Photo by Marvin Meyer on Unsplash

Lab 7

Reductions • Mappings • Filterings • Animated Pacman with IO.animate • Animated Pacman with IO.interact • Interactive Pacman Application with IO.interact

Copy Your Lab 5 Toolbelt

In Lab 5 you added methods to your `Toolbelt` class. The `Toolbelt` class included in the Lab 7 starter repository is missing these methods. Please **copy the methods of your Lab 5 `Toolbelt`** class into the Lab 7 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future).

A. Reductions

There are some very common reductions. Let's implement methods with good names for those, so we can just call those methods instead of having to call `reduce` and think about the **combining function** and **neutral element** each time.

Task A1

<table>
<tr><td>Class:</td><td><code>Toolbelt</code></td></tr>
<tr><td>Task:</td><td>

First, implement a `map`, `filter`, and `reduce` method in your toolbelt.

```
public static <A,B> Sequence<B> map(
  Function1<A,B> mapper,
  Sequence<A> sequence
)
public static <E> Sequence<E> filter(
  Function1<E,Boolean> predicate,
  Sequence<E> sequence
)
public static <R,E> R reduce(
  R neutralElement,
  Function2<E,R,R> combiner,
  Sequence<E> sequence
)
```

</td></tr>
</table>

Task A2

<table>
<tr><td>Class:</td><td><code>Reductions</code></td></tr>
<tr><td>Task:</td><td>

Given the following existing methods in the `Reduction` class:

```
public static double sumAB(double a, double b)
public static double minAB(double a, double b)
public static double maxAB(double a, double b)
public static <T> int incB(T a, int b) { return b + 1; }
```

Using **method references** to the above methods, and `Toolbelt.reduce`, implement the following methods:

```
public static double sum(Sequence<Double> values)
public static double min(Sequence<Double> values)
public static double max(Sequence<Double> values)
public static <T> int length(Sequence<T> values)
```

Think hard about how `length`'s combining function differs from the other three combining functions. What does it do with the element? What does it need to know about the element? What does it need to know about the type of the element?

Hint: if you run into some error regarding type incompatibility between `Integer` and `Double`, make sure that you are using the literals of the

</td></tr>
</table>

	appropriate type (`double` literals have a decimal part, while `int` literals don't) **Note**: use `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` as the return value for when an empty sequence is given as the argument of the `min` and `max` methods respectively.
Run in JShell:	`Reductions.sum(of(1.0, 2.0, 3.0))`
Result:	`==> 6.0`
Run in JShell:	`Reductions.sum(empty())`
Result:	`==> 0.0`
Run in JShell:	`Reductions.min(of(1.0, 2.0, 3.0))`
Result:	`==> 1.0`
Run in JShell:	`Reductions.min(empty())`
Result:	`==> Infinity`
Run in JShell:	`Reductions.max(of(1.0, 2.0, 3.0))`
Result:	`==> 3.0`
Run in JShell:	`Reductions.max(empty())`
Result:	`==> -Infinity`
Run in JShell:	`Reductions.length(of(1.0, 2.0, 3.0))`
Result:	`==> 3`
Run in JShell:	`Reductions.length(empty())`
Result:	`==> 0`

Task A3

Class:	`Reductions`

Task:	Given the following existing methods in the `Reduction` class:
	`public static boolean andAB(boolean a, boolean b)`
	`public static boolean orAB(boolean a, boolean b)`
	`public static String joinBA(String a, String b)`
	Using **method references** to the above methods, and `Toolbelt.reduce`, implement the following methods:
	`public static boolean and(Sequence<Boolean> values)`
	`public static boolean or(Sequence<Boolean> values)`
	`public static String join(Sequence<String> values)`
	Note: always return `true` and `false` when the sequences are empty in the implementation of the methods `and` and `or` respectively. For the `join` method, return an empty string instead.

Run in JShell:	`Reductions.and(of(true, false, true))`
Result:	`==> false`
Run in JShell:	`Reductions.and(empty())`
Result:	`==> true`
Run in JShell:	`Reductions.or(of(true, false, true))`
Result:	`==> true`
Run in JShell:	`Reductions.or(empty())`
Result:	`==> false`
Run in JShell:	`Reductions.join(of("He", "ll", "o"))`
Result:	`==> "Hello"`
Run in JShell:	`Reductions.join(empty())`
Result:	`==> ""`

Task A4

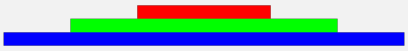
Class:	`ReductionsPropertiesTest`
Tests:	Test your code by running the tests provided in the `ReductionsPropertiesTest` class.

Task A5

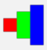
Class:	`Reductions`

Task:	Given the following existing methods in class `Graphics`:
	`public static Graphic above(Graphic a, Graphic b)`
	`public static Graphic beside(Graphic a, Graphic b)`
	Using **method references** to the above methods, and `Toolbelt.reduce`, implement the following methods:
	`public static Graphic aboves(Sequence<Graphic> values)`
	`public static Graphic besides(Sequence<Graphic> values)`

Run in JShell:	`show(`
	`  Reductions.aboves(of(`
	`    rectangle(100, 10, RED),`
	`    rectangle(200, 10, GREEN),`
	`    rectangle(300, 10, BLUE)`
	`  ))`
	`);`

Output:	
Run in JShell:	`show(Reductions.aboves(empty()));`
Output:	(empty graphic)

Run in JShell:	`show(`
	`  Reductions.besides(of(`
	`    rectangle(10, 10, RED),`
	`    rectangle(10, 20, GREEN),`
	`    rectangle(10, 30, BLUE)`
	`  ))`
	`);`

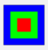
Output:	
Run in JShell:	`show(Reductions.besides(empty()));`
Output:	(empty graphic)

Task A6

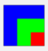
Class:	`Reductions`

Task:	Given the following existing methods in class `Graphics`:
	
	`public static Graphic overlay(Graphic a, Graphic b)`
	`public static Graphic compose(Graphic a, Graphic b)`
	
	Using **method references** to the above methods, and `Toolbelt.reduce`, implement the following methods:
	
	`public static Graphic overlays(Sequence<Graphic> values)`
	`public static Graphic composes(Sequence<Graphic> values)`

Run in JShell:	`show(`
	`  Reductions.overlays(of(`
	`    rectangle(10, 10, RED),`
	`    rectangle(20, 20, GREEN),`
	`    rectangle(30, 30, BLUE)`
	`  ))`
	`);`

Output:	
Run in JShell:	`show(Reductions.overlays(empty()));`
Output:	(empty graphic)

Run in JShell:	`show(`
	`  Reductions.composes(of(`
	`    pin(BOTTOM_RIGHT, rectangle(10, 10, RED)),`
	`    pin(BOTTOM_RIGHT, rectangle(20, 20, GREEN)),`
	`    pin(BOTTOM_RIGHT, rectangle(30, 30, BLUE))`
	`  ))`
	`);`

Output:	
Run in JShell:	`show(Reductions.composes(empty()));`
Output:	(empty graphic)

B. Mappings

Task B1

Class:	`Mappings`

Task:	Given the following existing methods (that convert between types) in your `Mappings` class:
	
	`public static int d2i(double d)`
	`public static int s2i(String s)`
	`public static double i2d(int i)`
	`public static double s2d(String s)`
	`public static String i2s(int i)`
	`public static String d2s(double d)`
	
	Using **method references** to the above methods, and `Toolbelt.map`, implement the following methods:
	
	`public static Sequence<Integer> d2iSeq(Sequence<Double> vals)`
	`public static Sequence<Integer> s2iSeq(Sequence<String> vals)`
	
	`public static Sequence<Double> i2dSeq(Sequence<Integer> vals)`
	`public static Sequence<Double> s2dSeq(Sequence<String> vals)`
	
	`public static Sequence<String> i2sSeq(Sequence<Integer> vals)`
	`public static Sequence<String> d2sSeq(Sequence<Double> vals)`

Run in JShell:	`Reductions.join(`
	`  intersperse(`
	`    " + ",`
	`    concat(`
	`      Mappings.d2sSeq(of(0.1, 3.14, 0.2)),`
	`      Mappings.i2sSeq(of(1, 3, 2))`
	`    )`
	`  )`
	`)`

Result:	`==> "0.1 + 3.14 + 0.2 + 1 + 3 + 2"`

Run in JShell:	`Reductions.sum(`
	`  concat(`
	`    Mappings.i2dSeq(of(1, 3, 2)),`
	`    of(0.1, 3.14, 0.2)`
	`  )`
	`)`

Result:	Determine for yourself what this should produce

Run in JShell:	`Reductions.sum(`
	`  concat(`
	`    of(0.1, 3.14, 0.2),`
	`    Mappings.i2dSeq(of(1, 3, 2))`
	`  )`
	`)`

Result:	Determine yourself what this should produce

Run in JShell:	`Reductions.sum(`
	`  concat(`
	`    Mappings.s2dSeq(of("0.1", "3.14", "0.2")),`
	`    Mappings.i2dSeq(Mappings.s2iSeq(of("1", "3", "2")))`
	`  )`
	`)`

Result:	Determine yourself what this should produce

Task B2

Class:	`Mappings`

Task:	Given the provided record class `Person`.
	Hint (not covered in workbook): When using method references for ***instance*** methods, a nullary method (e.g., a getter) will have type `Function1<C,R>`, where C is the type of the class, and R is the return type of the method.
	Using **method references** to the `firstName`, `age`, and `fullName` instance methods of `Person`, and `Toolbelt.map`, implement the following methods:
	`public static Sequence<String> firstNames(Sequence<Person> ps)`
	`public static Sequence<String> fullNames(Sequence<Person> ps)`
	`public static Sequence<Integer> ages(Sequence<Person> ps)`

Run in JShell:	`println(`
	`  Mappings.firstNames(of(`
	`    new Person("John", "Java", 24),`
	`    new Person("Sarah", "Scala", 25),`
	`    new Person("Olin", "OCaml", 23)`
	`  ))`
	`)`

Output:	`John`
	`Sarah`
	`Olin`

Run in JShell:	`println(`
	`  Mappings.ages(of(`
	`    new Person("John", "Java", 24),`
	`    new Person("Sarah", "Scala", 25),`
	`    new Person("Olin", "OCaml", 23)`
	`  ))`
	`)`

Output:	`24`
	`25`
	`23`

Run in JShell:	`println(`
	`  Mappings.fullNames(of(`
	`    new Person("John", "Java", 24),`
	`    new Person("Sarah", "Scala", 25),`
	`    new Person("Olin", "OCaml", 23)`
	`  ))`
	`)`

Result:	`John Java`
	`Sarah Scala`
	`Olin OCaml`

Task B3

Class:	`MappingsPropertiesTest`
Tests:	Test your code by running the tests provided in the `MappingsPropertiesTest` class.

C. Filterings

Task C1

Class:	`Person, Filterings`

Task:	In the provided record class `Person`, implement the following instance methods:
	
	`// is person older than 70?`
	`public` `boolean isOldie()`
	
	`// does person have first name "Jim"`
	`public` `boolean isJim()`
	
	Using **method references** to the `isOldie` and `isJim` instance methods of `Person`, and `Toolbelt.filter`, implement the following methods:
	
	`// persons older than 70`
	`public static` `Sequence<Person> oldies(Sequence<Person> ps)`
	
	`// persons with first name "Jim"`
	`public static` `Sequence<Person> jims(Sequence<Person> ps)`

Run in JShell:	`println(`
	`  Filterings.oldies(of(`
	`    new Person("Jim", "Halpert", 31),`
	`    new Person("James Morgan", "McGill", 48),`
	`    new Person("Marquis", "Warren", 74)`
	`  ))`
	`);`

Result:	`Person[firstName=Marquis, lastName=Warren, age=74]`

Run in JShell:	`println(`
	`  Filterings.jims(of(`
	`    new Person("Jim", "Halpert", 31),`
	`    new Person("James Morgan", "McGill", 48),`
	`    new Person("Marquis", "Warren", 74)`
	`  ))`
	`);`

Result:	`Person[firstName=Jim, lastName=Halpert, age=31]`

Task C2

Class:	`PersonTest`
Task:	Write unit tests for the methods of the `Person` record class in the appropriate file (`src/tests/java/lab/PersonTest.java`).
Tests:	The written tests should pass

Task C3

Class:	`Filterings`

Task:	**Challenge Question!**
	If you can't find a solution to this one, skip it and try again next week.
	Implement the following method using `Toolbelt.filter`:
	`// persons with exact given age`
	`public static Sequence<Person> aged(int age, Sequence<Person> ps)`
	Hint: You can add another class if that helps, but you are **only** allowed to use `filter` to process the list (**no loop or recursion**).

Run in JShell:	`println(`
	`  Filterings.aged(48, of(`
	`    new Person("Jim", "Halpert", 31),`
	`    new Person("James Morgan", "McGill", 48),`
	`    new Person("Marquis", "Warren", 74)`
	`  ))`
	`);`

Result:	`Person[firstName=James Morgan, lastName=McGill, age=48]`

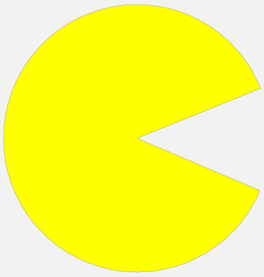
Run in JShell:	`println(`
	`  Filterings.aged(30, of(`
	`    new Person("Jim", "Halpert", 31),`
	`    new Person("James Morgan", "McGill", 48),`
	`    new Person("Marquis", "Warren", 74)`
	`  ))`
	`);`

Result:	`(nothing / empty list)`
Tests:	Test your code by running the tests provided in the `FilteringPropertiesTest` class.

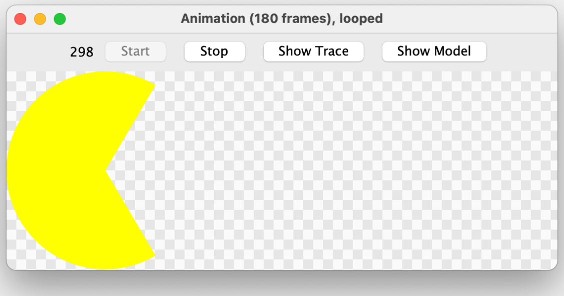
D. Animated Pacman with IO.animate

First, you just (re)implement an animated pacman using `IO.animate`.

Task D1

Class:	`AnimatedPacman`
Task:	Implement the method `pacman`: `public static Graphic pacman(int mouthAngle)` It should produce a pacman with a partially open mouth (0 to 180 degrees), like what you already implemented in Lab 1 and produced a film strip of and animated in Lab 3. Pacman's height should be 100.
Run in JShell:	`show(AnimatedPacman.pacman(45))`
Output:	

Task D2

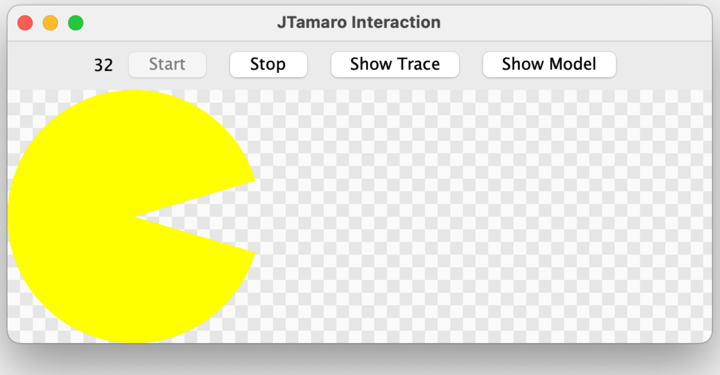
Class:	`AnimatedPacman`
Task:	Implement the method `animation`: **`public static`** `void animation()` This method is special. It has `void` as a return type. It does not actually return any value. It should just call `IO.animate` with three arguments: - a sequence of pacman Graphics with gradually more open mouths (use `range` and `map` to produce them) - the value `true` (to loop the animation) - the value 25 (for 25 milliseconds between animation frames) **Note**: the mouth angle should change from 0 to 180 and then “jump back” to 0. **Note**: this is a method of type `void`. Such methods do not return any value, which means that you don’t have to write a return statement.
Run in JShell:	`AnimatedPacman.animation();`
Output:	This should open a window showing the pacman opening and closing its mouth.  Animation (180 frames), looped 298 Start Stop Show Trace Show Model

E. Animated Pacman with IO.interact

Now you will reproduce the same animation using a more powerful API. In fact, `IO.animate` internally uses this API. It allows to animate, but also to interact.

Task E1

Class:	`AnimatedPacman`
Task:	Implement the method `interaction`: **`public static`** `void interaction()` This method should just call `IO.interact` with one argument: the initial mouth angle (of type `int` or `Integer`). Method `IO.interact` returns an object of type `Interaction`. `Interaction` is a class that implements what's known in Software Engineering as a "**fluent API**" or "fluent interface". It has several instance methods you can call on `Interaction` objects. You can call them in a "chain" of method calls, like this: `IO.interact(0).withXxx(…).withYyy(…).withZzz(…)…` Each `with…` method returns an `Interaction` object, so you can call another method on the object. You end the chain of methods by calling method `run()`. That run call causes the interaction, which you configured with all the `with…` calls, to execute. Use the following `with…` calls to configure the interaction: - `withRenderer(…)` – pass a function object that will be called each time the animation should be rendered. The function object should have type `Function1<Integer,Graphic>`. You could e.g., pass `AnimatedPacman::pacman`. - `withTickHandler(…)` – pass a function object that will be called at each timer tick. This object should have type `Function1<Integer, Integer>`. It should return a new mouth angle given the passed mouth angle. It could e.g., simply return an angle that's, say, 10 degrees larger than the parameter value, and that is 0 if the parameter value is larger than 180 degrees. - `withMsBetweenTicks(…)` – pass how many milliseconds to wait between each tick of the animation Configure your `Interaction` object so it behaves like `IO.animate` did in Exercise D Task 2.
Run in JShell:	`AnimatedPacman.interaction();`
Output:	This should open a window with the title "JTamaro Interaction" showing the pacman opening and closing its mouth.

	JTamaro Interaction 32 Start Stop Show Trace Show Model

F. Interactive Pacman Application with IO.interact

Now we will develop a simple but complete *interactive* application. We want the arrow keys on our keyboard to turn the Pacman into the corresponding direction, and we want the mouth of the Pacman to open and close with each timer tick.
We will re-use this code in a future lab where we'll implement a playable Pacman game.

When developing any interactive application, it is best practice to split the program into two separate parts: the **model** and the **user interface**.

Task F1

Class:	`PacmanState`

Task:	It is a good strategy to first develop the **model**. What exactly is the information that can change during the interaction?
	- How much the mouth is open (modeled as an angle, `mouthAngle`)
	- The direction towards which the pacman is facing (also modeled as an angle, `rotation`)
	Let's create a **record class** `PacmanState` that implements this model.
	Add two components to the record class (one named `mouthAngle`, of type `int`, another named `rotation`, of type `int`).
	Then add two instance methods, which allow us to create a new `PacmanState` with a new mouth angle, or with a new rotation:
	`public PacmanState withMouthAngle(int newMouthAngle)`
	`public PacmanState withRotation(int newRotation)`

Run in JShell:	`new PacmanState(10, 0)`
Output:	`==> PacmanState[mouthAngle=10, rotation=0]`
Run in JShell:	`new PacmanState(10, 0).withMouthAngle(50)`
Output:	`==> PacmanState[mouthAngle=50, rotation=0]`
Run in JShell:	`new PacmanState(10, 0).withRotation(90)`
Output:	`==> PacmanState[mouthAngle=10, rotation=90]`

Task F2

Class:	`PacmanStateTest`
Task:	Write unit tests for the methods of the `PacmanState` record class in the appropriate file (`src/tests/java/lab/PacmanState.java`).
Tests:	The written tests should pass

Now that we have a working model of our interaction, we need to “connect” the model with the user interface (UI). This consists of two “connections”:

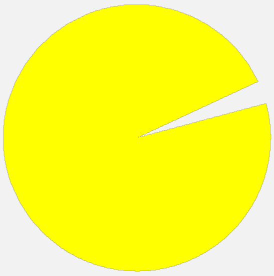
- **View**: Model -> UI
  Map from model information to UI output (in this case, JTamaro graphics)
- **Controller**: UI -> Model
  Map from UI input (mouse, keyboard, timers) to the model

This kind of architecture is common for interactive applications. It's known as the "model-view-controller" pattern. You will find it in some form or another in pretty much every framework for GUI, mobile, or web applications.

The **view** defines how to represent the information from the model in the user interface. For us, the view is implemented as a method that renders the model into a JTamaro Graphic.

The **controller** is how to deal with user (and other) external events. The controller usually updates the model. The controller parts are often also known as "event handlers", "observers", "callbacks", or "listeners". For us, the controllers are implemented as methods that produce a new model.

Task F3

Class:	`InteractivePacman`
Task:	Let's do a clean design and separate the different aspects into different methods. First, let's implement a method for rendering the view: `public static Graphic render(PacmanState state)` This method should produce a pacman that is rotated by the given rotation angle and that has the mouth open as much as specified by the mouth angle.
Run in JShell:	`show(InteractivePacman.render(new PacmanState(10, 20)));`
Output:	

Task F4

Class:	`InteractivePacman`
Task:	Now, let's implement a method that handles timer ticks by opening the mouth a bit (by 10 degrees), or closing it completely (when we exceed the maximum mouth angle of 180 degrees): `public static PacmanState onTick(PacmanState before)` In your implementation, call `PacmanState.withMouthAngle`.
Run in JShell:	`InteractivePacman.onTick(new PacmanState(10, 20))`
Output:	`==> PacmanState[mouthAngle=20, rotation=20]`
Run in JShell:	`InteractivePacman.onTick(new PacmanState(180, 20))`
Output:	`==> PacmanState[mouthAngle=0, rotation=20]`
Tests	Test your implementation by running the tests provided in the `InteractivePacmanPropertiesTest` class.

Task F5

Class:	`InteractivePacman`
Task:	Now, let's implement a method that handles keyboard key releases by the user by rotating the pacman in the direction of the arrow key: **`public static`** `PacmanState onKeyRelease(` `  PacmanState before, KeyboardKey key)` In your implementation, call `PacmanState.withRotation`. To determine whether an arrow key was (pressed and then) released, and which key it was, compare `key.getCode()` with one of the constants using the `==` operator: • `KeyboardKey.RIGHT` • `KeyboardKey.UP` • `KeyboardKey.LEFT` • `KeyboardKey.DOWN` **Note**: do not change the rotation unless the released key is one of those listed above.

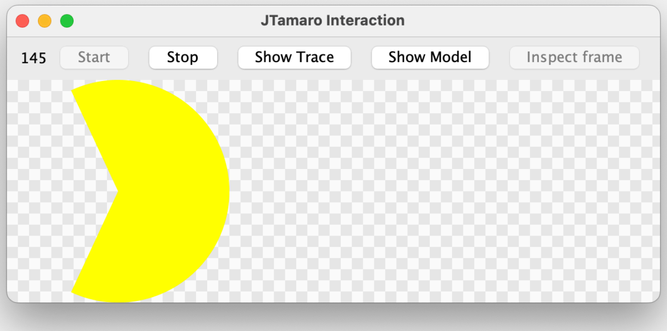
Run in JShell:	`InteractivePacman.onKeyRelease(`
	`  new PacmanState(10, 20),`
	`  new KeyboardKey(KeyboardKey.UP)`
	`).rotation()`

Output:	`==> 90`

Run in JShell:	`InteractivePacman.onKeyRelease(`
	`  new PacmanState(10, 20),`
	`  new KeyboardKey(KeyboardKey.DOWN)`
	`).rotation()`

Output:	`==> 270`
Tests	Test your implementation by running the tests provided in the InteractivePacmanPropertiesTest class.

Task F6

Class:	`InteractivePacman`
Task:	Now we have a model, a view, and controllers. We just have to wire all of them up. To do this, implement the interaction method: **`public static`** `void interaction()` In your implementation, call `IO.interact(new PacmanState(0, 0))` On the returned `Interaction` object call: - `withRenderer` - `withTickHandler` - `withKeyReleaseHandler` - `run` Pass appropriate arguments to each `with…` method (most arguments should be method references of the view and controller methods you just implemented). **Recall**: this is a method of type `void`. Such methods do not return any value, which means that you don't have to write a return statement.
Run in JShell:	`InteractivePacman.interaction();`
Output:	This should open up a window with the title "JTamaro Interaction" showing the pacman opening and closing its mouth. You should be able to control the direction the pacman is facing using the four arrow keys on your keyboard.  JTamaro Interaction 145 Start Stop Show Trace Show Model Inspect frame