Programming Assignment #4 CS671

1st submission due 7/14/2021
2nd submission due 7/21/2021

This assignment involves some of the techniques that can be used when designing library code. Library designers cannot anticipate all the ways users might want to user their services, so flexibility must be designed in from the start. This can be hard to do with strongly typed languages, but Scala offers mechanisms that facilitate flexibility in design.

Fireworks

This assignment implements a *domain-specific language* (DSL) for specifying fireworks shows. These shows usually involve colorful pyrotechnic effects in the sky involving light and sound, but they are also often coupled with visual effects on the ground and audio tracks coming over loudspeakers or listener's radios.

The Fireworks DSL allows users to specify when different effects should run without having to worry about when they should be launched. The DSL allows users to combine `Effect` objects of varying types, allowing them to be timed according to when other effects start or finish. The `Effect` combinations are stored in `CompositeEffect` objects. These objects provide a way to simulate the fireworks show by "playing" themselves out in text at regular time intervals. `CompositeEffect` objects also provide mechanisms to analyze the fireworks show in various ways, with flexibility to add other analyses later.

The Fireworks library is split into several portions that you need to fill in to make it all work together.

Value Classes: `Meters`, `Seconds`, and `Decibels`

There are a few different units that are important to the Fireworks library. Each one is represented by a `Double`, and it needs to support the methods and operators specified in the starter code. In addition each one should overload the `toString` method and support an implicit conversion method that takes a Double and returns an instance of the value class. They should be defined in the `fireworks` package object. Details are in Table 1, below.

Value class	Implicit method	String format example (for value 5.231)	Notes
`Meters`	`m`	5.23 m	Notice multiplication takes a `Double` argument
`Seconds`	`s`	[given in starter code]	Ranges require `Int` values
`Decibels`	`db`	5.2 dB	Decibels are calculated for *x* + *y* using the formula $$10 \log_{10}(10^{x/10} + 10^{y/10})$$ https://www.noisemeters.com/apps/db-calculator/ For multiplication by *n*, you add 3*n*

Table 1: Key aspects of the value classes representing units of meters, seconds, and decibels.

The `Effect` trait and `CompositeEffect` class

The key parts at the center of the Fireworks library are the `Effect` trait, which represents one of the smallest units of a fireworks display, and the `CompositeEffect` class, which represents a structured arrangement of `Effect`s.

Every `Effect` has a `duration`, indicating how long the effect is expected to last in real-life. Each `Effect` also knows how to `render` itself to text, and how to combine itself with another effect into a `CompositeEffect`. All you need to add to this trait are the operations that determine how a `CompositeEffect` should be created from the current (left-hand-side) `Effect`, so that it can be combined using the `CompositeEffect` version of the same operation. These operations (valid for both `Effect` and `CompositeEffect`) are given in Table 2, below.

Basic operation	Symbol	Meaning
`followedBy`	`+`	The `Effect` on the right-hand-side of the operation begins **immediately after completion of** the `Effect` that was *most recently added* to the `CompositeEffect` on the left-hand-side. If the right-hand-argument is an integer, it adds a delay of that many seconds after **completion of** the *most recently added* `Effect`.
`togetherWith`	`!`	The `Effect` on the right-hand-side starts **at the same time as** the `Effect` that was *most recently added* to the `CompositeEffect` on the left-hand-side. If the right-hand-argument is an integer, it adds a delay of that many seconds after **the start of** the *most recently added* `Effect`.
`alongside`	`||`	The `Effect` on the right-hand-side starts **at the same time as** the `Effect` on the left-hand-side, effectively meaning the *first* `Effect` in the `CompositeEffect` on the left-hand-side.

Table 2: Operations for combining `Effect` objects

The `CompositeEffect` class both extends `Effect` and maintains a collection of `Effect`s, making it is a recursive data structure. You are responsible for maintaining a mutable map of time (`Seconds`) to `List[Effect]` called `timings`, as well as measurement of the full duration of the `Effects` called `fullDuration`. As each `Effect` is added to the `CompositeEffect`, it must be added to the head of the list in `timings` that is appropriate for its starting time (see Table 2). Also, `fullDuration` must be updated to reflect the span of time from the start of the `CompositeEffect` until the end of the last `Effect` (which may not be the most recently-added effect). It is suggested that you use `lastInsertionTime` and `lastDuration` to help you keep track of what time is the appropriate insertion point for the next `Effect`.

To get a better sense for how these operations should work to compose `CompositeEffect`s, consider the illustration in Figure 1. This figure shows the construction of the `sequence` variable given in the tests. The code is given at the top, followed by an illustration that shows how each of the input `Effect`s would line up on a timeline. Note that the `+` operator chains Effects end-to-end, `!` stacks the next one on top of its predecessor, and `||` stacks things starting right at the beginning again. At the bottom, the figure shows how the `sequence` variable is constructed. Notice that each operation generally just adds to the `CompositeEffect` that is already present on the left-hand-side (constructed when one of the operators is applied to a non-composite `Effect` as a left-hand operand). No nesting occurs until either parentheses or operator precedence cause later `Effect`s to combine before earlier `Effect`s.

```
val sequence = (
  Peony(red, 10.m, 15.m)
    + Ring("clover", green, 5.m)
    + 5 + Peony(blue)
    ! Salute()
  ) || //alongside
  Music("Yankee Doodle Dandy snippet", 15.s, 100.db, "ydd.mp4",
    "I'm a Yankee Doodle dandy\n" +
      "A Yankee Doodle, do or die\n" +
      "A real live nephew of my Uncle Sam\n" +
      "Born on the Fourth of July.") + greenFlare
```

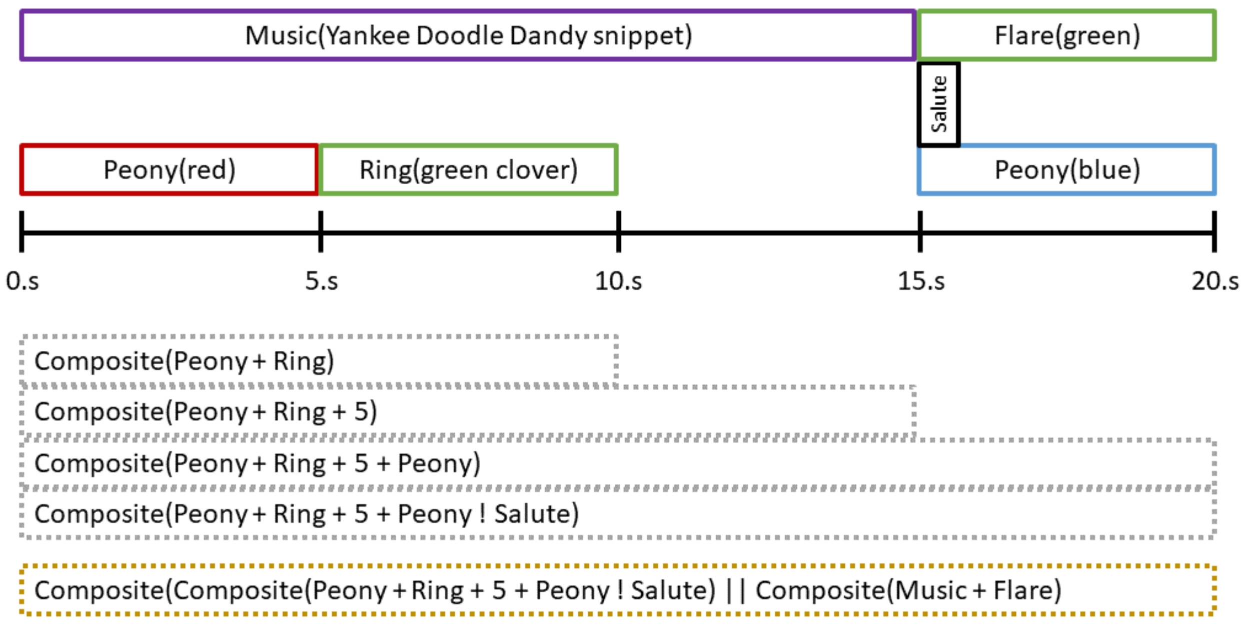
Music(Yankee Doodle Dandy snippet)
Flare(green)
Salute
Peony(red)
Ring(green clover)
Peony(blue)
0.s
5.s
10.s
15.s
20.s
Composite(Peony + Ring)
Composite(Peony + Ring + 5)
Composite(Peony + Ring + 5 + Peony)
Composite(Peony + Ring + 5 + Peony ! Salute)
Composite(Composite(Peony + Ring + 5 + Peony ! Salute) | | Composite(Music + Flare)

Figure 1: The code that defines `sequence`, followed by the effects in `sequence` plotted against time, with each `Effect`'s `duration` suggested by box width. The lower portion shows how `sequence` is constructed with each successive operator, with each `CompositeEffect`'s `fullDuration` suggested by box width.

In addition to these operators, you are responsible for implementing the `render(timeOffset, step)` method for `CompositeEffect`. This method should return the result of concatenating the rendering of all `Effect`s in `timings` that start or continue through the period defined by [`timeOffset`, `timeOffset` + `step`), meaning the period that starts at `timeOffset`, and continues all the way up to (but not including) `timeOffset` + `step`. If there is nothing that fits the time period, an empty string should be returned.

A few more methods are already implemented for you:

- `evaluateAt(step, startTime, endTime)(format)(f)`—this method prints a time-series rendering of the function `f`, formatted by the function `format`. It does this by looping through the range of time [`startTime`, `endTime`] (including endTime), at an interval of `step` and it calls the passed function `f` for each time-step (with the current time and `step` as arguments). This method is one reason your `Seconds` value class needs to support to and `until`.

- `timedHistory(t, event)`—this method formats a time and event-rendering to be printed. It is already given as an example of how the formatting should work.
- `play(step, startTime, endTime)`—this method shows how to call `evaluateAt` with methods appropriate for rendering the `CompositeEffect` using `timeHistory` to format it.
- `generateTimings(offsetFn)`—this method calls the `generateTimingsHelper` method that you are responsible for implementing with an empty mutable map and zero offset. It transforms the output into a sorted immutable map, representing times mapped to a list of effects.

You are responsible for writing the `generateTimingsHelper(result, offset, offsetFn)` method. This method is intended to be a recursive method that fills the passed `result` map by adding each effect to the list corresponding to the appropriate time in the map. This time is determined by offsetting the time of each effect in timings by the result of calling `offsetFn` on that effect. This may move effects to different times in `result` than where they started in `timings`.

You will also be responsible for writing the `traverse` method. This will be explained more in the section about `EffectVisitor`s.

High-level `Effect` traits

There are a few basic traits that extend `Effect` that you need to be familiar with. The `VisualEffect` trait represents anything with a visual component in a real-world fireworks display. It specifies things such as the central height of the effect, the horizonal and vertical spread, colors, and other aspects. The properties `shape` and `baseVisual` are used to help render the `VisualEffect` to text, and they represent the overall shape of the effect and what individual components of the effect look like. The public render method is already overridden for you to properly render `Effects` of with this trait, but you are responsible for writing the support method:

- `render(t: Double)`—If `t` is zero, render as a flash of color in angle brackets, followed by whatever the underlying `Effect` would render. For example, if `color` were `green`, the output would start with `<green flash>`. Otherwise if `t` is as high as 1.0, render as the color, shape, arrangement, and size, as illustrated in the tests named "render of VisualEffect @t=…". If an effect returns `isSingular` as `true`, then the shape and size are not rendered. Nothing should be rendered if `t` is greater than 1.

The `SoundEffect` trait represents anything with an audio component, including explosions, music, and other sound effects. It specifies the (maximum) volume of the effect, and has a sense of whether or not the sound is instantaneous or continual. The properties `baseSound` and `continueSound` are used to help render the `SoundEffect` to text. You are responsible for writing the supporting `render` method:

- `render(t: Double)`—If `t` is zero, render as the `baseSound` in inverted angle brackets with the value of `volume` in square brackets, followed by whatever the underlying `Effect` would render. Otherwise if `t` is as high as 1.0, render as the `continueSound.` These are illustrated in the tests named "render of SoundEffect @t=…". Nothing should be rendered if `t` is greater than 1.

The `SkyEffect` trait represents any effect that must be launched into the air. It specifies information important for staging the display and for determining launch times, including the tube size needed to hold the effect casing, whether it needs a mortar to be launched (or has its own rocket mechanism), and what its weight is. Many other effects are mixed with the SkyEffect trait, but there currently is no functionality built around it.

Pre-populated `Effects`

There are a number of derived traits and case classes already written for you. Here's a brief description of each.

- Case class `EmptyEffect`—represents a "null" `Effect` that has zero duration and has no rendering. This class is not strictly necessary, but some may find it useful during development.
- Case class `Music`—represents an audio clip with lyrics. If the render method of SoundEffect is written properly, it should render as illustrated in the tests named "rend of Music @t=…".
- Trait `SkyVisualEffect`—this is just a convenience trait that combines `SkyEffects` with `VisualEffects`.
- Trait `PopEffect`—this is a convenience trait that defines `volume` at 100 dB.
- Case class Flare—represents a colored constant bright light (star).
- Case class Peony—represents a colored sphere of expanding stars.
- Case class Ring—represents a colored ring of expanding stars that take on the shape specified in the first argument.
- Case class Salute—represents a loud explosion with no visual effect specified.

The `CompositeEffect.traverse` method and `EffectVisitor` traits/objects

As the Fireworks library grows, there are likely to be more effect types that have their own data that users would care about (flying drones?), and there might be more functionality that users are going to want. It might be important to get a profile of the sound levels to make sure no point in the show is too loud, or it might be good to get a list of how many of each type of effect is involved in a show to generate a purchasing list. While the library can't necessarily anticipate all these needs, it can be made very flexible so that additional functionality is easy to add. This is what the `traverse` method of `CompositeEffect` is for, and it uses a special object that extends `EffectVisitor`.

- traverse[A, E](timeOffset, step)(visitor)—Walks through the `CompositeEffect`'s `timing` structure in a way similar to the `render` method, but using `visitor` to collect and assemble information. For each `Effect` in `timings` that overlaps with the window [`timeOffset`, `timeOffset` + `step`) that is of type `E`, this method must call the `visitor.apply` method on that `Effect` to get the desired result (of type `A`) back. Once all the resulting data is collected, it must be processed using `foldLeft`, with `visitor.identity` and `visitor.combine`. A tricky part is making sure to only process `Effects` that are of type `E`. This is done by using `collect` early in the process with the `visitor.select` method—doing this not only eliminates the `Effects` of inappropriate types, but yields a collection that has only effects of type `E` in it. Without this transformation, it would be impossible to call `visitor.apply` because the input effects might not be of the right type.

Here is a brief description of the purpose of each of the `EffectVisitor[A, E]` methods:

- `select`—A partial function that is only defined when an `Effect` of the correct subtype `E` is passed to it (and otherwise is undefined). This is what makes a call to `collect` work properly.
- `apply(effect, t, step)`—This function is meant to return a value of type `A` given the effect of type `E`, a starting time `t` and a time `step`.
- `identity`—This function returns the identity value for the operation defined in `combine`. It is useful as a seed value for calls to `foldLeft`.
- `combine(left, right)`—This function combines two value of type `A` and returns another `A`. It is useful as the operation for calls to `foldLeft`.

Given how all of this is supposed to work, you are responsible for implementing the following:

- `StringAdder`—A trait that defines `identity` and `combine` appropriately to concatenate two strings together, such that it could be used to combine the results of two calls to `Effect.render`, for example. This trait could be mixed with `EffectVisitor` so no code has to be copied for these two methods, as is illustrated in the object `RenderVisitor`.
- `DecibelAdder`—A trait that defines `identity` and `combine` appropriately to add two `Decibel` values together, such that it could be used to find volume levels at a given time.
- `VolumeVisitor.apply(effect, t, step)`—This method should return the `volume` of the passed `effect` for the given time frame (note that the effect is actually of type `SoundEffect`). Be sure to return a value of `Double.NegativeInfinity` if `effect` does not have a continual sound and `t` > 0.
- `VolumeVisitor.select`—This method should return the the passed `Effect` as a `SoundEffect` and it should only be defined when the passed `Effect` *is* a `SoundEffect`.

Note that if you implement traverse and `StringAdder` properly, you should be able to replace your `CompositeEffect.render` code with a call to `traverse` with `RenderVisitor` in the second argument list, and it should function the same way.

Finally, you are responsible for implementing the `MortarSizeVisitor` class. It relies on the fact that the `SkyEffect` trait has a self-type of `Effect` in order to allow for collection of information about what mortar tube is needed to launch a given effect. Here is a description of its methods:

- `select`—A partial function that is only defined the effect has a `needsMortar` property and it evaluates to true.
- `apply(effect, t, step)`—This method returns the size of mortar tube needed for `effect`. It also stores information about this size locally for later use (for the return value of `listing`)
- `identity`—This method returns the identity value for the operation defined in `combine`.
- `combine(left, right)`—This method returns the max of two value of type `Meters`.
- `listing`—This method returns an immutable map of mortar-tube sizes (in `Meters`) to counts (how many effects require a mortar tube of that size). The map should be sorted by tube size. Note that this method is not an `EffectVisitor` method. This is a method that users are expected to call for additional information after the visitor has been used in traversing a `CompositeEffect`.

Notes

- Source code is submitted for testing using Git (master branch). The first submission must be tagged 4a and the second submission 4b (even if the first submission is missing). If modifications to the code are needed after tagging a submission, a new tag should be created of the form 4a.1, 4a.2, 4b.1, etc. Do not delete tags!
- In addition to this Git submission, students need to upload on MyCourses a sample of their code for review. Code should be uploaded as a pdf file created from a pretty-printer (e.g., a2ps, pandoc, highlight, IntelliJ, etc.). The sample should be about one page long, and can include specific questions (e.g., “This code is very convoluted; could it be made simpler?” or “This code seems to be very slow and I don’t know why” or “I’m very proud of this code, which follows a pure functional pattern; what do you think?”).
- I would suggest starting implementation with the value classes, followed by the `render` methods of the high-level `Effect` traits. Then work on the composition operators and the `render` method of `CompositeEffect`. Implementation having to do with the `traverse` method of `CompositeEffect` and `EffectVisitor`s should wait until the other portions are in decent shape.
- When making your implicit conversions from `Double` to `Decibels`, `Seconds`, and `Meters`, it won't work with an implicit function. You'll need to use an implicit class with an appropriate method, as was done with `Int2ToTimes` in one of the worksheets (12-2).
- You may find it helpful to add other methods to the value types as you implement code in other parts of the project.