Photo by Marvin Meyer on Unsplash

Lab 4

Mapping · Ghost · Mapping · Rotation · Color Hues · Employees

Copy Your Lab 3 Toolbelt

In Lab 3 you added methods to your `Toolbelt` class. The `Toolbelt` class included in the Lab 4 starter repository is missing these methods. Please **copy the methods of your Lab 3 `Toolbelt`** class into the Lab 4 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future).

Task A – Mapping

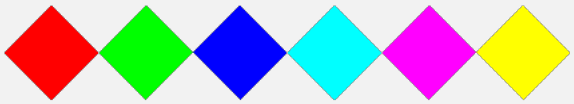
To implement these tasks, **add** to your `Toolbelt` class the `besides` and `aboves` methods from Workbook 04:

```
Graphic besides(Sequence<Graphic> graphics)
Graphic aboves(Sequence<Graphic> graphics)
```

Task A1

Class:	`Mapping`
Task:	Implement the `colorsToRotatedSquares` method. Each rotated squares should be a square with a side of 50 and the given color, rotated by 45 degrees.

Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.colorsToRotatedSquares(Mapping.standardColors())`
	`))`

Output:	

Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.colorsToRotatedSquares(empty())`
	`))`

Output:	

Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.colorsToRotatedSquares(of(BLACK))`
	`))`

Output:	

Task A2

Class:	`Mapping`
Task:	Implement the `stringsToTexts` method. Render the text in `MONOSPACED` font with size 100, in black. Can you figure out what the `intersperse` method, used below, does?

Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.stringsToTexts(of("Hi", "Hello", "BYE!"))`
	`))`

Output:	

Run in JShell:	`show(Toolbelt.besides(`
	`  intersperse(`
	`  Mapping.redDot(),`
	`  Mapping.stringsToTexts(of("Hi", "Hello", "BYE!"))`
	`  )`
	`))`

Output:	

Task A3

Class:	`Mapping`
Task:	Implement the `cosRectangles` method. It should produce a sequence of black rectangles given a sequence of `Double`s. Each rectangle has width `10` and height `100 * Math.cos(num)`, where `num` is the numeric value from the sequence.

Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.cosRectangles(range(0, 2 * Math.PI, 0.1))`
	`))`

Output:	

Task A4

Refactor the methods you implemented in the `Mapping` class. For each of the three methods, create an additional method that performs the desired mapping on a **single** element, and returns a **single** element. In all your existing methods, call the corresponding single-element method to do the mapping of each element.
For example, for Task A2:

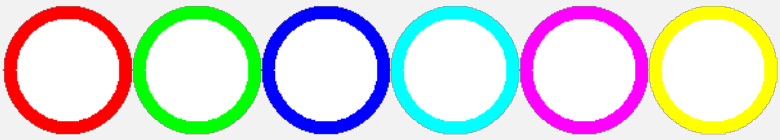
```
public static Sequence<Graphic> stringsToTexts(Sequence<String> strings) { … }

public static Graphic stringToText(String str) { … }
```

Do you see the **similarity** between the three sequence mapping methods?

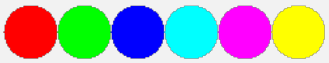
Task A5

Class:	`Mapping`
Task:	Implement a `ring` method to draw a ring with the given `outerDiameter`, `innerDiameter`, and `color`. You can construct the ring by overlaying a white circle on top of a colored circle. Implement the `colorsToRings` method. The `outerDiameter` and `innerDiameter` of the rings should be provided via parameters. **Assert** appropriate parameter values. Use your toolbelt. How does your recursive call differ from the recursive calls in the previous four methods?

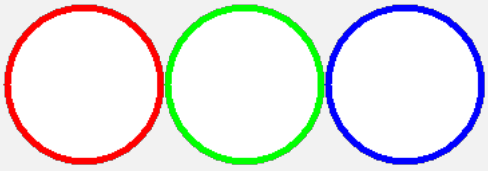
Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.colorsToRings(50, 40, Mapping.standardColors())`
	`))`

Output:	

Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.colorsToRings(20, 0, Mapping.standardColors())`
	`))`

Output:	

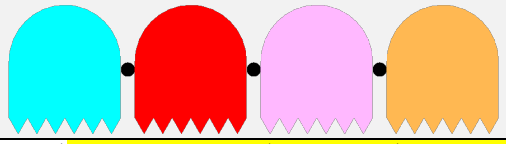
Run in JShell:	`show(Toolbelt.besides(`
	`  Mapping.colorsToRings(60, 55, of(RED, GREEN, BLUE))`
	`))`

Output:	

Task B - Ghost

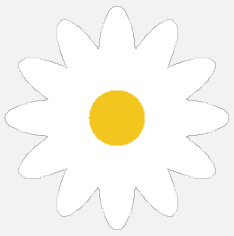
One of the key components of the Pacman game are the adversaries: the four ghosts. Their names are Inky, Blinky, Pinky, and Clyde.

Task B1

Class:	`Ghost`
Task:	Implement the `ghost` and `ghosts` methods. Create helper methods for the meaningful components of a ghost. The `ghost` method takes the ghost width and color as its parameters. The `ghosts` method takes the ghosts width, a `Sequence` of `Color`s and a `boolean` as parameters. For each `Color` in the sequence, a ghost of the corresponding color should be created. The boolean value determines whether the ghosts should be aligned horizontally (`true`) or vertically (`false`). Add meaningful **assert** statements to each method. Use the `besides`, `aboves` and `intersperse` methods when implementing the `ghosts` method!
Run in JShell:	`show(Ghost.ghost(200, Ghost.pinkyColor()))`
Output:	
Run in JShell:	`show(Ghost.ghosts(200, of(Ghost.inkyColor(), Ghost.blinkyColor(), Ghost.pinkyColor(), Ghost.clydeColor()), true))`
Output:	
Run in JShell:	`show(Ghost.ghosts(200, of(Ghost.pinkyColor(), Ghost.clydeColor(), Ghost.inkyColor(), Ghost.blinkyColor()), false))`
Output:	

Task C – Rotation

Task C1

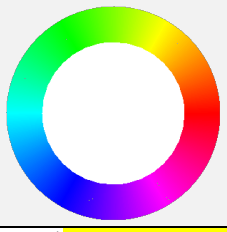
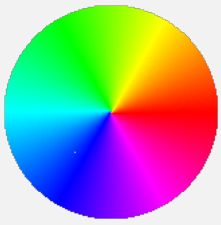
Class:	`Daisy`
Task:	Implement the `daisy` method. Create helper methods for the meaningful components of a daisy, such as a petal, the entire ring of petals, and the bud in the center. A petal should be half the flower's diameter long, and $^1/_3$ as wide as it is long. The diameter of the bud should be $^1/_4$ the diameter of the flower. Methods that produce the colors of the petals and the bud are already provided. Feel free to change the colors to your liking. Implement a `composes` method in your Toolbelt that, similarly to the `aboves` and `besides` methods, takes a `Sequence<Graphic>` and produces a `Graphic` by applying compose to all the graphics in the sequence. Use `range` to create a sequence of integers that contains the angles of each petal. Write a mapping method that maps the sequence of angles to a sequence of rotated petals. Write a method that reduces the sequence of rotated petals by composing them into a single graphic. Add meaningful **assert** statements where needed.
Run in JShell:	`show(Daisy.daisy(200))`
Output:	

Task D - Color Hues

Task D1

Class:	`ColorHues`
Task:	Implement the `hueBar` method. Add helper methods to properly decompose your computation, such as `hueToTile` and `huesToTiles` This should produce a hue bar consisting of 360 tiles, each tile is a rectangle colored with a fully saturated (saturation 1.0), "full light" (value 1.0) color in the corresponding hue (0 to 359). Use the `hsv` method to create the color from the hue, saturation, and value. Each color tile in the hue bar should have height 10. Use the `hsv` method.
Run in JShell:	`show(ColorHues.hueBar())`
Output:	

Task D2

Class:	`ColorHues`
Task:	Implement the `hueRing` method with parameters for the `outerRadius` and the `innerRadius` of the ring. You may want to create a `hueToSector` and a `huesToSectors` method, to **map** from a sequence of hues to a sequence of sectors. Use the composes method you implemented in Task C1 for **reduction**, i.e., to compose a sequence of appropriately rotated and colored circular sectors. **Assert** that parameters have acceptable values. Use your toolbelt.
Run in JShell:	`show(ColorHues.hueRing(150, 100))`
Output:	
Run in JShell:	`show(ColorHues.hueRing(80, 0))`
Output:	

Task E – Employees

In this task we will model the hierarchy of a company with a tree data structure. A node of this tree contains the name of an employee, their salary, and references to their subordinates. Each employee may have an arbitrary number of subordinates (zero, one, or more).

We model a node of our tree with the `Employee` record class.

Your task is to implement a number of methods that traverse an `Employee` tree to gather information.

For convenience when testing your implementations in JShell, you can use the `Employees.demoData()` method that returns an `Employee` instance with the data of a fictional company. It models the following hierarchy:

- Uniquely positioned employee
  - Persistent employee
    - Motivated employee
      - Proactive employee
  - Innovative employee
    - Visionary employee
      - Impactful employee
      - Systematic employee
        - Severe employee
        - Functional employee
        - Amazing employee
      - Exclusive employee
    - Balanced employee

Task E1

Class:	`Employees`
Task:	Implement the `numberOfSubordinates` method, which given an Employee, returns the total number of (direct and indirect) subordinates. **Hint**: traverse the tree using by writing recursive functions to count the number of subordinates. **Hint**: remember to not include the given employee when counting, as they are not their own subordinate!
Run in JShell:	`Employees.numberOfSubordinates(Employees.demoData())`
Output:	`==> 12`

Task E2

Class:	`Employees`
Task:	Implement the `totalSalary` method, which given an `Employee`, returns the total amount of salary of the given employee and all their subordinates. **Hint**: traverse the tree using by writing recursive functions to sum the salaries.
Run in JShell:	`Employees.totalSalary(Employees.demoData())`
Output:	`==> 737`

Task E3

Class:	`Employees`
Task:	Implement the `nameOfSubordinates` method, which given an `Employee`, returns a sequence containing the names of all their **subordinates.** **Note:** The order of the elements in the sequence matters! When traversing the tree of employees, proceed *depth–first* and in *pre-order*. In our example, this implies that the "Motivated employee" should come before the "Innovative employee".
Run in JShell:	`println(Employees.nameOfSubordinates(Employees.demoData()))`

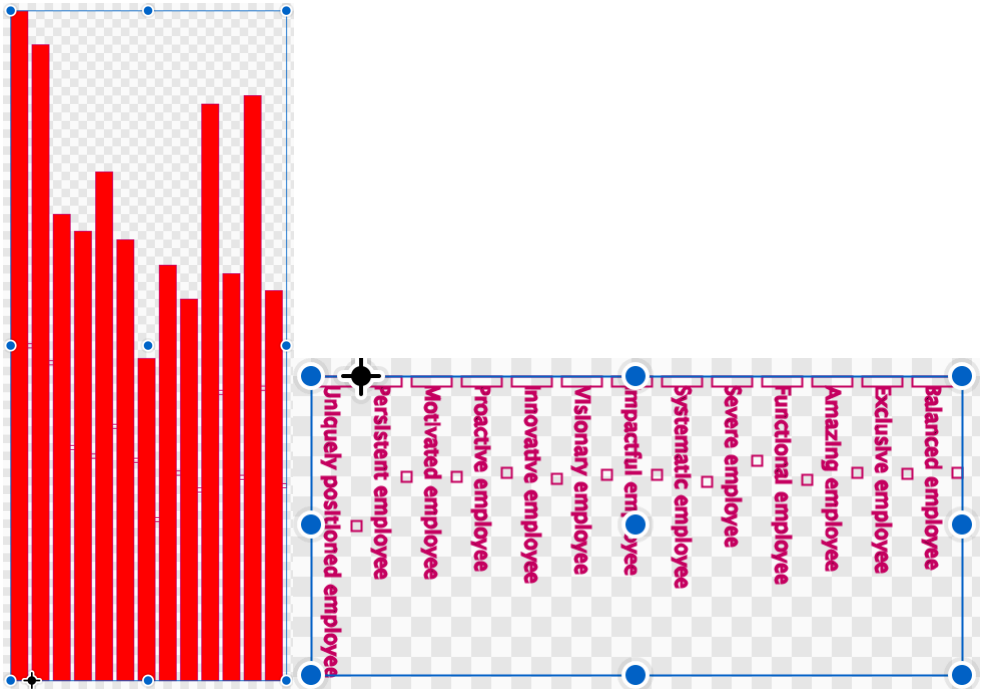
Output:	`Persistent employee`
	`Motivated employee`
	`Proactive employee`
	`[…]`
	`Exclusive employee`
	`Balanced employee`

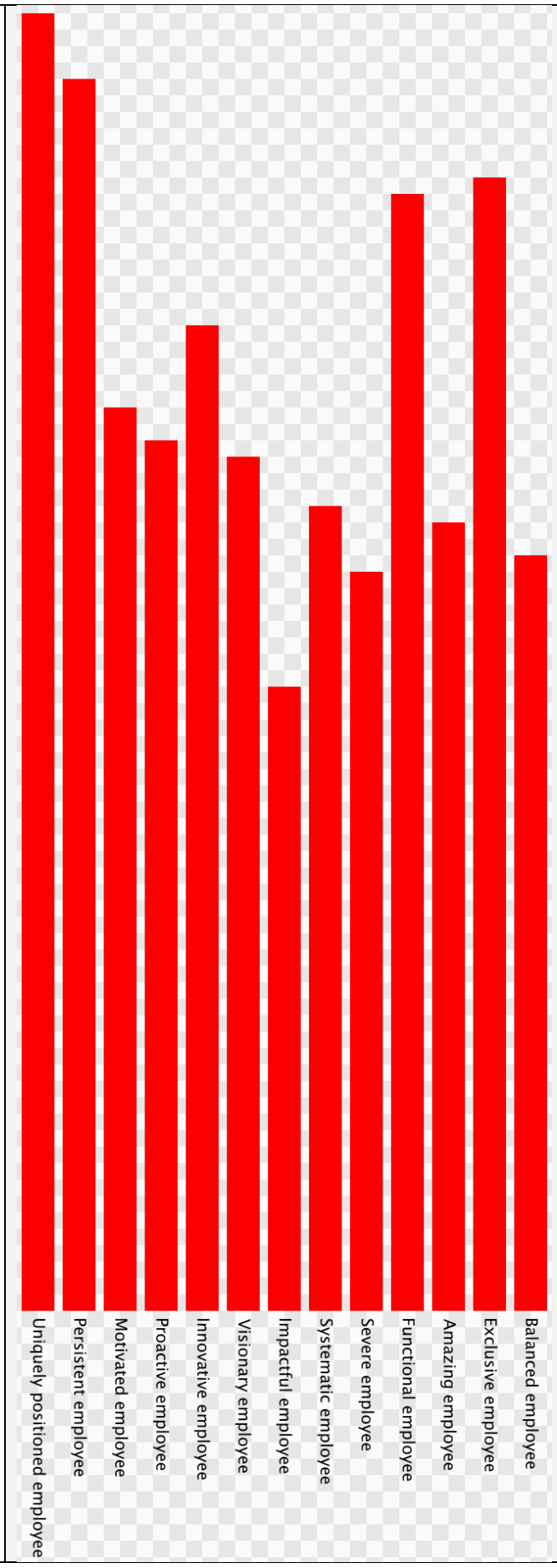
Task E4

Class:	`Employees`
Task:	Implement the `flatten` method, which given a hierarchy of employees (an `Employee` instance), returns a sequence containing all employees found within that hierarchy. **Note**: this list includes the given employee as well, not just their subordinates! **Note:** The order of the elements in the sequence matters! When traversing the tree of employees, proceed *depth–first* and in *pre-order*. **Note**: The example output presented below will differ slightly from your output. The textual representation of the `subordinates` field has an hexadecimal suffix which is irrelevant and you should ignore.
Run in JShell:	`println(Employees.flatten(Employees.demoData()))`

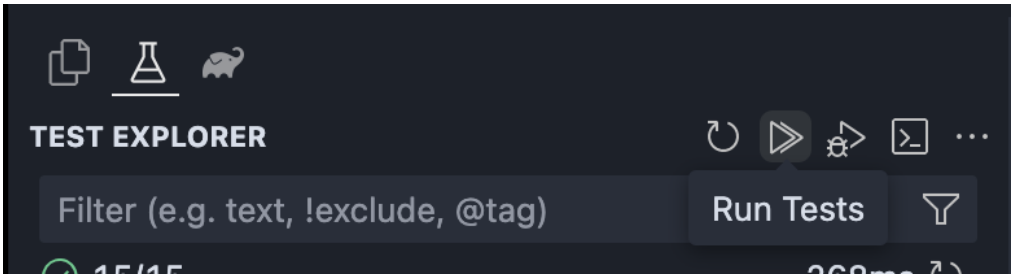
Output:	`Employee[name=Uniquely positioned employee, salary=79,`
	`  subordinates=jtamaro.en.data.Cons@33f88ab]`
	`Employee[name=Persistent employee, salary=75,`
	`  subordinates=jtamaro.en.data.Cons@27a8c74e]`
	`Employee[name=Motivated employee, salary=55,`
	`  subordinates=jtamaro.en.data.Empty@2d8f65a4]`
	`[…]`
	`Employee[name=Exclusive employee, salary=69,`
	`  subordinates=jtamaro.en.data.Empty@6ee52dcd]`
	`Employee[name=Empowering employee, salary=46,`
	`  subordinates=jtamaro.en.data.Empty@4493d195]`

Task E5

Class:	`Employees`
Task:	Implement the `plotSalaries` method, which given a sequence of employees (a `Sequence<Employee>` instance), returns a graphic which depicts a bar plot of the salaries of the various employees. The plot has two parts: the bars that indicate the salaries and the names of the employees.  To draw the bars, create rectangles of width `salaryPlotBarWidth()` and height equal to the salary value for each `employee`. Then, add a transparent rectangle of width `salaryPlotBarPadding()` in between the bars to pad them out. To render the names, create a text of size 10 with color black and rotate it. Add some vertical padding (again of size `salaryPlotBarPadding()`) to the rendered name, so that the label will have some minimal separation from the bars. Make sure the text is horizontally aligned with the bars (hint: look at the picture above and use the value of `salaryPlotBarWidth()`). Finally, put the two parts one above the other.
Run in JShell:	`show(Employees.plotSalaries(Employees.flatten(Employees.demoData())))`
Output:	(see next page)


Uniquely positioned employee
Persistent employee
Motivated employee
Proactive employee
Innovative employee
Visionary employee
Impactful employee
Systematic employee
Severe employee
Functional employee
Amazing employee
Exclusive employee
Balanced employee

Task E6

Class:	`EmployeesTest`
Task:	To avoid having to "test" our growing implementation manually all the time, we now make use of the testing code from a so-called test class (instead of typing it into JShell over and over again). The class `EmployeesTests` (found in the `src/test/java/lab` directory) provides a comprehensive number of tests that will help you verify whether your solution behaves correctly with respect to the given specifications. The convention we will follow here is: the tests for the hypothetical method `myMethod` of the hypothetical class `MyClass` will be placed in the methods which names start with `testMyMethod` of the hypothetical class `MyClassTest`. If you have an expression `codeToTest` and you want to check that it evaluates to a certain expected value (or expression) `expectedValue`, call `Assert.assertEquals` as follows: `Assert.assertEquals(expectedValue, codeToTest);` This is very similar to the function `check-expect` in **Racket**. There you would write (`check-expect actual-expression expected-expression`) to check that `actual-expression` evaluates to the same value as `expected-expression`. In Racket, we place calls to `check-expect` at the top-level of a program. Here, calls to `assertEquals` are placed inside methods of a JUnit test class. Open the “Testing view” of VSCode (flask icon) and execute the tests by clicking on the “Run tests” icon at the top (“double play” icon):  If some test fails, it means that there’s some problem with your code. Identify which method is not working correctly by looking at the failed test names and their assertions and fix your implementation of said method. **DO NOT** edit the tests’ code. The tests are there to help you uncover certain situations you might have not accounted for.

Output:	All tests should pass: