Photo by Marvin Meyer on Unsplash

Lab 5

Fibonacci · TimeStamp · TimeInterval

Copy Your Lab 4 Toolbelt

In Lab 4 you added methods to your `Toolbelt` class. The `Toolbelt` class included in the Lab 5 starter repository is missing these methods. Please **copy the methods of your Lab 4 `Toolbelt`** class into the Lab 5 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future).

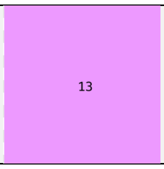
A. Fibonacci

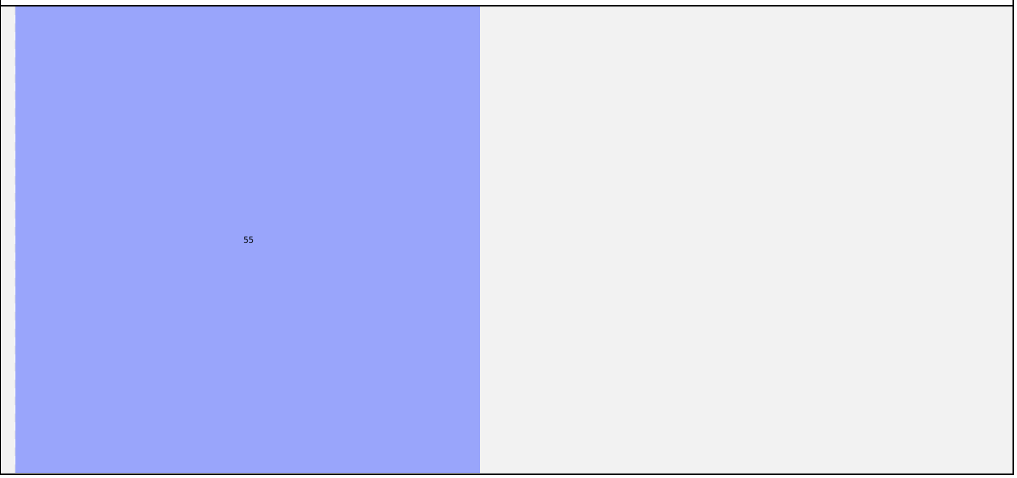
We will gradually build up a visualization of the Fibonacci sequence as shown in this TED talk by Arthur Benjamin (watching it is not required to complete the lab, but it might be instructive anyway – it's a great one!).

Task A1

Class:	`Fibonacci`
Task:	The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones (starting from 0): $$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$ $$fib(n) = \begin{cases} 0\ if\ n = 0 \\ 1\ if\ n = 1 \\ fib(n-1) + fib(n-2) \end{cases}$$ Implement using recursion the `fib` static method which given an integer n, computes the nth number of the Fibonacci sequence.
Run in JShell:	`Fibonacci.fib(5)`
Output:	`==> 5`
Run in JShell:	`Fibonacci.fib(6)`
Output:	`==> 8`
Run in JShell:	`Fibonacci.fib(7)`
Output:	`==> 13`

Task A2

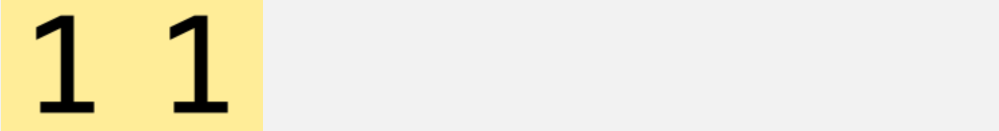
Class:	`Fibonacci`
Task:	Implement the static method `tile`, which takes one parameter `fibN` of type `int` and produces a square proportional to the side length (use $fibN \times 10$), color determined by `tileColor(fibN)`, and has the textual representation of the number `n` overlaid on top of it. This text is of color `BLACK`, has size `10` and uses the `MONOSPACED` font. **Note**: to convert an `int` to a `String`, use the `String.valueOf(…)` method.
Run in JShell:	`show(Fibonacci.tile(Fibonacci.fib(7)))`
Output:	

	`show(Fibonacci.tile(Fibonacci.fib(10)))`
	

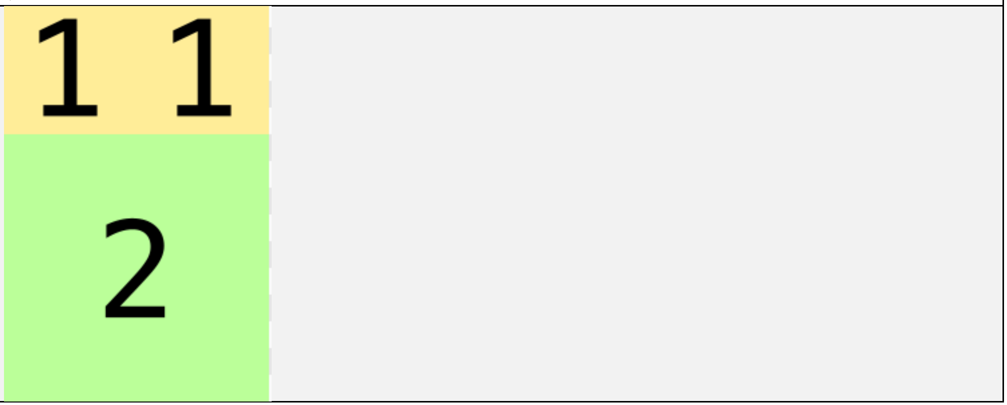
Task A3

Class:	`Fibonacci`
Task:	Implement the `justxapose` static method, which takes two `Graphic` instances and a `boolean`. The value of the `boolean` determines whether the two graphics should be juxtaposed horizontally (`true`) or vertically (`false`).

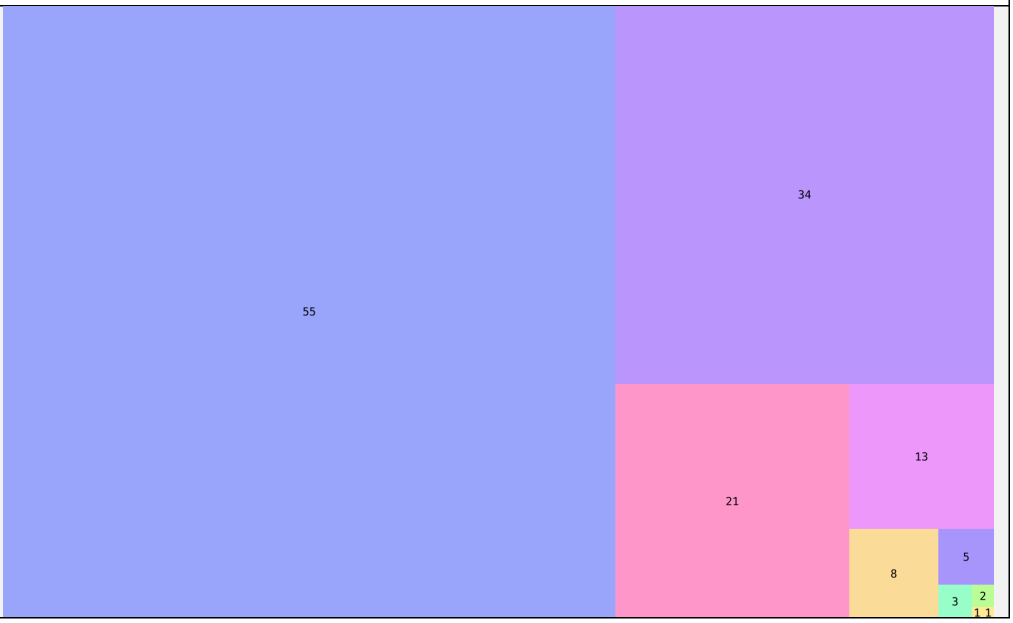
Run in JShell:	`show(Fibonacci.juxtapose(Fibonacci.tile(Fibonacci.fib(1)),`
	`                         Fibonacci.tile(Fibonacci.fib(2)),`
	`                         true))`

Output:	

	`show(Fibonacci.juxtapose(`
	`         Fibonacci.juxtapose(`
	`             Fibonacci.tile(Fibonacci.fib(1)),`
	`             Fibonacci.tile(Fibonacci.fib(2)),`
	`             true),`
	`         Fibonacci.tile(Fibonacci.fib(3)),`
	`         false)`
	`)`

Task A4

Class:	`Fibonacci`
Task:	Implement using recursion the `fibonacciRectangle` static method, which takes an `int` parameter `n` and produces the rectangle constructed by juxtaposing `n` tiles, each having the side of the `n`th Fibonacci number. **Note**: there is no tile corresponding to the 0th Fibonacci number. **Note**: each time the tile is juxtaposed in the opposite direction compared to the previous juxtaposition: the first two tiles (1 and 1) are juxtaposed horizontally, the third tile (2) is juxtaposed vertically, the fourth tile (3) is juxtaposed horizontally again and so on.
Run in JShell:	`show(Fibonacci.fibonacciRectangle(10))`
Output:	

B. Modeling a Point in Time

(Throughout these exercises, you will encounter some questions that help you to understand the problem. Answer them in the dedicated comments directly within the Java source code *before* implementing the methods.)

Task B1

Let's develop a class named `TimeStamp` that can be used to represent a point in time. Let's assume that the time in the modelled system is always increasing.

Before hacking any code, let's see how we would use such a `TimeStamp` class.

```
new TimeStamp(4) // Create a TimeStamp
new TimeStamp(2) // Create another TimeStamp
```

The above code would create two `TimeStamp` objects, one representing time 4, the other representing time 2. Let's assume that smaller numbers represent earlier points in time.

Class:	`TimeStamp`
Task:	Implement the `TimeStamp` record class. It should have only 1 component of type `int`, named `time`.
Run in JShell:	`new TimeStamp(1)`
Output:	`==> TimeStamp[time=1]`
Run in JShell:	`new TimeStamp(2).time()`
Output:	`==> 2`

Task B2

Now we may want to compare two `TimeStamp` instances.
Assume that `start` is `new TimeStamp(2)` and `end` is `new TimeStamp(4)`:

```
end.equalTo(start) // Check whether start and end represent the same point in time
end.after(start)   // Check whether end comes after start
end.before(start)  // Check whether end comes before start
```

- What value will the above tree expressions produce?
- Do we really need a `before` and an `after` method? Do we want both?

Now, consider we also have another `TimeStamp` instance, `t2`, standing for `new TimeStamp(2)`. Given this expression:

```
t2.equalTo(start)
```

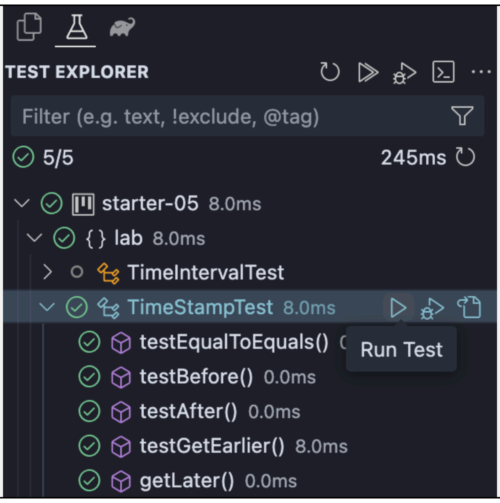
- What value should the above expression have?
- Are `t2` and `start` two separate objects?

Finally, we want to easily get the *later* or *earlier* of two `TimeStamp`s as follows:

```
start.getEarlier(end)
start.getLater(end)
```

- What value would the following expression produce?

```
(end.getEarlier(start)).equalTo(start.getLater(end))
```

Class:	`TimeStamp`
Task:	Implement the `equalTo`, `before`, `after`, `getEarlier`, `getLater` instance methods of the TimeStamp class as described above.
Test:	Make sure all tests of the `TimeStampTest` class successfully pass before proceeding to the next task!
Output:	 TEST EXPLORER Filter (e.g. text, !exclude, @tag) 5/5 245ms starter-05 8.0ms { } lab 8.0ms TimeIntervalTest TimeStampTest 8.0ms testEqualToEquals() (Run Test testBefore() 0.0ms testAfter() 0.0ms testGetEarlier() 8.0ms getLater() 0.0ms

C. Modeling a Time Interval

Task C1

Now we develop a class called `TimeInterval` that represents the interval between two `TimeStamp` instances.

```
new TimeInterval(begin, end)
```

A `TimeInterval` is a half-open interval $[begin, end)$. It excludes the end point. We can visualize a `TimeInterval` as a sequence of letters (e.g., O), one for every `TimeStamp` it includes. Note that the end point is not included.

```
..OOOOOO.........
  │     │
  │     └─ end
  └─────── begin
```

Class:	`TimeInterval`
Task:	Implement the `TimeInterval` record class. It should have two components of type `TimeStamp`, named `begin` and `end`.
Run in JShell:	`new TimeInterval(new TimeStamp(1), new TimeStamp(3))`

Output:	`==> TimeInterval[begin=TimeStamp[time=1],`
	`end=TimeStamp[time=3]]`

Run in JShell:	`new TimeInterval(new TimeStamp(3), new TimeStamp(6)).begin()`
Output:	`==> TimeStamp[time=3]`
Run in JShell:	`new TimeInterval(new TimeStamp(3), new TimeStamp(6)).end()`
Output:	`==> TimeStamp[time=6]`

Task C2

As Allen has shown in his paper Maintaining Knowledge about Temporal Intervals, a time interval supports 13 different predicates.

The following table lists and visualizes them with a little diagram. `TTT` refers to "this" `TimeInterval`, i.e. the one we are invoking the method on, whereas `OOO` refers to the "other" `TimeInterval`, i.e. the one passed into the parameter. For each predicate we want to have an instance method in `TimeInterval` that checks whether two `TimeInterval` instances (e.g., `TTT` and `OOO`) are in that relation.

Allen Sym	Method Name	Diagram	Comment

`=`	`equalTo`	`....TTT....`
		`....OOO....`
			Symmetric

`<`	`before`	`TTT........`
		`....OOO....`
			Inverse of `after`

`>`	`after`	`........TTT`
		`....OOO....`
			Inverse of `before`

`m`	`meetsBeginOf`	`.TTT.......`
		`....OOO....`
			Inverse of `meetsEndOf`

`mi`	`meetsEndOf`	`.......TTT.`
		`....OOO....`
			Inverse of `meetsBeginOf`

`o`	`overlapsBeginOf`	`...TTT.....`
		`....OOO....`
			Inverse of `overlapsEndOf`

`oi`	`overlapsEndOf`	`.....TTT...`
		`....OOO....`
			Inverse of `overlapsBeginOf`

`d`	`during`	`....TTT....`
		`...OOOOO...`
			Inverse of `contains`

`di`	`contains`	`...TTTTT...`
		`....OOO....`
			Inverse of `during`

`s`	`starts`	`...TTT.....`
		`...OOOOO...`
			Inverse of `startedBy`

`si`	`startedBy`	`...TTTTT...`
		`...OOO.....`
			Inverse of `starts`

`f`	`finishes`	`.....TTT...`
		`...OOOOO...`
			Inverse of `finishedBy`

`fi`	`finishedBy`	`...TTTTT...`
		`.....OOO...`
			Inverse of `finishes`

In the above table, e.g., the `before` predicate (Allen's "<") represents whether this interval (`TTT`) happened before the other interval (`OOO`).

Note that any given pair of intervals is exactly in one of the 13 predicates. Thus, for any given pair of `TimeInterval` instances, one and only one of the 13 predicates will be `true`. For example, if `a.before(b)` then it does not `a.meetsBeginOf(b)`, since `a.before(b)` implies that there is a gap between `a` and `b`.

12 of the 13 predicates (all except for `equalTo`) have inverses. E.g., the inverse of `starts` is `startedBy`: `a.starts(b)` is the same as `b.startedBy(a)`.

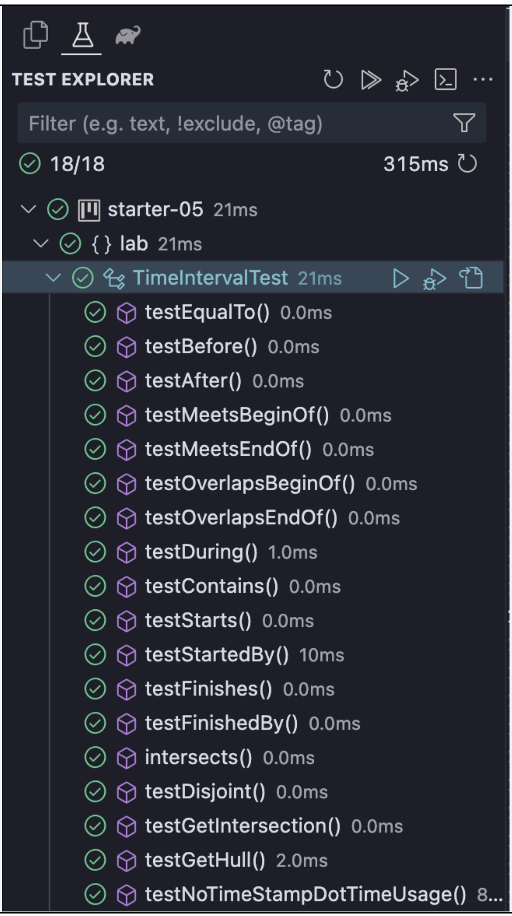
- For the intervals `i13` = $[1, 3)$ and `i47` = $[4, 7)$, which predicate is true (i.e., what is the name of the method `xxx` that returns `true` when called like `i13.xxx(i47)`)?
- And which predicate is its inverse (i.e., what is the name of the method `yyy` that returns true when called like `i47.yyy(i13)`)?

Beside the above 13 predicates, we would also like to provide two derived predicates: `intersects` and `disjoint`.
Two `TimeInterval` instances intersect if they have some point in common.
Two `TimeInterval` instances are disjoint if they have no point in common.

- Try to define `intersects` as a disjunction (a logical formula that connects clauses using "*or*") of some of the above 13 predicates.
- Try to define `disjoint` as a disjunction of some of the above 13 predicates.
- Are these two predicates (`intersect` and `disjoint`) mutually exclusive, that is, a pair of `TimeInterval` instances either intersects or is disjoint?

Finally, we want a way to compute the `intersection` and the `hull` of two `TimeInterval` instances. The intersection corresponds to the largest interval *included* in both intervals. The hull corresponds to the smallest interval *including* both intervals.

Class:	`TimeInterval`
Task:	Implement the 13 predicates and `intersects`, `disjoint`, `intersection` and `hull` as instance methods of the `TimeInterval` record class. **Note**: each predicate, `intersects` and `disjoint` should all take one parameter of type `TimeInterval` and return a `boolean` value. On the other hand, `intersection` and `hull` take one parameter of type `TimeInterval` and return a `TimeInterval` instance. **Note**: take advantage of the information about certain predicates being the inverse of others, this will help you simplify your code. **Note**: if we have to return a `TimeInterval` but there's no meaningful `TimeInterval`, such as when computing `a.intersection(b)` when `a.intersects(b)` is `false`, return a `TimeInterval` from `TimeStamp 0` to `TimeStamp 0` (empty interval). **Note**: you are **NOT** allowed to use `TimeStamp.time()` to implement any method in `TimeInterval` (there is no need!). Instead, use the instance methods implemented in Task B2.

Test:	Make sure all tests of the `TimeIntervalTest` class are passing.
Output:	 TEST EXPLORER Filter (e.g. text, !exclude, @tag) ✓ 18/18 315ms ✓ starter-05 21ms ✓ { } lab 21ms ✓ TimeIntervalTest 21ms ✓ testEqualTo() 0.0ms ✓ testBefore() 0.0ms ✓ testAfter() 0.0ms ✓ testMeetsBeginOf() 0.0ms ✓ testMeetsEndOf() 0.0ms ✓ testOverlapsBeginOf() 0.0ms ✓ testOverlapsEndOf() 0.0ms ✓ testDuring() 1.0ms ✓ testContains() 0.0ms ✓ testStarts() 0.0ms ✓ testStartedBy() 10ms ✓ testFinishes() 0.0ms ✓ testFinishedBy() 0.0ms ✓ intersects() 0.0ms ✓ testDisjoint() 0.0ms ✓ testGetIntersection() 0.0ms ✓ testGetHull() 2.0ms ✓ testNoTimeStampDotTimeUsage() 8...