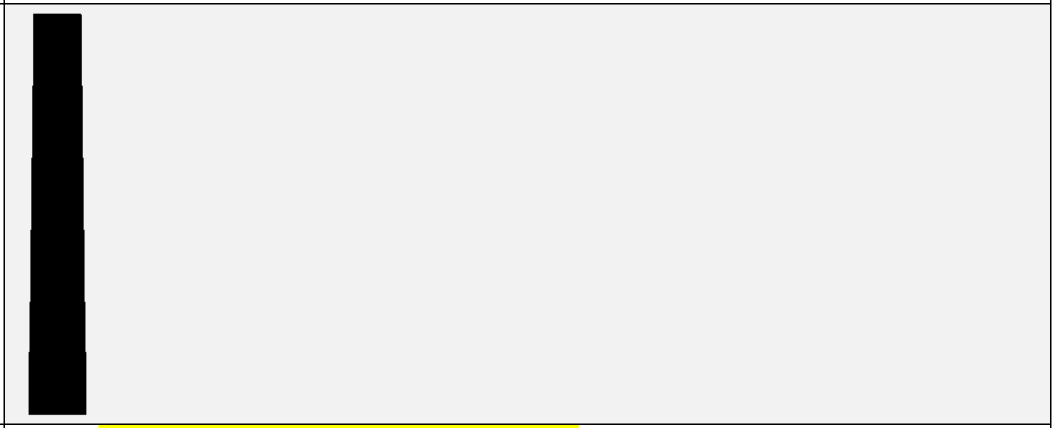
Class:	`SwissClock`
Task:	Implement the `clock` method to draw the Swiss Railway Clock. The parameters determine the size of the unit (α in the technical drawing), and the hours, minutes and seconds, respectively. The original drawing is published on the [Museum für Gestaltung Zürich's website]() and reproduced here for your convenience:  10,5a 46a 32a 5,2 3,6a 12a 3,5a 12a 12a 31,2a 3,5a 6,4a 16,5a 5,2a 1,4a 1,4a R=48,5a Einheit a = Nenn ϕ (mm) / 100 For now, we **IGNORE** the external marks (the 60 smaller marks for the minutes and the 12 bigger marks for the hours).

To tackle this big problem, it's wise to break it down into smaller subproblems. We already did it for you: implement and use the methods defined in the class `SwissClock` to generate the individual components of the clock, and then assemble it.

Hint: after carefully perusing the technical drawing, you will realize that the hands are actually isosceles trapezoids. To build a hand with the pinning position at the appropriate point (so that it can be composed with the center of the dial), we can join two isosceles trapezoids. Here is an example (artificially colored so that the two trapezoids can be distinguished – in the actual clock, both parts of the hand will be black):



	We recommend you implement the methods in the following order. As soon as you progress, invoke your methods in JShell to see if they produce what you expect, as shown below. Implement: • `dial`, to produce the external white circle. • `secondsHand`, to produce the red hand for the seconds. • `hand`, that calls the method `twoPartsHand` providing the length of the “middle base” (the one in common between the two trapezoids), computed using the already implemented `middle` method. • `twoPartsHand`, that actually builds a hand composing two isosceles trapezoids as shown above. Use the `isoscelesTrapezoid` method you added to the toolbelt. • `minutesHand` and `hoursHand`, that call `hand` with the right arguments as defined in the technical drawing. • `toAngle`, that computes the angle of rotation for the different hands (e.g., the minutes hand for 45 minutes needs to be rotated by 90 degrees counterclockwise). • `clock`, to compose all the pieces together. For each method, **assert** that the value of the parameters is acceptable.
Run in JShell:	`show(SwissClock.secondsHand(10))`
Output:	
Run in JShell:	`show(SwissClock.minutesHand(10))`

Output:	
Run in JShell:	`show(SwissClock.hoursHand(10))`
Output:	
Run in JShell:	`show(SwissClock.clock(5, 1, 45, 23))`
Output: