

Collaboration between students is not allowed on this assignment. You are responsible for keeping your code hidden from all other students. Turn in your work on this assignment before 22h00 on the due date by following the instructions on the class web site

<http://www.ccs.neu.edu/home/riccardo/csu370>

Every source file you submit should begin with a block comment that lists

- (1) Your name, as you want the instructor to write it.
- (2) Your email address.
- (3) Any remarks that you wish to make to the instructor.

Part of your grade will depend on the quality and correctness of your code; part will depend on the readability of your code (comments and indentation), and part will depend on your following the procedure above for submitting your work. Late assignments may be discounted, and very late assignments may be discarded.

## Characters and Behaviors

This is the second of two homeworks on animation. Last homework, we saw the basics of movies, where a movie is just a sequence of frames, where a frame is just a picture to be drawn on the screen. To play a movie, we just repeatedly display every frame of the movie at some rate, clearing the screen whenever we get to the next frame. By controlling the rate at which frames are displayed, we can control how fast the movie goes.

As we saw, though, building a sequence of frames by hand is a bit of a pain—we have to draw the frames one by one, and add them together. Much better is to do things the way real animators do it: define characters, and animate them separately by applying a *behavior* to them. A behavior can represent a motion (such as moving in a circle), or a scaling (such as shrinking and enlarging), or a rotation (such as spinning). Behaviors can be composed, such as moving in a circle while spinning.

## Some Support Code

Two pieces of code have been provided for you.

First off, I've given you an updated MFrame ADT, augmenting the signature on the last homework, with additional operations. Here is the signature of the new MFrame class which is available from the course web site:

```
public static MFrame empty ();
public static MFrame line (Line);
public static MFrame merge (MFrame, MFrame);

public abstract MFrame move (Point);
public abstract MFrame scaleFrom (Point, double);
public abstract MFrame scale (double);
public abstract MFrame rotateAround (Point, double);
public abstract MFrame rotate (double);

public abstract boolean hasElement ();
public abstract Line current ();
public abstract FuncIterator<Line> advance ();
```

As last time, the MFrame ADT is its own iterator (as opposed to relying on a `getFuncIterator` operation), and so it implements the `FuncIterator<Line>` interface. Just like last time, a frame is just a sequence of lines. The movie player has a window that goes from (0,0) to (500,500), and a frame is just a picture drawn in that rectangle. There are three creators for frames: `empty()`, that creates an empty frame, `line(l)`, that creates a frame with a single line  $l$  in it, and `merge( $f_1, f_2$ )`, that creates a frame with all the lines in frames  $f_1$  and  $f_2$  in it. These are all as you had last homework.

What's new then? A few methods to operate on frames:

- Method `move(p)` moves every line in the frame by the vector described by point  $p$ —if  $p$  is the point  $(x, y)$ , then it moves every line by adding  $x$  to the x-coordinate of its start and end line, and adding  $y$  to the y-coordinate of its start and end line.
- Method `scaleFrom(p,d)` scales the frame by a factor  $d$  taking point  $p$  as the fixed point—it expands or shrinks the picture around point  $p$ .
- Method `scale(d)` scales the frame by factor  $d$  around the *center point* of the picture, computed as the center of the smallest box containing all the lines in the frame.
- Method `rotateAround(p,d)` rotates the picture by  $d$  degrees counterclockwise around point  $p$ .
- Method `rotate(d)` rotates the picture by  $d$  degrees counterclockwise around the center point of the frame.

The other class I've given you is a class `Stream<T>` of streams of values of type  $T$ , with the following signature:

```

public static <U> Stream<U> empty ();
public static <U> Stream<U> cons (U, Stream<U>);
public static <U> Stream<U> append (Stream<U>, Stream<U>)
public static <U> Stream<U> loop (Stream<U>);
public static <U> Stream<U> array (U[], int);

public abstract boolean hasElement ();
public abstract T current ();
public abstract FuncIterator<T> advance();

```

A stream is just a glorified functional iterator that supplies values of some type `T`. An `empty()` stream is just that, empty. A `cons(v, str)` stream is a stream that first supplies value `v`, and then supplies all the values in stream `str`. A `append(str1, str2)` stream first supplies all the values in `str1`, then all the values in `str2`. A `loop(str)` stream supplies all the values in `str`, and then loops, supplying all the values in `str` again, and again, and again. A `array(arr, i)` stream supplies all the values in array `arr` starting from index position `i` (where index position 0 is the beginning of the array). Streams are handy to help define behaviors, as we will see shortly.

## Question 1: Behaviors

A behavior is just a sequence of transformations you can perform on frames. For instance, a left-to-right motion may be represented as the sequence of transformations that moves a frame left repeatedly. A shrinking behavior may be represented as the sequence of transformations that shrinks a frame by a small amount repeatedly.

The Behavior ADT has the following signature:

```

public static Behavior move (Stream<Point>);
public static Behavior scale (Stream<Double>);
public static Behavior rotate (Stream<Double>);

public abstract boolean hasElement ();
public abstract Function<MFrame, MFrame> current ();
public abstract FuncIterator<Function<MFrame, MFrame>> advance ();

```

The Behavior ADT is, just like MFrame ADT, its own iterator, and so it implements the `FuncIterator<Function<MFrame, MFrame>>` interface. Recall from lecture that the `Function<T, U>` interface represents functions from values of type `T` to values of type `U`:

```

public interface Function<T, U> {
    public U call (T arg);
}

```

Thus, `Function<MFrame,MFrame>` represents functions from `MFrame` to `MFrame`, that is, frame-transformation functions. A `move(str)` behavior takes a stream of points and produces a sequence of transformations that move a frame by the specified x- and y-distances, using the `move` operation on frames. The `scale(str)` behavior takes a stream of doubles<sup>1</sup> and produces a sequence of transformations that scale a frame by the specified ratio, using the `scale` operation on frames. The `rotate(str)` behavior takes a stream of doubles and produces a sequence of transformations that rotate a frame by the specified degrees, using the `rotate` operation on frames.

The following specification makes the above precise:

```
move(str).hasElement() = true      iff str.hasElement()=true
scale(str).hasElement() = true     iff str.hasElement()=true
rotate(str).hasElement() = true   iff str.hasElement()=true

move(str).advance() = move(str.advance())
scale(str).advance() = scale(str.advance())
rotate(str).advance() = rotate(str.advance())

move(str).current() =
    "an instance implementing Function<MFrame,MFrame> that
    moves a frame by str.current()"
scale(str).current() =
    "an instance implementing Function<MFrame,MFrame> that
    scales a frame by str.current()"
rotate(str).current() =
    "an instance implementing Function<MFrame,MFrame> that
    rotates a frame by str.current"
```

Note that the specification for `current()` is not algebraic, but should be reasonably clear. Of course, you'll want to define classes implementing the `Function<MFrame,MFrame>` interface from which to get the corresponding instances. You can simply put them in the same file as the `Behavior` class by not making them public. Or nest them instance the appropriate concrete subclasses of the `Behavior` abstract class. Either way is fine with me.

Your job for this question is to implement the Behavior ADT in Java<sup>2</sup> **using the recipe we saw in class to go from an ADT to an implementation**. That is, implement an abstract class `Behavior`, with concrete subclasses. Your implementation, as usual, should match the signature above, and satisfy the specification. The code for the `Behavior` class

---

<sup>1</sup>Recall that `Double` is a class that wraps around values of primitive type `double`, and while the Java compiler will most of the time translate back and forth automatically between `Double` and `double`, you can do it by hand by using `new Double(v)` to create a `Double` form a `double v`, or `V.doubleValue()` returns the `double` corresponding to `Double V`.

<sup>2</sup>using Sun's Java 2 Runtime Environment, Standard Edition, version 5.0.

and associated subclasses should be in a single source file, `Behavior.java`. Again, your implementation should only make the operations in the signature public. Everything else should be private.

What to submit for this question: submit your file `Behavior.java`.

## Question 2: Updated Movie ADT

A frame is a single frame of a movie. A movie is a sequence of frames. Here is Movie ADT:

```
public static Movie empty ();
public static Movie frame (MFrame);
public static Movie then (Movie, Movie);
public static Movie wait (int, Movie);
public static Movie overlay (Movie, Movie);
public static Movie loop (Movie);
public static Movie apply (Behavior, Movie);

public boolean hasElement ();
public MFrame current ();
public FuncIterator<MFrame> advance ();
```

A movie is just a (possibly infinite) sequence of frames. There are the creators that you would expect for movies: `empty()` creates an empty movie with no frames at all; `frame(m)` creates a movie with a single frame *m* in it. Then we have ways of creating movies by editing smaller movies together: `then(m1, m2)` creates a movie by attaching *m*<sub>2</sub> at the end of *m*<sub>1</sub> (of course, if *m*<sub>1</sub> never finishes, then *m*<sub>2</sub> will never be played, but that's okay); `wait(i, m)` adds *i* empty frames in front of *m*, effectively delayed movie *m* by *i* time units; `overlay(m1, m2)` creates a movie by simply overlaying the respective frames from *m*<sub>1</sub> and from *m*<sub>2</sub>. When played, an overlay of *m*<sub>1</sub> and *m*<sub>2</sub> ends when either *m*<sub>1</sub> or *m*<sub>2</sub> does. Finally, `loop(m)` creates a movie by simply repeatedly cycling over the frames of *m*. It is the only way we have of creating an infinite movie.

The difference from the Movie ADT from last homework is the addition of a new creator `apply` that applies a behavior to a movie. Intuitively, applying a behavior (which is just a sequence of frame transformations) to a movie (which is just a sequence of frames) consists of applying the first transformation to the first frame to produce the new first frame, the second transformation to the second frame to produce the new second frame, the third transformation to the third frame to produce the new third frame, and so on.

Movies are functional iterators implementing the `FuncIterator<MFrame>` interface. Here are the specifications:

```
empty().hasElement() = false
frame(f).hasElement() = true
```

```

then(m1,m2).hasElement() = true
  iff m1.hasElement()==true or m2.hasElement()==true
wait(i,m).hasElement() = true
overlay(m1,m2).hasElement() = true
  iff m1.hasElement()==true and m2.hasElement()==true
loop(m).hasElement() = true
  iff m.hasElement()==true
apply(b,m).hasElement() = true
  iff b.hasElement()==true and m.hasElement()==true

frame(f).current() = f
then(m1,m2).current() =
  m1.current()    if m1.hasElement()==true
  m2.current()    otherwise
wait(i,m).current() = MFrame.empty()
overlay(m1,m2).current() = MFrame.merge(m1.current(),m2.current())
loop(m).current() = m.current()
apply(b,m).current() = b.current().call(m.current())

frame(f).advance() = empty()
then(m1,m2).advance() =
  then(m1.advance(),m2)    if m1.hasElement()==true
  m2.advance()            otherwise
wait(i,m).advance() =
  wait(i-1,m)    if i>1
  m              if i=1
overlay(m1,m2).advance() = overlay(m1.advance(),m2.advance())
loop(m).advance() = then(m.advance(),loop(m))
apply(b,m).advance() = apply(b.advance(),m.advance())

```

As last time, `wait(i,m)` should throw an `IllegalArgumentException` when given an input  $i$  less than 1.

Make sure you understand how the above specification for `apply` actually achieves the description I gave earlier.

Your job for this question is to implement the Movie ADT in Java<sup>3</sup> **using the recipe we saw in class to go from an ADT to an implementation**. That is, implement an abstract class `Movie`, with concrete subclasses. Your implementation, as usual, should match the signature above, and satisfy the specification. The code for the `Movie` class and associated subclasses should be in a single source file, `Movie.java`. Again, your implementation should only make the operations in the signature public. Everything else should be private.

<sup>3</sup>using Sun's Java 2 Runtime Environment, Standard Edition, version 5.0.

I made this easy for you by provide most of the implementation of `Movie` already—I just don't want you to start handicapped because you may not have completed the last homework correctly. If that's the case, make sure you understand how I implemented my version.

We will use the class `MoviePlayer` from last homework to play movies. Recall that it supports the following methods:

```
public static void initialize ();
public static void draw (MFrame);
public static void play (int, Movie);
```

You need to call `MoviePlayer.initialize()` once at the beginning of your main program—it creates the window, with a size of 500 by 500. Method `draw` takes a `MFrame` draws it on the screen. (It can be useful to try out the new frame operations.) For `play`, the integer supplied as first argument is the rate at which to play the movie. Precisely, it is the number of 1/10th of seconds to wait between frames. So passing in an argument of 10 plays the movie at a single frame per second. Playing at rate of 100 plays the movie at a single frame per 10 seconds, and so on. I usually use a rate of 1.

What to submit for this question: submit your file `Movie.java`.