Beat It

Due: Monday, October 1, 2012 at 11:59pm Pacific
Complete this assignment on your own — 140 total points..

Overview

In this assignment, we take a little break from implementing internal DSLs to focus on semantics. We'll implement something that's conceptually simple — integer and modular division — by composing a series of adders. Then we'll use our semantics to count the number of beats in a measure of "music". The goal of this assignment is to explore Scala's scalable components: traits, classes, and objects. I strongly suggest reading Sections 12.1, 12.2, 12.3, and 12.5 of *Programming in Scala 2/e* before starting the assignment.

1. Set the Beat *(13 total points)*

In this part of the assignment, you'll implement a tiny piece of a domain — music! In particular, you'll implement very simple data structures that correspond to musical durations. A *duration* is a musical entity that lasts for a specified number of beats. Beats are, therefore, a unit of musical time. In this domain, there are three beat durations: *quarter beats* (which have duration 1), *half beats* (which have duration 2), and *whole beats* (which have duration 4). Furthermore, there are two kinds of beats: *notes* and *rests*. Combining the beat durations and the kinds of beats gives the domain six distinct objects.

To implement musical durations, you'll modify the file `Beats.scala`. If you open this file, you will see that it contains the following components:

The "Materials" section contains information on how to get the files you need.

`Duration`: an abstract class that corresponds to a *duration*.
`Note`: a trait that corresponds to the *note* kind of duration.
`Rest`: a trait that corresponds to the *rest* kind of duration.

Change the definitions of `Note` and `Rest` to be the following:

```
abstract class Note extends Duration
abstract class Rest extends Duration
```

Compile `Beats.scala` and observe the result.

(a) What did you observe? Write your answer in `Answers.txt` *(1 point)*

Revert the definitions to traits, re-compile and observe the results.

(b) What did you observe? Write your answer in `Answers.txt` *(1 point)*

The difference in these two behaviors can be explained by the fact that `extends` means one thing when we use it with two classes and something else when we use it with a class and a trait.

(c) What does `class C1 extends C2` mean (when `C2` is a class)? Write your answer in `Answers.txt` *(3 points)*

(d) What does `trait T extends C` mean (when `C` is a class)? Write your answer in `Answers.txt` *(3 points)*

(e) Complete the definition of the domain by defining the following objects in `Beats.scala`: *(5 points)*

```
QuarterNote
HalfNote
WholeNote
QuarterRest
HalfRest
WholeRest
```

Hint: Each of these objects is a mix of two components: its duration and its kind.

That's it for now! Our ultimate goal is to count a series of beats to see how many measures they compose. Before we can do that, though, we must first learn to count.

2. Learn to Count *(117 total points)*

When we implement a DSL, we often can re-use some of the host language's features in our DSL. Of course, if the host language lacks the feature that we need, we have to implement it ourselves (or change our host language, but alas, that's not an option today!). In this part of the assignment, we're going to imagine that Scala has neither modular nor integer division. Instead, we'll implement them ourselves by constructing these operations from a combination of lower-level operations.

We'll start with a simple counter. If you look at the file `Counting.scala`, you will see that it contains the following components:

`VarInt`: a class that defines a simple wrapper for an `Int` whose value can vary.
`Incrementable`: a trait that specifies the `inc` behavior. Note that `inc` results in an `Int`.
`Resetable`: a trait that specifies the `reset` behavior. Note that `rest` results in an `Int`.
`Counter`: an incrementable `Int`.
`Alarmable`: an extension to `Incrementable` that "trips" when it hits a pre-defined limit.
`LimitException`: a custom exception to be used with `Alarmable`.

`Alarmable` *and* `LimitException` *are commented out, for now.*

(a) In the Scala REPL, load `Counting.scala`, create an instance of `Counter`, and invoke its `inc` method three times. Copy-paste these commands and results into the file `Answers.txt`. *(2 points)*

Uncomment the commented-out code, compile `Counting.scala` and observe the result.

(b) What did you observe? Write your answer in `Answers.txt`. *(1 point)*

(c) Why did this happen, i.e., if this wasn't how Scala behaved, what would be the consequences? Write your answer in `Answers.txt`. *(5 points)*

Modify `Counting.scala` to fix the problem.

It might take some trial and error to get (d). It's possible to do it in one legible line of code.

(d) In the Scala REPL, load `Counting.scala`. Create an instance of an "alarm-able" counter with a limit of `3`. Invoke this object's `inc` method three times. Copy-paste these commands and results into the file `Answers.txt`. *(7 points)*

(e) Traits can't have parameters, but there's a workaround. What is it? Write your answer in `Answers.txt` *(2 points)*

With the components we've already created, we can now build a low-level *modular counter*: A modular counter is an alarm-able, reset-able counter. A user of a modular counter can call `inc` until the alarm trips. The user can also call `reset` to start the counter over from 0. Here's an example of its use:

```
scala> val c = new ModularCounter(3)
c: ModularCounter = ModularCounter@448bcf49

scala> c.inc()
res6: Int = 1

scala> c.inc()
res7: Int = 2

scala> c.inc()
LimitException$: Limit reached
       at LimitException$.<clinit>(<console>)
       at Alarmable$class.inc(<console>:16)
       at ModularCounter.inc(<console>:13)
       at .<init>(<console>:16)
       at .<clinit>(<console>)
       ...

scala> c.reset()
res9: Int = 0
```

(f) Modify `Counting.scala` to add a definition for `ModularCounter`. Furthermore, add a requirement to the class that the modulus (i.e., the number that trips the alarm) must be greater than 0. *(25 points)*

It's a bit of a pain for the user to manually reset the `ModularCounter` when it trips. It'd be nicer if we made an `AutoModularCounter` that behaved as follows:

```
scala> val c = new AutoModularCounter(3)
c: AutoModularCounter = AutoModularCounter@4eb3ea0f

scala> c.inc()
res1: Int = 1

scala> c.inc()
res2: Int = 2

scala> c.inc()
res3: Int = 0

scala> c.inc()
res4: Int = 1
```

(g) Modify `Counting.scala` to add a definition for `AutoModularCounter`. This class should extend `ModularCounter` and override its `inc` method. *(25 points)*

We've now implemented modular addition, but what we *want* is to pair integer division with modular division. These two operations together would tell us how many times a counter has tripped and how many increments have happened since the last trip. Such a counter would behave as follows:

With this information, we could, for example, count the number of full measures denoted by a sequence of beats, determine whether that sequence ends with a partial measure, and determine the number of beats in the partial measure.

```
scala> val c = new DivModCounter(3)
c: DivModCounter = DivModCounter@78aeaea6

scala> c.inc()
res5: (Int, Int) = (0,1)

scala> c.inc()
res6: (Int, Int) = (0,2)

scala> c.inc()
res7: (Int, Int) = (1,0)

scala> c.inc()
res8: (Int, Int) = (1,1)

scala> c.inc()
res9: (Int, Int) = (1,2)

scala> c.inc()
res10: (Int, Int) = (2,0)
```

We can implement `DivModCounter` by composing our existing counters. Specifically, we can define it as an `Incrementable` class that contains:

> `div`: a `Counter`, that tracks how many times the modulus has tripped.
> `mod`: a `ModularCounter` that tracks how many `inc`s have executed since the last trip.

There's just one problem: a `DivModCounter`'s `inc` method results in a pair of `Int`s, i.e., a value of type `(Int, Int)`, but `Incrementable` defines `inc`'s result to be a single `Int`. We're in a bind. Sometimes we need to count `Int`s, and sometimes we need to count pairs of `Int`s. We could copy-paste the code we've written and specialize it for the two different circumstances, but that's not ideal. Fortunately, Scala (like many other languages) provides a more scalable solution: *type parameterization*. We can modify our code to take advantage of type parameterization like so:

If you're not sure (or don't remember) what type parameterization is, think:
`List[String]`
Here, `List` *is defined with a type parameter, whose value we specify to be* `String`.

i. Define each trait to take a type parameter that corresponds to the result of its operation.
ii. Modify each class that mixes in a trait so that it instantiates its trait(s) with type `Int`.

Listing 19.4 on page 428 of Programming in Scala, 2/e *may help.*

Once we've added type parameterization, we can define `DivModCounter` as above.

(h) Modify `Counting.scala` to add type parameterization. *(25 points)*

(i) Define `DivModCounter`. *(25 points)*

3. Count the Beats *(10 total points)*

Now we're ready to use integer and modular division to count the number of measures described by a sequence of beats. Examine the file `BeatCounter.scala`.

> `countMeasures`: an incomplete function that is intended to count the number of full measures denoted by a sequence of beats, determine whether that sequence ends with a partial measure, and determine the number of beats in the partial measure.
> `beats`: a sequence of beats
> `(measures, remainingBeats)`: the result of the call `countMeasures(beats)`

Make sure you've compiled the other files, too!

Compile this file and run the program with the command `scala BeatCounter`.

(d) We're running a program without defining a `main` function. How?! *(3 points)*

(e) Implement `countMeasures` using a `DivModCounter` to compute the results. *(7 points)*

For this assignment, we'll assume common time, *i.e., that each measure has four beats and that a quarter is one beat.*

Feedback *(participation points)*

Answer the questions in the file `feedback.txt`.

Materials

The materials are available in the Subversion repository. In your working copy, execute:

```
svn copy https://svn.cs.hmc.edu/dsl/fall12/hw/5 hw5
```

Modify the materials to supply your answers and use the instructions below to submit.

Submitting Your Solution

Make sure you're in the `hw4` directory of your working copy and execute:

```
svn commit
```

Committing is like voting in Chicago: you should do it early and often.

Collaboration and Honor Code

I expect you to abide by the Harvey Mudd Honor code. Your solution to this assignment should be produced by you alone. You should discuss concepts at a high level with any student in the class and / or with the course staff. However, you may not copy solutions from anyone, nor may you search for solutions on the Internet.

If you have any questions about what behavior is acceptable, it is your responsibility to come see me before you engage in this behavior. I will be happy to answer any of your questions.