Photo by Marvin Meyer on Unsplash

Lab 8

Data Science (Fancy Bar Chart) · Pacman Game (First Part)

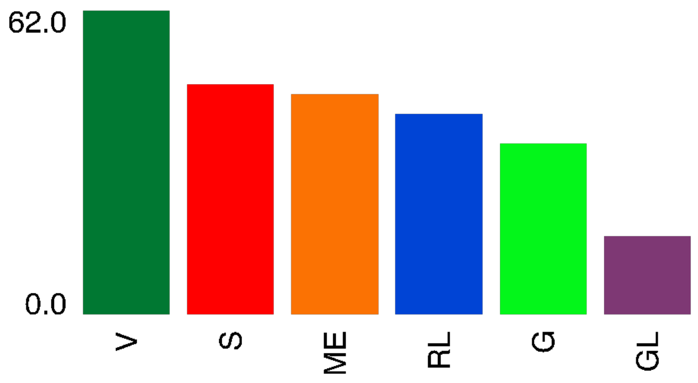
Copy Your Lab 7 Toolbelt

In Lab 7 you added methods to your `Toolbelt` class. The `Toolbelt` class included in the Lab 8 starter repository is missing these methods. Please **copy the methods of your Lab 7 `Toolbelt`** class into the Lab 8 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future).

A. Data Science (Fancy Bar Chart)

In a previous lab we developed a simple bar char. Let's be more ambitious and develop `FancyBarChart`, a more powerful class that can render bars in different colors, add labels below bars, and add labels for the range (minimum and maximum) on the y-axis.

Here is an example bar chart based on the Swiss Parliamentary Groups data:



We will use our `FancyBarChart` class on two data sets, which are available in classes `Lugano` and `SwissParliament`.

Task A1

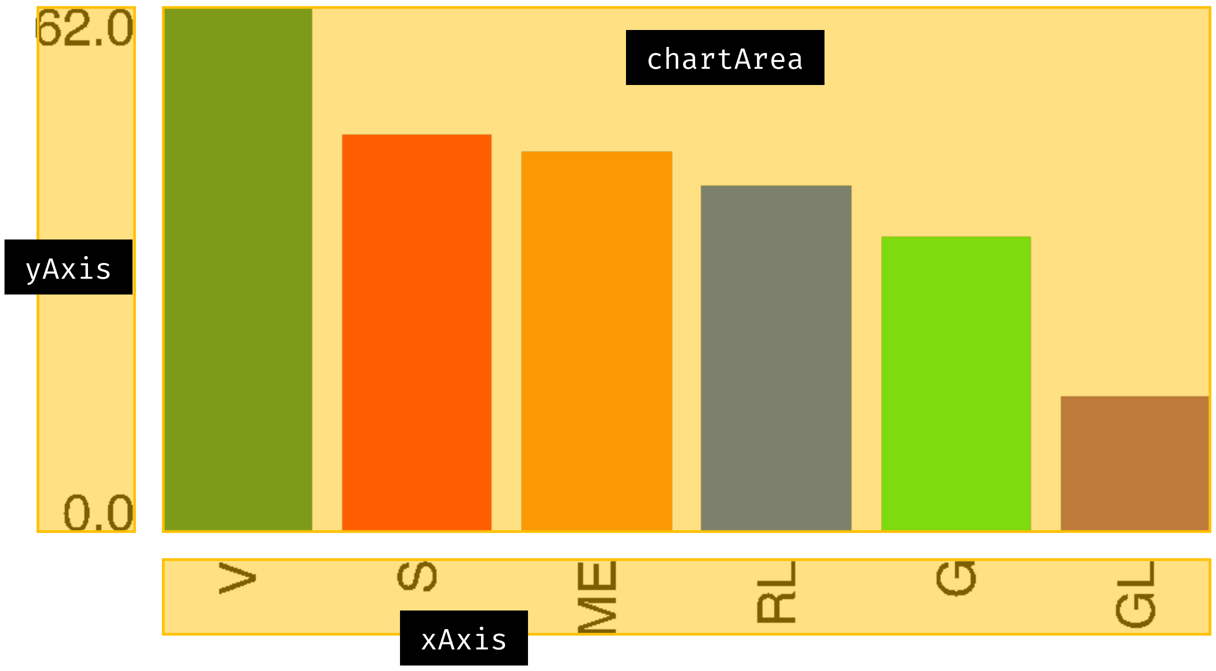
Class:	`QuickSort`
Task:	We will want to sort data, so first copy the `sort` method from the worksheet into the `QuickSort` class.

Run in JShell:	
	```
	println(
	  QuickSort.sort(
	    (a, b) -> a <= b,
	    of(5, 1, 2, 4, 3)
	  )
	)
	```

Output:	
	```
	1
	2
	3
	4
	5
	```

Tests:	`All tests in lab.barchart.QuickSortTest should pass`

Let's decompose the graphic, so that we can create a bunch of functions that create the different parts.



Note that there is a gap between the `yAxis` and the `chartArea`, and an equally wide gap between the `xAxis` and the `chartArea`. Those gaps have the same width as the gap between bars.

The `chartArea` is like the one you have implemented in the past. The only difference is that the bars also have different colors.

The `yAxis` is rather simple: it has the same height as the `chartArea`, and it has a text at its top and a text aligned at its bottom.

The `xAxis` consists of rotated texts, centered below each bar. It may make sense to create rectangles with the same width as one bar, and then center the rotated text within those rectangles.

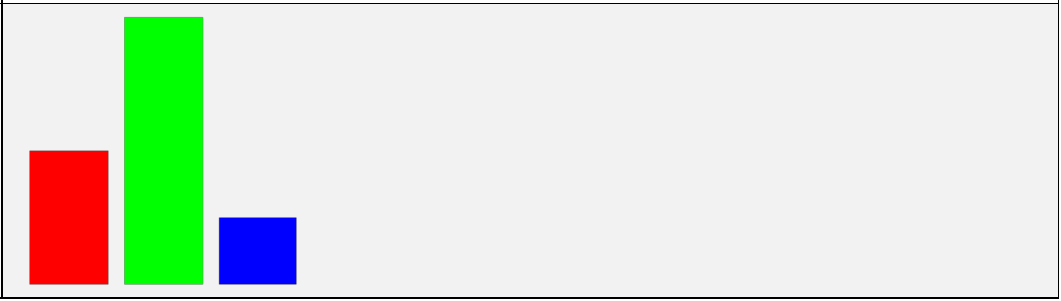
While implementing your `FancyBarChart`, use **lambdas** wherever it's useful.

Task A2

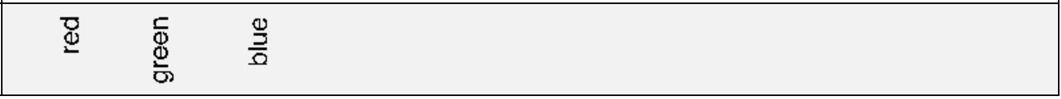
Class:	`FancyBarChart`

Task:	Implement a method to draw the chart area.
	
	`public static Graphic chartArea(`
	`  Sequence<Double> values,`
	`  Sequence<Color> colors,`
	`  double width,`
	`  double height`
	`)`
	
	Hint: You can `zip` the two sequences and do a single reduction.
	
	Assert acceptable parameter values.

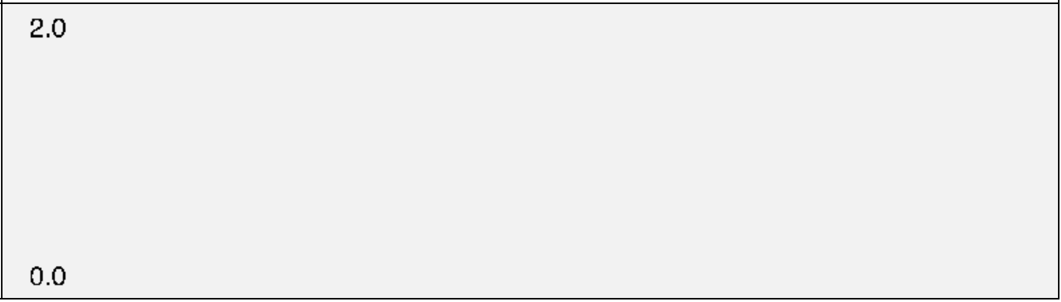
Run in JShell:	`show(`
	`  FancyBarChart.chartArea(`
	`    of(1.0, 2.0, 0.5),`
	`    of(RED, GREEN, BLUE),`
	`    200,`
	`    200`
	`  )`
	`)`

Output:	

Task A3

Class:	`FancyBarChart`
Task:	Implement a method to draw the x-axis. `public static Graphic xAxis( Sequence<String> labelStrings, double width, double fontSize )` Use the given font size for the texts. **Assert** acceptable parameter values.
Run in JShell:	`show( FancyBarChart.xAxis( of("red", "green", "blue"), 200, 20 ) );`
Output:	

Task A4

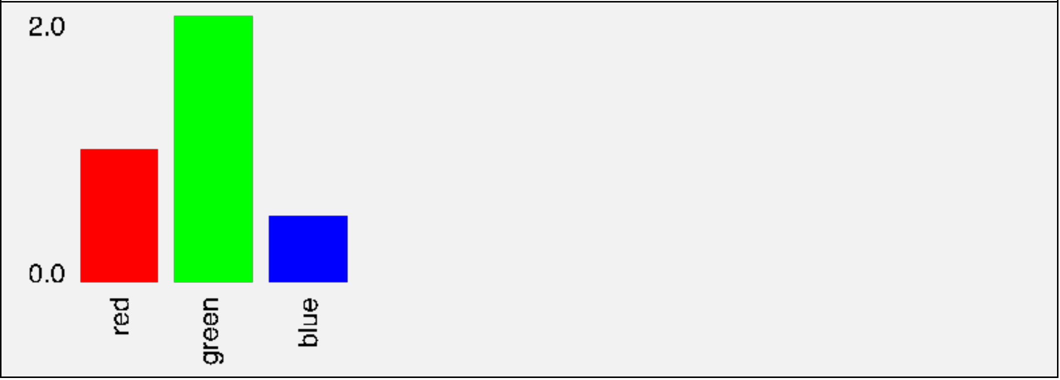
Class:	`FancyBarChart`
Task:	Implement a method to draw the y-axis. `public static Graphic yAxis( Sequence<Double> values, double height, double fontSize )` Use the given font size for the texts. **Assert** acceptable parameter values.
Run in JShell:	`show( FancyBarChart.yAxis( of(1.0, 2.0, 0.5), 200, 20 ) )`
Output:	

Task A5

Class:	`FancyBarChart`

Task:	Implement a method to compose the entire bar chart.
	
	`public static Graphic barChart(`
	`  Sequence<Double> values,`
	`  Sequence<String> labels,`
	`  Sequence<Color> colors,`
	`  double areaWidth,`
	`  double areaHeight,`
	`  double fontSize`
	`)`
	
	Call `chartArea`, `xAxis`, and `yAxis`, and inject gaps between the different parts.

Run in JShell:	`show(`
	`  FancyBarChart.barChart(`
	`    of(1.0, 2.0, 0.5),`
	`    of("red", "green", "blue"),`
	`    of(RED, GREEN, BLUE),`
	`    200,`
	`    200,`
	`    20`
	`  )`
	`)`

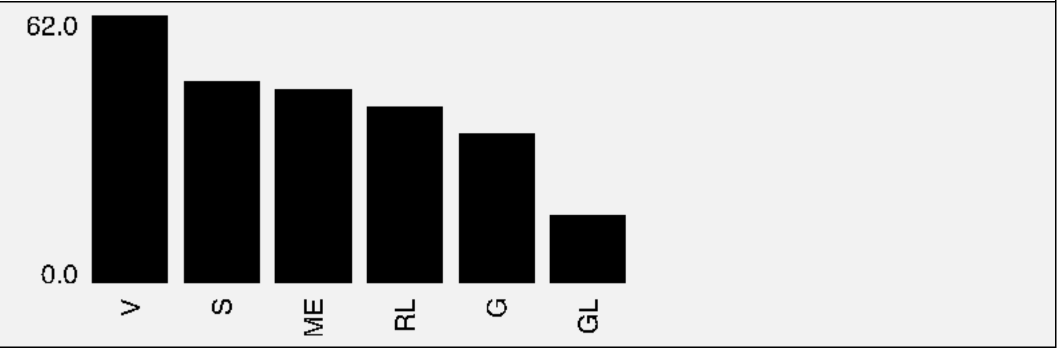
Output:	

Task A6

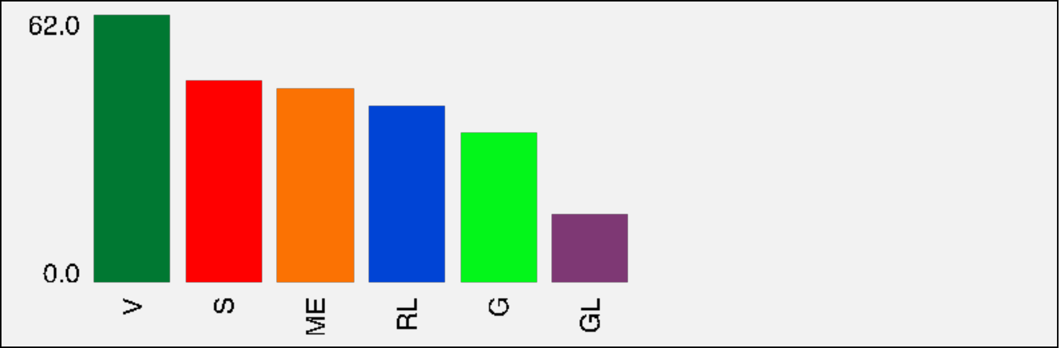
Class:	`FancyBarChart`

Task:	Implement a convenience method to compose a bar chart from a sequence of some type of records. Instead of taking three sequences (values, labels, colors), this method just takes one sequence (e.g., a sequence of `Neighborhoods`), and it takes three functions that map from elements of the given sequence to a sequence of values (bar heights), a sequence of strings (bar labels), and a sequence of colors (bar colors).
	
	`public static <T> Graphic barChartFromRecords(`
	`  Sequence<T> items,`
	`  Function1<T,Double> itemValue,`
	`  Function1<T,String> itemLabel,`
	`  Function1<T,Color> itemColor,`
	`  double width,`
	`  double height,`
	`  double fontSize`
	`)`
	
	Assert acceptable parameter values.

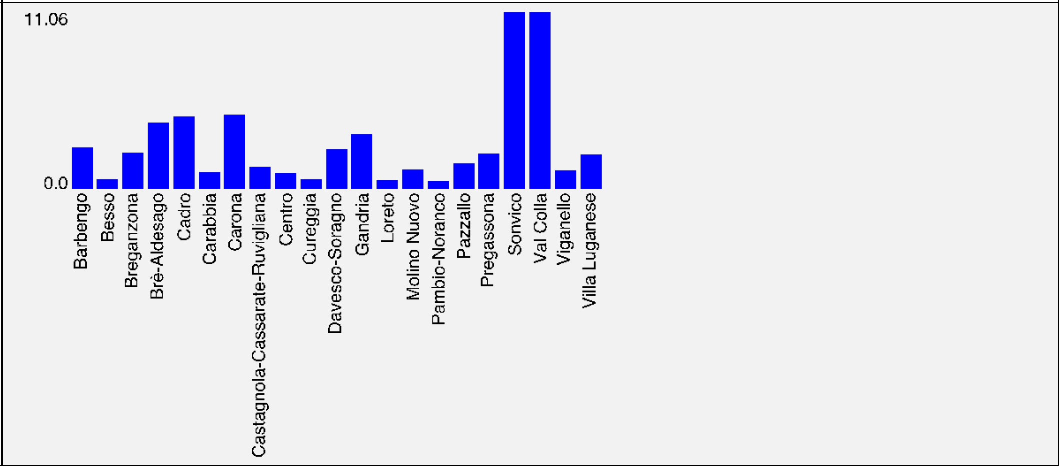
Run in JShell:	`show(`
	`  FancyBarChart.barChartFromRecords(`
	`    SwissParliament.parliamentaryGroups(),`
	`    g -> (double)g.members(),`
	`    g -> g.acronym(),`
	`    g -> BLACK,`
	`    400,`
	`    200,`
	`    20`
	`  )`
	`)`

Output:	

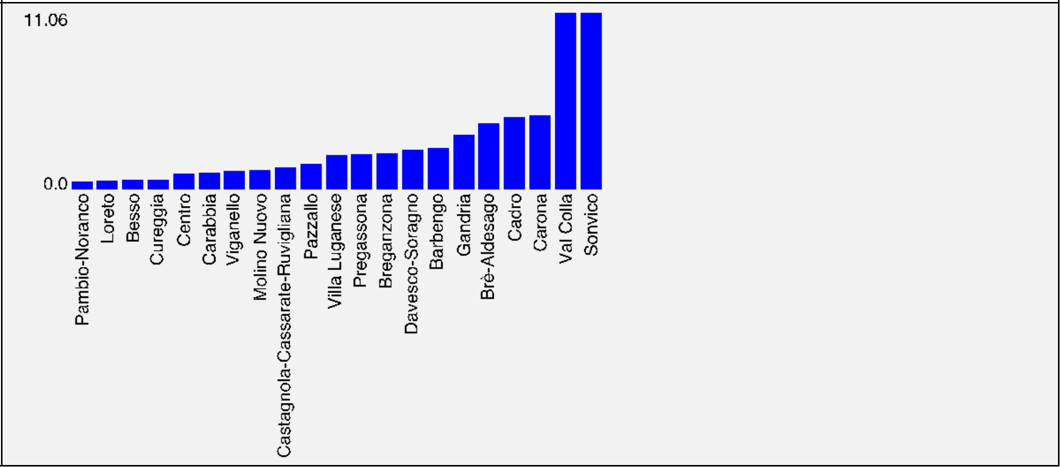
Task A7

Class:	`ExampleCharts`
Task:	Write a method that creates a bar chart, like the one just before, but with bars colored by the parliamentary group's colors. **`public static`** `Graphic swissParliamentSeatDistribution()` The `SwissParliament` class contains a method `acronymToColor` that helps you to map from an acronym of a parliamentary group to the color that is usually used to represent that group.
Run in JShell:	`show(ExampleCharts.swissParliamentSeatDistribution())`
Output:	

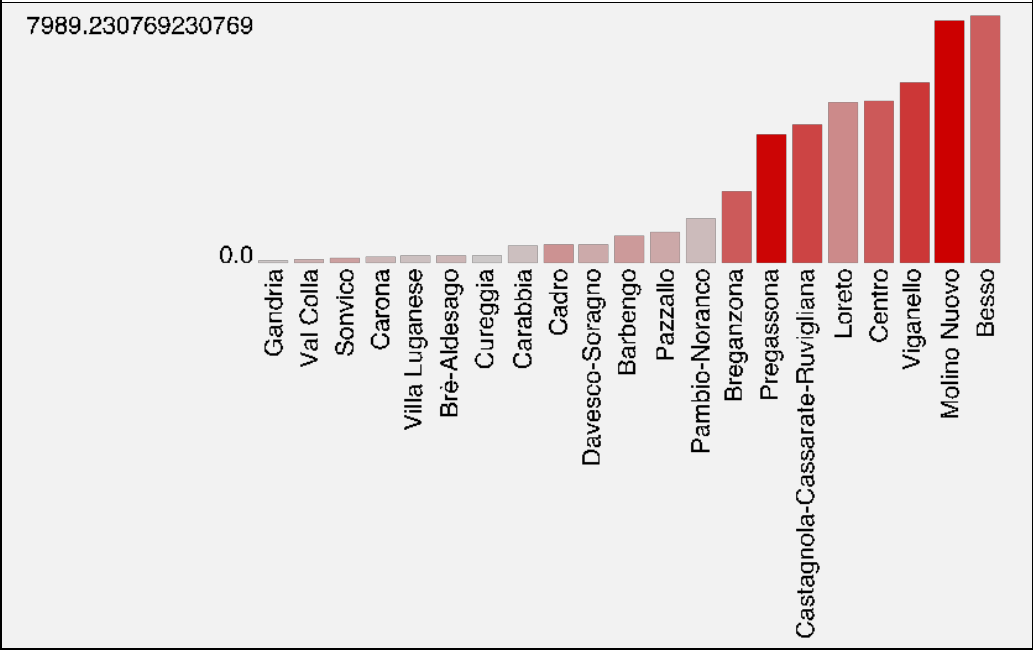
Task A8

Class:	`ExampleCharts`
Task:	Write a method that creates a bar chart of Lugano neighborhoods, in alphabetical order, showing their areas. **`public static`** `Graphic neighborhoodAreasByName()` Use method `luganoNeighborhoods` in class `Lugano`.
Run in JShell:	`show(ExampleCharts.neighborhoodAreasByName());`
Output:	

Task A9

Class:	`ExampleCharts`
Task:	Write a method that creates a bar chart of Lugano neighborhoods, in increasing order of their area, showing their areas. **`public static`** `Graphic neighborhoodAreasByArea()` Use method `luganoNeighborhoods` in class `Lugano`.
Run in JShell:	`show(ExampleCharts.neighborhoodAreasByArea());`
Output:	

Task A10

Class:	`ExampleCharts`
Task:	Write a method that creates a bar chart of Lugano neighborhoods, in increasing order of their population densities. The height of the bar should correspond to the population density (population / area), and the saturation of the bar should correspond to the population. **`public static`** `Graphic neighborhoodPopulationDensities()` **Use** `hsv` to create the color, with a hue of 0, a value of 0.8, and a saturation that corresponds to the population (0 for a population of 0, 1.0 for a population that corresponds to the maximum population across neighborhoods).
Run in JShell:	`show(ExampleCharts.neighborhoodPopulationDensities());`
Output:	

Isn't this pretty darn amazing?

You built this from scratch. All you had was a very limited graphics library. Everything was composed by you!

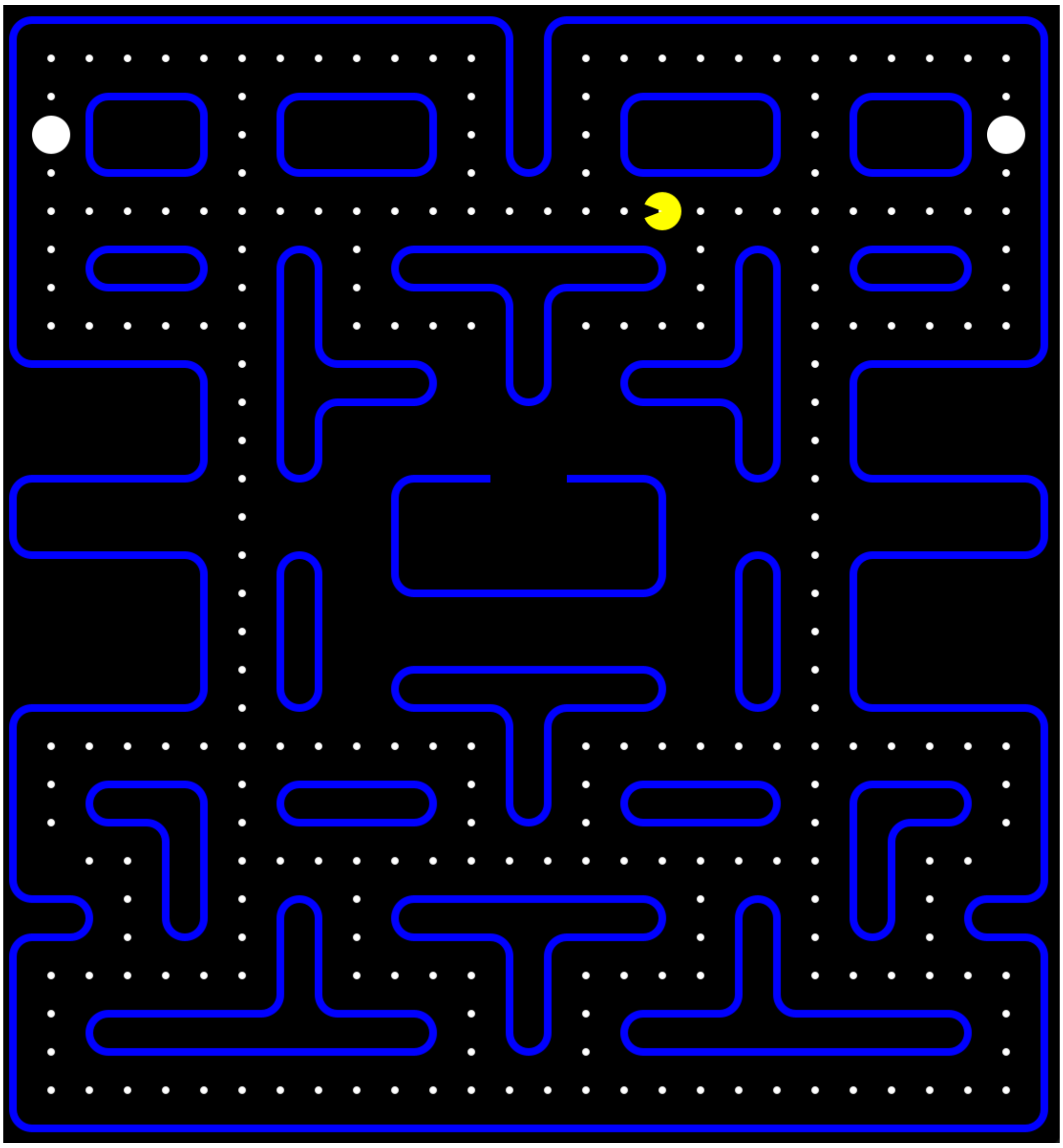
Given your fancy bar chart method, you can now throw together insightful visualizations of any sequence of records with just a few lambdas.

All you need is lambda.

B. Pacman Game (First Part)

In an earlier lab, we decomposed the maze of a Pacman game and rendered the different tiles (e.g., a pill, a straight wall, a corner wall, the pacman itself).
We then *manually* combined the graphics to create a simple small maze.

In this lab, we want to *programmatically* draw a realistic and configurable maze. We will then implement the necessary functionalities so that our pacman can move around across tiles. Our result will look like this:

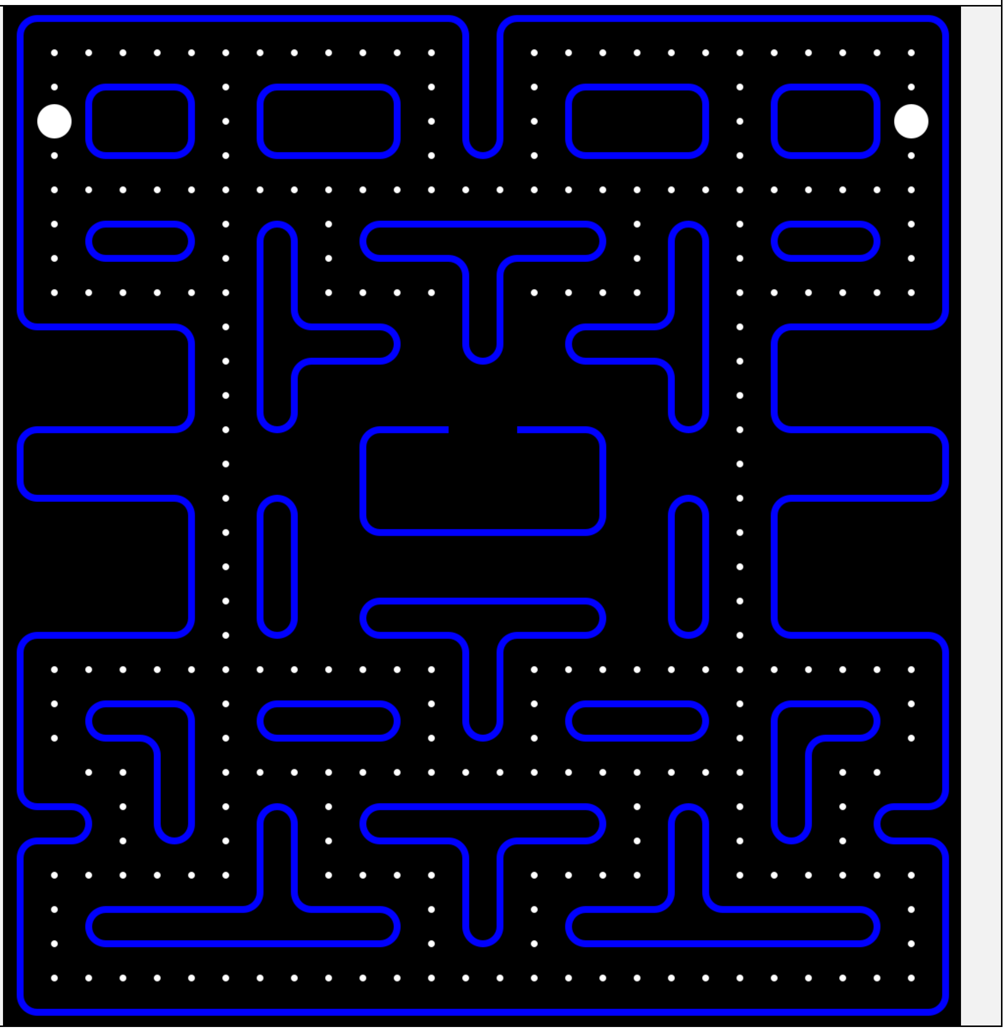


In a future lab, we'll return on what we accomplished here to complete our game (e.g., adding a ghost, constraining the pacman so that it cannot step on walls, and detecting the collision between the pacman and the ghost).

Task B1

Class:	`Main`
Task:	We describe the layout of our maze as a string of characters, where each one resembles one of the kinds of tiles. For example, ┐ corresponds to an upper-right wall corner, ─ to a straight horizontal wall, and `0` to a power pill. With this encoding, it's relatively straightforward to write the configuration of a maze in a file. Each line corresponds to a row of our maze, and each character corresponds to a column in that row (i.e., a tile). Start by implementing `readMazeDescriptor` in class `Main`. It takes a `Path` as an argument, reads the file and returns a grid (i.e., a sequence of sequence) of `Characters`. **Hint**: You can read the content of a file as a string by calling `IO.readFile(Path)`. **Hint:** You can split a string at every "new line character" and get back a sequence of strings (one per line) with `ofStringLines`. **Hint:** You can split a string at every character and get back a sequence of characters (one per line) with `ofStringCharacters`. You may find it convenient to use `map`.
Run in JShell:	`print(first(Main.readMazeDescriptor(Path.of("level.txt"))))`
Result:	`┌────────────────────────┐ ┌────────────────────────┐`

Task B2

Class:	`Maze`
Task:	Complete the implementation of the method `charToTileGraphic` in `Maze` that renders one character into the appropriate graphic. **Be Lazy (aka: Reuse Code).** The implementation should call the methods you already implemented in Lab 3 Task F. Each square tile should have side length `tileSize` (a parameter of `charToTileGraphic`). Then, complete the method `render` that transforms a description of an entire maze (a grid of characters) into the entire graphic.
Run in JShell:	`show(Maze.render(20.0, Main.readMazeDescriptor(Path.of("level.txt"))))`
Output:	

Task B3

Class:	`Maze`
Task:	A maze is modeled as a record with two components (`numColumns` and `numRows`) that describe the number of tiles in each dimension. The position of the pacman on the maze is modeled as a `Position`. `Position` is a record with two components, `column` and `row`. When the pacman will move, we want to confine it within the boundaries of our maze. Implement a predicate `isWithinBounds` that takes a position and checks whether it fits within the boundaries of the maze. Then, inside the method play of the class Main, instantiate a `Maze` object providing the correct dimensions of the maze. Inspect the grid of characters contained in `mazeDescriptor`.
Run in JShell:	`new Maze(10, 10).isWithinBounds(new Position(2, 3))`
Output:	`==> true`
Run in JShell:	`new Maze(16, 18).isWithinBounds(new Position(15, 52))`
Output:	`==> false`

Task B4

Class:	`Heading`
Task:	As we already saw in previous labs, a pacman has a *heading*: a direction towards which it faces. Now we want an interactive game, and therefore our pacman has to move. But to do that properly, we need to take into account its heading. The next position of the pacman depends indeed on its current heading. Instead of implementing this in a method with several nested conditional expressions, we can exploit the beauty of object-oriented programming and take advantage of dynamic dispatch. To do this, let's model each heading (north, south, west, east) as a record class implementing a common interface, `Heading`. What can a heading tell us? We need two capabilities from a specific heading: being able to know the angle of rotation to appropriately render the pacman and being able to compute the next position.

	Create four record classes named `North`, `South`, `West`, `East` that implement `Heading` and its two methods.
Run in JShell	`new lab.pacman.game.heading.North().toRotation()`
Output:	`90.0`
Run in Jshell	`new lab.pacman.game.heading.South().nextPosition(new Position(1, 1))`
Output:	`==> Position[column=1, row=2]`

Task B5

Class:	`HeadingFactory`
Task:	We now need to map from a key pressed on the keyboard to one of our headings, so that we will be able to change the pacman's heading once everything is properly wired. Complete the `fromKeyCode` method and instantiate the correct heading in each case using the record classes created in the previous task.
Run in JShell:	`HeadingsFactory.fromKeyCode(38, new lab.pacman.game.heading.South())`
Output:	`==> North[]`

Task B6

Class:	`Configuration, Game`
Task:	We now are ready to model our protagonist, pacman. Define an `INITIAL_PACMAN` constant of type `Pacman` in the `Configuration` class as follows: - Heading west - At position 0, 0 - Mouth angle set to 30 Use this constant in the `Game.initialGame` method to define the default state of the Pacman when the game starts.
Run in JShell:	`Configuration.INITIAL_PACMAN`
Output:	`==> Pacman[heading=West[], position=Position[column=0, row=0], mouthAngle=30]`

Task B7

Class:	`Pacman`
Task:	Implement the `turn` method, which should create a new pacman that is the same as the current one but with an updated heading. This method will get called when we press a key on the keyboard. The method should have this signature: `Pacman turn(Heading newHeading)`

Run in JShell:	`new Pacman(new lab.pacman.game.heading.East(),`
	`           new Position(2, 2),`
	`           30).turn(new lab.pacman.game.heading.West())`

Output:	`==> Pacman[heading=West[], position=Position[column=2, row=2],`
	`                 mouthAngle=30]`

Task B8

Class:	`Pacman`
Task:	Implement the `evolve` method, which defines the core behavior of the pacman at each step of our game (or, if you prefer, each time our "world" evolves). There are two key functionalities: - The pacman tries to move one step in the direction of its current heading - The pacman assumes a new mouth angle Let's implement each functionality in its own little helper method. For the first one, define a method `Position nextPosition(Maze maze)` To implement it, use the method `nextPosition` on the pacman's **heading** to compute a tentative new position. You should then make sure that the position fits within the boundary of the maze (remember the maze's `isWithinBounds` method we implemented earlier). If it fits, the "next position" is the tentative one. Otherwise, the pacman does not move. For the second functionality, define a parameterless method `nextMouthAngle` that returns an `int`. Each time the mouth opens 6 degrees more, until it reaches 60 degrees. At that point, it closes completely (0 degrees).

	Finally, define and implement `Pacman evolve(Maze maze)` that creates a new pacman with the updated position and mouth angle, as returned by the two auxiliary methods.

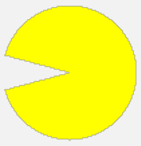
Run in JShell:	
	```
	new Pacman(new lab.pacman.game.heading.East(),
	           new Position(2, 2),
	           30).evolve(new Maze(10, 10))
	```

Output:	
	```
	==> Pacman[heading=East[], position=Position[column=3, row=2],
	           mouthAngle=36]
	```

Task B9

Class:	`Pacman`
Task:	Define and implement the `render` method: `Graphic render(double tileSize)` You can reuse the graphic for the pacman you already produced in the previous lab, but make sure its width is exactly the size of a tile (the parameter). Also, take into account the `mouthAngle`.

Run in JShell:	
	```
	show(new Pacman(new lab.pacman.game.heading.West(),
	           new Position(0, 0),
	           30).render(100.0))
	```

Output:	

Task B10

Class:	`Game`
Task:	We now focus on the `Game` class. It contains all the information about the state of the game.
	Implement the `turnPacman` method, which returns an updated game instance with a pacman that is facing towards the heading provided as a parameter.
	Be smart: this method should delegate the work of turning the pacman to the `Pacman` object (indeed, earlier we implemented `Pacman.turn`).
Run in JShell:	`new Game(`
	`  Configuration.INITIAL_PACMAN,`
	`  new Maze(8, 8)`
	`).turnPacman(new lab.pacman.game.heading.South())`
Output:	`==> Game[pacman=Pacman[heading=South[],`
	`                       position=Position[column=0, row=0],`
	`                       mouthAngle=30],`
	`          maze=Maze[numColumns=8, numRows=8]]`

Task B11

Class:	`Game`
Task:	Implement the `evolve` method of `Game` class, which acts on the entire game and “evolves” it one step.
	For now, that just means evolving the pacman. In a future lab, here we will also evolve all the other sprites, such as ghosts.
Tests:	`All tests in lab.pacman.game.GameTest should pass`

Task B12

Class:	`Main`
Task:	Implement the `interaction` at the end of the body of the `play` method. Call `IO.interact` with one argument: the initial `Game` object, which you may obtain by invoking the `Game.initialGame` method. Then, using the fluent API we've seen in the previous lab, configure the interaction using the following `with…` methods: - `withBackground(…)` – pass the background Maze graphic which you may produce with the `Maze.render` method. - `withRenderer(…)` – pass a lambda that, given a game instance, renders it using the `Game.render` method. - `withKeyPressHandler(…)` – pass a method reference to the `Main.onKeyPress` method. - `withTickHandler(…)` – which evolves the given Game instance (method `Game.evolve`). Finally, conclude the chain of method invocations with `.run()` to execute your interaction. Now, call the `Main.play` method from JShell to run the pacman game! At each step, the library will call the `onTick` event handler, which in turn evolves the entire `Game` using the method you implemented. The maze is rendered once at the beginning, and then at each step the `render` method is invoked on your `game` instance. Finally, a keyboard handler reacts to key presses, producing a new heading for the pacman and updating the game. If everything has been implemented correctly, you should be able to control the direction pacman moves using the directional keys on your keyboard. The pacman should stay within the boundaries of our world, but it still doesn't meaningfully interact with the maze (e.g., it goes over the walls). The maze description is contained in a file named `level.txt`. Try to change its content and verify that your fantasy gets reflected in the rendered maze!
Run in JShell:	`Main.play(20.0, "level.txt")`
Tests:	`All tests should pass`