# Cheloniidae

Spencer Tipping

December 27, 2010

# Contents

# Preface

Cheloniidae is the product of two orthogonal goals. One is to write the highest-capacity turtle graphics system possible, and the other is to write Java code with the elegance of Lisp or Haskell. I'm not sure whether I've managed to achieve these goals yet, but this paper explains the thoughtwork behind the code with a linear approach.

There are a number of ideas that as far as I know originated with this project. One is the notion of having turtles navigate a three-dimensional space instead of a two-dimensional one. Another is using turtles to render polygons, and the necessary mathematical constructs required for that to make sense. A third notion is the algebra underlying turtle commands and the Lisp-like model used to transform them.

I recommend reading through this paper if you are interested in:

1. Using Java as a functional programming language

2. Techniques for writing very concise code

3. Extending Cheloniidae

4. Writing mathematical systems in Java

## Website

The Cheloniidae website is http://spencertipping.com#section=cheloniidae. This website includes the downloadable distribution, some examples, and a place to download older versions of the software.

Any suggestions for improvements are welcome and should be directed to spencer@spencertipping.com.

# Part I

# Conceptual Basis

# Chapter 1

# Design Patterns

## 1.1 Public Final

In languages like Haskell it is perfectly acceptable to pass exposed data around because it is immutable. The danger of neglecting to enforce access in other languages is generally not that someone will rely on implementation details (though sometimes it is), but rather that someone will change a value without your permission. One way around this is to write accessor methods, which normally are implemented this way:

```
private int x;
public int  getX ()       {return x;}
public void setX (int _x) {x = _x;}
```

The advantage of a set of accessor methods is twofold:

1. Access, particularly mutation, has a thread-path and can be traced easily, e.g.

   ```
   public void setX (int _x) {
     new Exception ("X set to " + _x).printStackTrace ();
     x = _x;
   }
   ```

2. The implementation details of the class are hidden, allowing internal refactoring without breaking other code.

Cheloniidae takes an alternate approach to instance data. All instance data is initialized in the constructor, and all fields are public and final. This ensures that an object is initialized atomically, and that it is immutable after initialization.

Along with the public-final design pattern, I also adhere to the practice that all local variables, with very few exceptions, are declared as final. This lays the groundwork for a functional and mathematical model of coding.

## 1.2 Interfaces as Sets

The primary method of categorization in mathematics is grouping objects into sets. Some of these sets reflect intrinsic attributes of objects – one example is the statement that $x \in \mathbb{R}$.[1] Other sets can be seen as objects for which a proposition holds; e.g. $x \in \mathbb{N}$ iff $x > 0$ and $x \in \mathbb{Z}$.

Java reflects the first use of sets – we can tag classes to say that their instances have certain intrinsic properties. To simulate the second use of sets, I've defined a framework of predicates in .

The interface-as-a-set design pattern states two things:

1. Every conceptually distinct use of an object should define an interface for that purpose. Often this interface will be empty.

2. The `Object` type should never be used.

## 1.3 No Void Methods

Very few methods should ever return void. Better is to return the object to which the method applies.[2] This way, you can chain method calls along and perform object mutation in one statement as opposed to several. This contributes to code readability, since mutations on the same object are grouped, and it also contributes to brevity.

Most classes in Cheloniidae are stateless, but one exception is the `Vector` class defined in . Methods in this class, such as `addScaled` and `multiply`, modify the original vector in-place and return a self-reference. This allows you to write code such as this:

```
// v1 becomes the average of v1 and v2
some_function (v1.multiply (0.5).addScaled (v2, 0.5));
```

---

[1] Not really, but let's eliminate complex and Hamiltonian numbers for the sake of example.

[2] Sometimes it is necessary to return void to work around deficiencies in Java's type checker. In a language with covariance and contravariance annotations, such as Scala, this would probably not be the case.

# Chapter 2

# Turtles

Like mathematical axiom systems, Cheloniidae is minimalistic in what it assumes about the implementation of a model. This approach can lead to some confusion because of what we take for granted. For instance, most turtle graphics systems implement a turtle as being a member of a two-dimensional plane, so the turtle has a location and a heading angle. Many allow the turtle to vary in its color and line size, so these too are part of a turtle model.

These things are not assumed in Cheloniidae. In fact, not even the coordinate model is assumed. A turtle (section 2.1) is an object with the following properties:

1. It can generate a list of `RenderActions` (section 9.2) to reflect things it has drawn.

2. It can save and restore its state (section 3.4).

3. It can be cloned (chapter 6).

4. It can be tagged with one or more attributes (section 4.3).

5. It can execute turtle commands (chapters 3 and 8).

6. It has a window that receives its output (section 9.5).

The result of such a general setup is that turtles can reside in non-Euclidean spaces, draw curved lines, or define custom commands for themselves. Also, a "turtle" can represent multiple turtles (section 3.2) or static objects.

## 2.1   Turtle

This interface, like many others in Cheloniidae, has a type parameter that is set to the class that implements it. This notion corresponds to a convention among Haskell typeclasses:

```
class Eq a where
  (==) :: a -> a -> Bool
```

7

wherein the implementation's type is a restriction on the parameters. In the case of `Turtle`, it is a restriction on the return types of the methods, since often we will want to chain method invocations along and we don't want Java to demote the type of the returned self-reference to a regular `Turtle`. When you run across this style of code in the Cheloniidae code base, you can interpret a return type of `T` as a signal that a method should return `this`.

```
1   package cheloniidae;
2   public interface Turtle<T extends Turtle>
3             extends Renderable, Cloneable, HasAttributes {
4     public TurtleState serialize   ();
5     public T           deserialize (TurtleState t);
6
7     public T run   (TurtleCommand c);
8     public T clone ();
9
10    public TurtleWindow window ();
11    public T           window (TurtleWindow _window);
12  }
```

## 2.2 BasicTurtle

The `BasicTurtle` class provides default implementations for the functionality that is likely to be invariant with the type of turtle used. Specifically, it implements the following assumptions made by the `Turtle` interface:

1. Basic handling of attributes

2. Belonging to a window

3. Default behavior for running a turtle command

4. Default implementation of the `map` method, used for acting as a turtle command for propagating state to other turtles

Most turtles are subclasses of `BasicTurtle` because these defaults usually work. However, there are exceptions; `TurtleGroup` (section 3.2), for instance, provides its own implementation of `run()` to support command distribution across turtles that belong to that group.

```
1   package cheloniidae;
2   import cheloniidae.commands.*;
3   import java.util.*;
4
5   public abstract class BasicTurtle<T extends BasicTurtle>
6              extends Replicable<T>
7            implements Turtle<T>, SupportsWindow {
8
9     public static class State
10             extends ImmutableTurtleState
11           implements TurtleState, TurtleCommand {
12
13      public final Set<Attribute> attributes = new TreeSet<Attribute> ();
14      public State (final Set<Attribute> _attributes) {attributes.addAll (_attributes);}
15
16      public State applyTo (final Turtle t) {
17        t.attributes ().addAll (attributes);
18        return this;
19      }
20    }
```

8

```
21
22    public final Set<Attribute> attributes = new TreeSet<Attribute> ();
23
24    protected TurtleWindow window = null;
25
26    public Set<Attribute> attributes ()                              {return attributes;}
27    public T              attribute  (final Attribute _attribute) {attributes.add (_attribute);
28                                                                    return (T) this;}
29
30    public TurtleWindow   window     ()                              {return window;}
31    public T              window     (final TurtleWindow _window) {window = _window; return (T) this;}
32
33    public TurtleState serialize () {return new State (attributes);}
34
35    public T run (final TurtleCommand c) {
36      c.applyTo (this);
37      return (T) this;
38    }
39
40    public TurtleCommand map (final Transformation<TurtleCommand> t) {
41      final TurtleState serialized = this.serialize ();
42      if (serialized instanceof TurtleCommand) return ((TurtleCommand) serialized).map (t);
43      else                                     return null;
44    }
45 }
```

## 2.3  EuclideanTurtle

Euclidean turtles make the assumption that they reside in a space with Euclidean geometry and travel in straight lines. Things that can be reasonably concluded from this include:

1. The turtle's position can be described in Cartesian coordinates. This implies that all lines generated by the turtle will have endpoints with Cartesian coordinates, so a `CartesianLine` (section 10.2) is a reasonable representation for the output.

2. The lines generated by the turtle can have thicknesses (referred to as "size" in this class) and solid colors. In theory we could implement gradients and such, but I haven't gotten around to that yet.

3. Since the turtle moves in straight lines, it makes sense to talk about a direction. However, it is not yet clear how that direction should be computed, so it is abstract.

Two nested classes are defined: `EuclideanTurtle.View` and `EuclideanTurtle.State`. `View` is a `RenderAction` that draws a circle and a line indicating the turtle's position and direction. When the turtle is asked for its list of render actions, it includes an instance of `View` in its result.

The `State` class stores the extra state for a Euclidean turtle. To keep the code compact, the inheritance chain of turtle states mirrors the inheritance chain of turtles whenever possible.

```
1    package cheloniidae;
2
3    import cheloniidae.commands.*;
4    import java.awt.BasicStroke;
5    import java.awt.Color;
6    import java.awt.Graphics2D;
7
8    import java.util.List;
9    import java.util.ArrayList;
10   import java.util.SortedSet;
```

```
11   import java.util.TreeSet;
12   import java.util.Set;
13
14   public abstract class EuclideanTurtle<T extends EuclideanTurtle>
15                   extends BasicTurtle<T>
16              implements SupportsPosition<T>, SupportsMove<T>, SupportsJump<T>, SupportsLineSize<T>,
17                         SupportsLineColor<T>, SupportsVisible<T>, TurtleCommand {
18
19     public static class View extends ViewportCaching implements RenderAction {
20       public final EuclideanTurtle turtle;
21       public View (final EuclideanTurtle _turtle) {turtle = _turtle;}
22
23       public double computeDepth (final Viewport v)
24         {return v.transformPoint (turtle.position ()).length ();}
25
26       public void render (final Viewport v) {
27         final Vector tp = v.transformPoint (turtle.position ());
28         final Vector td = v.transformPoint (turtle.position ().clone ().add (turtle.direction ()));
29         if (tp.z > 0 && td.z > 0) {
30           final double scale = 4.0 * v.scaleFactor () / tp.z;
31
32           // pp = perceived position, pd = perceived direction.
33           // These are the position and direction as seen by the user.
34           final Vector pp = v.projectPoint (tp);
35           final Vector pd = v.projectPoint (td);
36
37           final Graphics2D g = v.context ();
38           g.setStroke (new BasicStroke ((float) (scale / 16.0)));
39           g.setColor  (turtle.color ());
40           g.drawOval  ((int) (pp.x - scale), (int) (pp.y - scale), (int) (scale * 2.0), (int) (scale * 2.0));
41           g.drawLine  ((int) pp.x, (int) pp.y, (int) pd.x, (int) pd.y);
42         }
43       }
44     }
45
46     public static class State extends BasicTurtle.State implements TurtleState, TurtleCommand {
47       public final Vector position;
48       public final double size;
49       public final Color  color;
50
51       public State (final Set<Attribute> _attributes, final Vector _position,
52                     final double _size, final Color _color)
53         {super (_attributes); position = _position.clone (); size = _size; color = _color;}
54
55       public State applyTo (final Turtle t) {
56         super.applyTo (t);
57         new Sequence (new Position  (position),
58                       new LineSize  (size),
59                       new LineColor (color)).applyTo (t);
60         return this;
61       }
62     }
63
64     protected final List<CartesianLine> lines    = new ArrayList<CartesianLine> ();
65     protected final View                view     = new View (this);
66     protected final Vector              position = new Vector ();
67     protected       double              size     = 0.25;
68     protected       Color               color    = new Color (0.2f, 0.3f, 0.3f, 0.3f);
69     protected       boolean             visible  = true;
70
71     public Vector              position ()                      {return position;}
72     public T                   position (final Vector _position) {position.assign (_position);
73                                                                     return (T) this;}
74     public double              size     ()                      {return size;}
75     public T                   size     (final double _size)    {size = _size; return (T) this;}
76     public double              lineSize ()                      {return size ();}
77     public T                   lineSize (final double _size)    {return size (_size);}
78     public Color               color    ()                      {return color;}
79     public T                   color    (final Color _color)    {color = _color; return (T) this;}
```

```
80    public Color                      lineColor ()                     {return color ();}
81    public T                          lineColor (final Color _color)   {return color (_color);}
82    public boolean                    visible   ()                     {return visible;}
83    public T                          visible   (final boolean _visible) {visible = _visible; return (T) this;}
84
85    public abstract Vector direction ();
86
87    public T jump (final double distance) {position.addScaled (this.direction (), distance); return (T) this;}
88    public T move (final double distance) {final Vector oldPosition = position.clone ();
89                                    return line (oldPosition,
90                                                position.addScaled (this.direction (), distance));}
91    public T line (final Vector p1, final Vector p2)
92      {lines.add (new CartesianLine (p1, p2, size, color)); return (T) this;}
93
94    public SortedSet<RenderAction> actions (final Viewport v) {
95      final SortedSet<RenderAction> result = new TreeSet<RenderAction> (new PerspectiveComparator (v));
96      if (visible) result.add (view);
97      for (final RenderAction r : lines) if (v.shouldCancel ()) break;
98                                    else                   result.add (r);
99      return result;
100   }
101
102   public State serialize   ()                      {return new State (attributes, position, size, color);}
103   public T     deserialize (final TurtleState t) {if (t instanceof TurtleCommand)
104                                                ((TurtleCommand) t).applyTo (this);
105                                            return (T) this;}
106   public T applyTo (final Turtle t) {
107     serialize ().applyTo (t);
108     return (T) this;
109   }
110
111   public String toString () {return "Euclidean turtle: location = " + position ().toString () +
112                                    ", direction = " + direction ().toString ();}
113 }
```

## 2.4   CartesianTurtle

The new assumption made by the Cartesian turtle (perhaps a misnomer from a mathematical
perspective) is that the direction is a persistent vector with Cartesian coordinates.  That is, it is
invariant with the position of the turtle.

Making this assumption is subtly different from stating that the turtle always travels in straight
lines.  While it is true that the limit of the turtle's movement is guaranteed to be straight in a
Cartesian direction model (this was not true in the Euclidean model), no matter how small the
distance traveled, the weaker Euclidean statement that any single turtle movement will yield a
straight path holds in both cases. Once using a Cartesian model, in other words, we are guaranteed
that a movement of distance $x$ followed by a movement of distance $y$ will yield the same turtle
position as a movement of distance $x + y$.

```
1   package cheloniidae;
2
3   import cheloniidae.commands.*;
4
5   import java.awt.Color;
6   import java.util.Set;
7
8   public abstract class CartesianTurtle<T extends CartesianTurtle> extends EuclideanTurtle<T>
9   implements SupportsDirection<T> {
10
11    public static class State extends EuclideanTurtle.State {
12      public final Vector direction;
13
```

```
14      public State (final Set<Attribute> _attributes, final Vector _position, final double _size,
15                    final Color _color, final Vector _direction) {
16        super (_attributes, _position, _size, _color);
17        direction = _direction.clone ();
18      }
19
20      public State applyTo (final Turtle t) {
21        new Direction (direction).applyTo (t);
22        super.applyTo (t);
23        return this;
24      }
25    }
26
27    protected Vector direction = new Vector (0, 1, 0);
28
29    public Vector direction ()                        {return new Vector (direction);}
30    public T       direction (final Vector _direction) {direction.assign (_direction); return (T) this;}
31
32    public State serialize () {return new State (attributes, position, size, color, direction);}
33  }
```

## 2.5  RotationalCartesianTurtle

The rotational Cartesian turtle, as the name suggests, uses a rotational model to manipulate the Cartesian direction unit vector. To do this, three methods are provided:

1. `turn()`: Rotates the direction about the turtle's "up" vector. Intuitively, this is like turning a car.

2. `bank()`: Rotates the turtle's "up" vector about the turtle's direction. Intuitively, this corresponds to tilting.

3. `pitch()`: Rotates both the turtle's direction and its "up" vector about the cross product of the two. This corresponds to an airplane diving or pulling up.

OK, so what is the "up" vector? It's an extra piece of information that is stored with the turtle so that it knows how to turn properly. In this implementation it's called `directionComplement`. When the turtle is initialized, its `directionComplement` points directly towards the camera and its `direction` points straight up (towards the top of the screen). These two vectors should always be at right angles to one another.

```
1   package cheloniidae;
2
3   import cheloniidae.commands.*;
4
5   import java.awt.Color;
6   import java.util.Set;
7
8   public abstract class RotationalCartesianTurtle<T extends RotationalCartesianTurtle> extends CartesianTurtle<T>
9   implements SupportsDirectionComplement<T>, SupportsPitch<T>, SupportsBank<T>, SupportsTurn<T> {
10
11    public static class State extends CartesianTurtle.State {
12      public final Vector directionComplement;
13
14      public State (final Set<Attribute> _attributes, final Vector _position, final double _size, final Color _color,
15                    final Vector _direction, final Vector _directionComplement) {
16        super (_attributes, _position, _size, _color, _direction);
17        directionComplement = _directionComplement.clone ();
18      }
```

```
19
20     public State applyTo (final Turtle t) {
21       new DirectionComplement (directionComplement).applyTo (t);
22       super.applyTo (t);
23       return this;
24     }
25   }
26
27   protected Vector directionComplement = new Vector (0, 0, -1);
28
29   public Vector directionComplement ()                                        {return directionComplement;}
30   public T       directionComplement (final Vector _directionComplement) {directionComplement = _directionComplement;
31                                                                             return (T) this;}
32
33   public T pitch (final double angle) {final Vector axis  = direction.cross (directionComplement);
34                                         direction           = direction.rotatedAbout            (axis, angle);
35                                         directionComplement = directionComplement.rotatedAbout (axis, angle);
36                                         return (T) this;}
37   public T bank (final double angle)
38     {directionComplement = directionComplement.rotatedAbout (direction, angle); return (T) this;}
39   public T turn (final double angle)
40     {direction = direction.rotatedAbout (directionComplement, angle);            return (T) this;}
41
42   public State serialize () {return new State (attributes, position, size, color, direction, directionComplement);}
43   public String toString () {return super.toString () + ", direction complement = " +
44                                      directionComplement ().toString ();}
45 }
```

## 2.6   StandardRotationalTurtle

This is just a class to provide a shorthand for creating a rotational Cartesian turtle. Its purpose is to fill in the parameterized type. I suppose it could be a standard practice to provide these "endpoint" classes that self-parameterize; that way people using your turtle can instantiate it without parameterizing it explicitly:

```
new MyTurtle<MyTurtle> ()
```

seems redundant; most people would expect

```
new MyTurtle ()
```

as a library interface. It is important to ultimately specify the type for method chaining (e.g. `t.move (100).turn (90)`), which you will probably want unless you are using turtle commands.

```
1  package cheloniidae;
2  public class StandardRotationalTurtle extends RotationalCartesianTurtle<StandardRotationalTurtle> {
3    public StandardRotationalTurtle create () {return new StandardRotationalTurtle ();}
4  }
```

13

# Chapter 3

# Commands 1

A turtle command is a functional abstraction of the methods that a turtle provides. It can be thought of as a first-class way of referring to some turtle method such as `move` or `turn`. This lets you assemble sequences of commands at runtime and then issue them to one or more turtles.

When defining various turtle classes, most notably `EuclideanTurtle` (section 2.3), you may have noticed the implementation of interfaces such as `SupportsMove`, `SupportsTurn`, etc. These interfaces indicate that a turtle is able to understand certain turtle commands. So, for instance, if you have a Euclidean turtle `t`, then you could say this:

```
t.move (100);
```

or, equivalently:

```
t.run (new Move (100));
```

Because `Move` is a regular Java class, instances are first-class objects; thus they can be stored and transformed (see chapter 5). They can also be composed more easily than explicit method calls:

```
for (int i = 0; i < 4; ++i) t.move (100).turn (90);
```

can also be written as:

```
t.run (new Repeat (4, new Move (100), new Turn (90)));
```

The word `new` obscures the meaning here, so the `CoreCommands` class (section 11.4) defines shorthand methods to allow us to write this equivalent statement:

```
t.run (repeat (4, move (100), turn (90)));
```

The `Repeat` command is discussed further in section 8.2.

```
1  package cheloniidae;
2  import java.io.Serializable;
3  public interface TurtleCommand extends Transformable<TurtleCommand>, Serializable {
4    public TurtleCommand applyTo (Turtle t);
5  }
```

## 3.1 Application to Turtles

There are two internal steps involved in a command invocation. The first is the turtle's logic for receiving the command (in the `run()` method), which varies depending on whether the turtle is representing just itself or other turtles. The second is the command's logic for applying itself to a turtle (the `applyTo()` method), which actually carries out the action involved.

Most normal turtles specify a pass-through `run()` method, so that invoking `t.run (move (100))` translates almost directly into `t.move (100)`. However, some turtles act as proxies for other turtles (see section 3.2 for one such example). In that case, they pass the commands on, so a call to `t.run (move (100))` might translate to something like this:

```
for (final Turtle t1 : turtles) t1.move (100);
```

## 3.2 TurtleGroup

A turtle group acts as a distributive proxy for talking to a bunch of turtles at once. It maintains a list of sub-turtles and issues each command you give it to each sub-turtle. It also provides its own command interface for adding a turtle, so you can add new turtles at runtime with the command interface.

Most of the complexity of this class has to do with figuring out which commands are intended to address the turtle group itself, and which ones are supposed to be distributed to the grouped turtles. For this purpose there is a marker interface called `NonDistributiveTurtleCommand`, described in section 3.3.

```java
 1  package cheloniidae;
 2
 3  import java.util.Collection;
 4  import java.util.List;
 5  import java.util.ArrayList;
 6  import java.util.SortedSet;
 7  import java.util.TreeSet;
 8
 9  public class TurtleGroup<T extends Turtle>
10      extends BasicTurtle<TurtleGroup<T>>
11    implements Turtle<TurtleGroup<T>>, TurtleCommand {
12
13    public static class State extends ImmutableTurtleState implements NonDistributiveTurtleCommand {
14      // Turtle states are stored positionally instead of by some form of map because of replication.
15      // We need to be able to produce a turtle state object that can apply itself to a cloned group,
16      // and the cloned group won't share object identity with the original.
17      public final List<TurtleState> states = new ArrayList<TurtleState> ();
18      public State (final TurtleGroup group) {
19        for (final Turtle t : (List<Turtle>) group.turtles ()) states.add (t.serialize ());
20      }
21
22      public State applyTo (final Turtle t) {
23        t.deserialize (this);
24        return this;
25      }
26    }
27
28    public TurtleGroup (final T ...        _turtles) {for (final T t : _turtles) turtles.add (t);}
29    public TurtleGroup (final Collection<T> _turtles) {turtles.addAll (_turtles);}
30
31    protected List<T> turtles = new ArrayList<T> ();
32
33    public TurtleGroup<T> create () {
34      final TurtleGroup<T> result = new TurtleGroup<T> ();
```

```
35        for (final Turtle t : turtles ()) result.turtles ().add ((T) t.clone ());
36        return result;
37      }
38
39      public List<T>          turtles ()                              {return turtles;}
40      public TurtleGroup<T> turtles (final List<T> _turtles)    {turtles = _turtles; return this;}
41      public TurtleGroup<T> window  (final TurtleWindow _window) {super.window (_window);
42                                                                 for (final T t : turtles) t.window (window);
43                                                                 return this;}
44      public TurtleGroup<T> add (final T turtle) {
45        turtles.add (turtle);
46        turtle.window (window ());
47        return this;
48      }
49
50      public TurtleCommand adder () {
51        return new TurtleCommand () {
52          public TurtleCommand applyTo (final Turtle t) {
53            add ((T) t);
54            return this;
55          }
56
57          public TurtleCommand map (Transformation<TurtleCommand> t) {return this;}
58        };
59      }
60
61      public SortedSet<RenderAction> actions (final Viewport v) {
62        final SortedSet<RenderAction> result = new TreeSet<RenderAction> (new PerspectiveComparator (v));
63        for (final T t : turtles) for (final RenderAction r : t.actions (v)) if (v.shouldCancel ()) break;
64                                                                              else            result.add (r);
65        return result;
66      }
67
68      public State          serialize   () {return new State (this);}
69      public TurtleGroup<T> deserialize (final TurtleState state) {
70        if (state instanceof State) {
71          final List<TurtleState> stateList = ((State) state).states;
72          for (int i = 0; i < stateList.size (); ++i) turtles ().get (i).deserialize (stateList.get (i));
73        }
74        return this;
75      }
76
77      public TurtleGroup<T> applyTo (final Turtle t) {
78        serialize ().applyTo (t);
79        return this;
80      }
81
82      public TurtleGroup<T> run (final TurtleCommand c) {
83        if (c instanceof NonDistributiveTurtleCommand) c.applyTo (this);
84        else for (final T t : turtles) t.run (c);
85        return this;
86      }
87    }
```

## 3.3   NonDistributiveTurtleCommand

This is a marker interface that indicates that a turtle command should not be sent to grouped turtles, but instead should act on the turtle group proxy as a whole. It ends up coming in handy for serialization (notice that TurtleGroup.State implements it), and later on it is useful to control the order of turtle movements for triangle rendering ().

```
1    package cheloniidae;
2    public interface NonDistributiveTurtleCommand extends TurtleCommand {}
```

```
1   package cheloniidae;
2
3   public class NonDistributiveProxy implements NonDistributiveTurtleCommand {
4     public final TurtleCommand wrapped;
5     public NonDistributiveProxy (final TurtleCommand _wrapped) {wrapped = _wrapped;}
6
7     // This needs to call applyTo instead of t.run because otherwise the turtle will see
8     // the (presumably distributive) wrapped command and distribute it normally.
9     public TurtleCommand applyTo (final Turtle t) {wrapped.applyTo (t); return this;}
10    public TurtleCommand map (final Transformation<TurtleCommand> t) {
11      final TurtleCommand newCommand = t.transform (this);
12      if (newCommand == this) {
13        final TurtleCommand transformedWrapped = wrapped.map (t);
14        return transformedWrapped instanceof NonDistributiveTurtleCommand ? transformedWrapped :
15                                                    new NonDistributiveProxy (transformedWrapped);
16      } else return newCommand;
17    }
18  }
```

## 3.4  TurtleState

A turtle state is simply a snapshot of a turtle's configuration. Turtles are required to generate instances of turtle states via the `serialize()` method, and they must also be able to restore their configuration from a state object via `deserialize()`.

Normally there wouldn't be very much complexity to this idea, but it turns out that with a little bit of generalization we can get a lot of flexibility. A turtle state object can be combined with a turtle in only one meaningful way; that combination results in the turtle restoring its state. So it makes sense for many turtle states to also be turtle commands. We see this in the definition of `BasicTurtle` in section 2.2.

Further, turtles themselves can be seen as state-propagation commands; for instance, if I say this:

`t1.run (t2);`

what I might mean is this:

`t1.deserialize (t2.serialize ());`

Thus turtles themselves can be turtle commands that, when issued to other turtles, synchronize their state. However, all that is explicitly required of a `TurtleState` object is that it supports Java's object-stream serialization.

```
1   package cheloniidae;
2   import java.io.Serializable;
3   public interface TurtleState extends Serializable {}
```

```
1   package cheloniidae;
2   public abstract class ImmutableTurtleState implements TurtleState, TurtleCommand {
3     public TurtleCommand map (final Transformation<TurtleCommand> t) {return this;}
4   }
```

## 3.5 TurtleStack

Sometimes you want to keep track of a series of turtle states and access them in LIFO order. One common case of this is a recursive algorithm that keeps turtle states on a separate stack. The `TurtleStack` class is provided to facilitate this, and provides two accessor commands `push` and `pop` to pass to turtles. So, for example:

```
TurtleStack ts = new TurtleStack ();
t.run (ts.push (), move (100), ts.pop ());
```

will result in `t` being back where it started.

```
1   package cheloniidae;
2
3   import java.util.Map;
4   import java.util.HashMap;
5   import java.util.Stack;
6
7   public class TurtleStack {
8     public class Push extends ImmutableTurtleState implements NonDistributiveTurtleCommand {
9       public TurtleCommand applyTo (final Turtle t) {
10        if (! states.containsKey (t)) states.put (t, new Stack<TurtleState> ());
11        states.get (t).push (t.serialize ());
12        return this;
13      }
14    }
15
16    public class Pop extends ImmutableTurtleState implements NonDistributiveTurtleCommand {
17      public TurtleCommand applyTo (final Turtle t) {
18        t.deserialize (states.get (t).pop ());
19        if (states.get (t).empty ()) states.remove (t);
20        return this;
21      }
22    }
23
24    protected final Map<Turtle, Stack<TurtleState>> states = new HashMap<Turtle, Stack<TurtleState>> ();
25
26    public TurtleCommand push () {return new Push ();}
27    public TurtleCommand pop  () {return new Pop ();}
28  }
```

## 3.6 Standard Commands

Each command that makes an assumption about a turtle also needs to define an interface that lets a turtle indicate its support. Turtles that do support these commands will implement the interfaces for them (see the `SupportsX` interfaces on the `EuclideanTurtle` in section 2.3, for example).

For reasons that will become apparent in chapter 5, most of these commands ultimately inherit from `AtomicCommand`.

### 3.6.1 AtomicCommand

Commands that do not contain other commands are considered atomic. This class provides a basic `map()` method for such commands.

```
1   package cheloniidae.commands;
2
3   import cheloniidae.Transformation;
```

```
4    import cheloniidae.TurtleCommand;
5
6    public abstract class AtomicCommand implements TurtleCommand {
7      public TurtleCommand map (final Transformation<TurtleCommand> t) {return t.transform (this);}
8    }
```

### 3.6.2   UnaryCommand

Most commands take only one value, and the mechanism for storing it is factored into a generic class called UnaryCommand.

```
1    package cheloniidae.commands;
2    public abstract class UnaryCommand<T> extends AtomicCommand {
3      public final T value;
4      public UnaryCommand (final T _value) {value = _value;}
5    }
```

### 3.6.3   Move

This command invokes a turtle's move() method, generally with the semantic that a turtle will move and generate a line from its previous location.

```
1    package cheloniidae.commands;
2    import cheloniidae.Turtle;
3    public final class Move extends UnaryCommand<Double> {
4      public Move (final double value) {super (value);}
5      public Move applyTo (final Turtle t) {
6        if (t instanceof SupportsMove) ((SupportsMove) t).move (value);
7        return this;
8      }
9    }
```

```
1    package cheloniidae.commands;
2    import cheloniidae.Turtle;
3    public interface SupportsMove<T extends Turtle> {
4      public T move (double distance);
5    }
```

### 3.6.4   Jump

This command invokes a turtle's jump() method, generally with the semantic that a turtle will move without generating a line.

```
1    package cheloniidae.commands;
2    import cheloniidae.Turtle;
3    public class Jump extends UnaryCommand<Double> {
4      public Jump (final double value) {super (value);}
5      public Jump applyTo (final Turtle t) {
6        if (t instanceof SupportsJump) ((SupportsJump) t).jump (value);
7        return this;
8      }
9    }
```

```
1    package cheloniidae.commands;
2    import cheloniidae.Turtle;
3    public interface SupportsJump<T extends Turtle> {
4      public T jump (double distance);
5    }
```

### 3.6.5 Turn

Invokes a turtle's `turn()` method.

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public class Turn extends UnaryCommand<Double> {
4    public Turn (final double value) {super (value);}
5    public Turn applyTo (final Turtle t) {
6      if (t instanceof SupportsTurn) ((SupportsTurn) t).turn (value);
7      return this;
8    }
9  }
```

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public interface SupportsTurn<T extends Turtle> {
4    public T turn (double angle);
5  }
```

### 3.6.6 Pitch

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public class Pitch extends UnaryCommand<Double> {
4    public Pitch (final double value) {super (value);}
5    public Pitch applyTo (final Turtle t) {
6      if (t instanceof SupportsPitch) ((SupportsPitch) t).pitch (value);
7      return this;
8    }
9  }
```

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public interface SupportsPitch<T extends Turtle> {
4    public T pitch (double angle);
5  }
```

### 3.6.7 Bank

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public class Bank extends UnaryCommand<Double> {
4    public Bank (final double value) {super (value);}
5    public Bank applyTo (final Turtle t) {
6      if (t instanceof SupportsBank) ((SupportsBank) t).bank (value);
7      return this;
8    }
9  }
```

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public interface SupportsBank<T extends Turtle> {
4    public T bank (double angle);
5  }
```

### 3.6.8 Position

This command sets the turtle's position. In doing so it assumes that the turtle has some way of converting its position from Cartesian coordinates if it resides in a different coordinate space.

For convenience, we also provide the `position()` accessor in the `SupportsPosition` interface – this lets us make the assumption that the turtle can locate itself, which becomes significant for rendering triangles (see section 7).

```
1   package cheloniidae.commands;
2
3   import cheloniidae.Turtle;
4   import cheloniidae.Vector;
5
6   public class Position extends UnaryCommand<Vector> {
7     public Position (final Vector value) {super (value);}
8     public Position applyTo (final Turtle t) {
9       if (t instanceof SupportsPosition) ((SupportsPosition) t).position (value);
10      return this;
11    }
12  }
```

```
1   package cheloniidae.commands;
2
3   import cheloniidae.Turtle;
4   import cheloniidae.Vector;
5
6   public interface SupportsPosition<T extends Turtle> {
7     public Vector position ();
8     public T       position (Vector _position);
9   }
```

### 3.6.9   Direction

Sets the direction of a turtle.

```
1   package cheloniidae.commands;
2
3   import cheloniidae.Turtle;
4   import cheloniidae.Vector;
5
6   public class Direction extends UnaryCommand<Vector> {
7     public Direction (final Vector value) {super (value);}
8     public Direction applyTo (final Turtle t) {
9       if (t instanceof SupportsDirection) ((SupportsDirection) t).direction (value);
10      return this;
11    }
12  }
```

```
1   package cheloniidae.commands;
2
3   import cheloniidae.Turtle;
4   import cheloniidae.Vector;
5
6   public interface SupportsDirection<T extends Turtle> {
7     public Vector direction ();
8     public T       direction (Vector _direction);
9   }
```

### 3.6.10   DirectionComplement

Sets the directionComplement of a turtle (see section 2.5 for an example of a turtle that supports this).

```
1   package cheloniidae.commands;
2
3   import cheloniidae.Turtle;
4   import cheloniidae.Vector;
5
6   public class DirectionComplement extends UnaryCommand<Vector> {
7     public DirectionComplement (final Vector value) {super (value);}
8     public DirectionComplement applyTo (final Turtle t) {
```

```
 9        if (t instanceof SupportsDirectionComplement) ((SupportsDirectionComplement) t).directionComplement (value);
10        return this;
11      }
12  }
```

```
 1  package cheloniidae.commands;
 2
 3  import cheloniidae.Turtle;
 4  import cheloniidae.Vector;
 5
 6  public interface SupportsDirectionComplement<T extends Turtle> {
 7    public Vector directionComplement ();
 8    public T        directionComplement (Vector _directionComplement);
 9  }
```

### 3.6.11   Visible

Changes the visibility of a turtle. Most of the time, a turtle will set its visibility by adding or failing to add a representation of itself to its list of render actions.

```
 1  package cheloniidae.commands;
 2  import cheloniidae.Turtle;
 3  public class Visible extends UnaryCommand<Boolean> {
 4    public Visible (final boolean value) {super (value);}
 5    public Visible applyTo (final Turtle t) {
 6      if (t instanceof SupportsVisible) ((SupportsVisible) t).visible (value);
 7      return this;
 8    }
 9  }
```

```
 1  package cheloniidae.commands;
 2  import cheloniidae.Turtle;
 3  public interface SupportsVisible<T extends Turtle> {
 4    public boolean visible ();
 5    public T        visible (boolean _visible);
 6  }
```

### 3.6.12   BodyColor

Changes the body color of a turtle. This won't impact the color of lines that are rendered; it just determines the color of the turtle's visual representation.

```
 1  package cheloniidae.commands;
 2
 3  import cheloniidae.Turtle;
 4  import java.awt.Color;
 5
 6  public class BodyColor extends UnaryCommand<Color> {
 7    public BodyColor (final Color value) {super (value);}
 8    public BodyColor applyTo (final Turtle t) {
 9      if (t instanceof SupportsBodyColor) ((SupportsBodyColor) t).bodyColor (value);
10      return this;
11    }
12  }
```

```
 1  package cheloniidae.commands;
 2
 3  import cheloniidae.Turtle;
 4  import java.awt.Color;
 5
 6  public interface SupportsBodyColor<T extends Turtle> {
```

```
7    public Color bodyColor ();
8    public T      bodyColor (Color c);
9  }
```

### 3.6.13   LineColor

Sets the color of lines (and presumably other things, including triangles) generated by the turtle.

```
1  package cheloniidae.commands;
2
3  import cheloniidae.Turtle;
4  import java.awt.Color;
5
6  public class LineColor extends UnaryCommand<Color> {
7    public LineColor (final Color value) {super (value);}
8    public LineColor applyTo (final Turtle t) {
9      if (t instanceof SupportsLineColor) ((SupportsLineColor) t).lineColor (value);
10     return this;
11   }
12 }
```

```
1  package cheloniidae.commands;
2
3  import cheloniidae.Turtle;
4  import java.awt.Color;
5
6  public interface SupportsLineColor<T extends Turtle> {
7    public Color lineColor ();
8    public T      lineColor (Color c);
9  }
```

### 3.6.14   LineSize

Sets the thickness of lines generated by the turtle.

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public class LineSize extends UnaryCommand<Double> {
4    public LineSize (final double value) {super (value);}
5    public LineSize applyTo (final Turtle t) {
6      if (t instanceof SupportsLineSize) ((SupportsLineSize) t).lineSize (value);
7      return this;
8    }
9  }
```

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public interface SupportsLineSize<T extends Turtle> {
4    public double lineSize ();
5    public T      lineSize (double size);
6  }
```

### 3.6.15   NullCommand

This command does nothing. It becomes useful much later on, especially for inductive replication (section 6.4). Naturally, it works on all turtles.

```
1  package cheloniidae.commands;
2
3  import cheloniidae.Turtle;
4  import cheloniidae.TurtleCommand;
5
6  public class NullCommand extends AtomicCommand {
7    public NullCommand applyTo (final Turtle t) {return this;}
8  }
```

## 3.7 Utility Commands

Some commands don't directly influence the behavior of a turtle, but rather are helpful for debugging or attribute-marking (see section 4.3).

### 3.7.1 Debug

Causes the turtle to print its `toString()` to standard output. You can provide a string to prepend to the message to identify the source of the debugging command. Because this command doesn't do any special interfacing with the turtle, there is no `SupportsDebug` interface; this command works with all turtles.

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public class Debug extends UnaryCommand<String> {
4    public Debug (final String value) {super (value);}
5    public Debug applyTo (final Turtle t) {
6      System.err.println (value + "␣" + t.toString ());
7      return this;
8    }
9  }
```

### 3.7.2 Pause

It can be very helpful to see a scene materialize slowly. The `Pause` command causes the turtle to take a short break, showing the intermediate results on the window. The only thing a turtle needs to do is support the `window()` method, allowing the command access to its window.

Note that this command is specific to the `TurtleWindow` (section 9.5) viewport class; it is not portable.

```
1  package cheloniidae.commands;
2  import cheloniidae.Turtle;
3  public final class Pause extends UnaryCommand<Long> {
4    public Pause (final long value) {super (value);}
5    public Pause applyTo (final Turtle t) {
6      if (t instanceof SupportsWindow) ((SupportsWindow) t).window ().pause (value);
7      return this;
8    }
9  }
```

```
1  package cheloniidae.commands;
2  import cheloniidae.TurtleWindow;
3  public interface SupportsWindow {
4    public TurtleWindow window ();
5  }
```

### 3.7.3   AddAttribute

Adds an attribute to a turtle. See section 4.3 for more information about this.

```
1  package cheloniidae.commands;
2  import cheloniidae.*;
3  public class AddAttribute extends UnaryCommand<Attribute> {
4    public AddAttribute (final Attribute _value) {super (_value);}
5    public TurtleCommand applyTo (final Turtle t) {
6      t.attributes ().add (value);
7      return this;
8    }
9  }
```

### 3.7.4   RemoveAttribute

Removes an attribute from a turtle.

```
1  package cheloniidae.commands;
2  import cheloniidae.*;
3  public class RemoveAttribute extends UnaryCommand<Attribute> {
4    public RemoveAttribute (final Attribute _value) {super (_value);}
5    public TurtleCommand applyTo (final Turtle t) {
6      t.attributes ().remove (value);
7      return this;
8    }
9  }
```

# Chapter 4

# Predicates

Predicates exist to let you define logical propositions for objects. They could be useful in any number of different ways, including matching against a particular type of turtle command, selecting a turtle from a group, or anything else requiring identification of an object as being either in or out of a set.

## 4.1 Basic Predicates

A predicate at its core is just an object that implements a method called `matches` which takes an object and returns a boolean. Because it extends `Transformable` (see chapter 5), predicates must also implement the `map` method.

```
1  package cheloniidae;
2  public interface Predicate<T> extends Transformable<Predicate<T>> {
3    public boolean matches (T value);
4  }
```

### 4.1.1 AtomicPredicate

The idea of atomic objects comes into play with transformations (see chapter 5). Basically, the idea is that an atomic predicate can't contain others, so the transformation must be applied only to one predicate. (This is not true, for instance, for the `Conjunction` complex predicate defined in section 4.2.2.)

```
1  package cheloniidae.predicates;
2  import cheloniidae.Predicate;
3  import cheloniidae.Transformation;
4  public abstract class AtomicPredicate<T> implements Predicate<T> {
5    public Predicate<T> map (final Transformation<Predicate<T>> t) {return t.transform (this);}
6  }
```

### 4.1.2 The True Predicate

There are cases where it's useful to have a predicate that matches everything. The `True` predicate does just that:

```
1  package cheloniidae.predicates;
2  import cheloniidae.Predicate;
3  public class True<T> extends AtomicPredicate<T> {
4    public boolean matches (final T value) {return true;}
5  }
```

### 4.1.3  HasAttribute

Sometimes it's valuable to know whether an object has an attribute (section 4.3) that matches a given predicate. This predicate goes through all of the attributes of an object to determine whether any of them match:

```
1   package cheloniidae.predicates;
2   import cheloniidae.*;
3   public class HasAttribute<T extends HasAttributes> extends AtomicPredicate<T> {
4     public final Predicate<Attribute> predicate;
5     public HasAttribute (final Predicate<Attribute> _predicate) {predicate = _predicate;}
6
7     public boolean matches (final T attributeContainer) {
8       for (final Attribute a : attributeContainer.attributes ())
9         if (predicate.matches (a)) return true;
10      return false;
11    }
12  }
```

## 4.2  Composition

There are a few different combinators for predicates including negation, conjunction, and type refinement.

### 4.2.1  Negation

The output of any predicate can be negated.

```
1   package cheloniidae.predicates;
2   import cheloniidae.*;
3   public class Negation<T> implements Predicate<T> {
4     public final Predicate<T> predicate;
5     public Negation (final Predicate<T> _predicate) {predicate = _predicate;}
6     public boolean matches (final T value) {return ! predicate.matches (value);}
7
8     public Predicate<T> map (final Transformation<Predicate<T>> t) {
9       final Predicate<T> newPredicate = t.transform (this);
10      if (newPredicate == this) return new Negation (predicate.map (t));
11      else                      return newPredicate;
12    }
13  }
```

### 4.2.2  Conjunction

Predicates can be conjoined, resulting in a new predicate $P(x) = Q(x) \wedge R(x)$.

```
1   package cheloniidae;
2
3   import java.util.Collection;
4   import java.util.LinkedList;
```

```
 5 | import java.util.List;
 6 |
 7 | public class Conjunction<T> implements Predicate<T> {
 8 |   public final List<Predicate<T>> conjuncts = new LinkedList<Predicate<T>> ();
 9 |   public Conjunction (final Predicate<T> ... _conjuncts)
10 |     {for (final Predicate<T> p : _conjuncts) conjuncts.add (p);}
11 |   public Conjunction (final Collection<Predicate<T>> _conjuncts) {conjuncts.addAll (_conjuncts);}
12 |
13 |   public boolean matches (final T value) {
14 |     for (final Predicate<T> p : conjuncts) if (! p.matches (value)) return false;
15 |     return true;
16 |   }
17 |
18 |   public Predicate<T> map (final Transformation<Predicate<T>> t) {
19 |     final Predicate<T> newPredicate = t.transform (this);
20 |     if (newPredicate == this) {
21 |       final List<Predicate<T>> newConjuncts = new LinkedList<Predicate<T>> ();
22 |       for (final Predicate<T> p : conjuncts) newConjuncts.add (p.map (t));
23 |       return new Conjunction<T> (newConjuncts);
24 |     } else return newPredicate;
25 |   }
26 | }
```

### 4.2.3 Disjunction

Just like conjunction, predicates can also be disjoined ($Q \vee R$). Ideally, this and `Conjunction` (see section 4.2.2) should be factored into a single base class that implements a binary reduction on the predicates' outputs. However, the optimized traversal of sub-predicates (that is, the fact that we stop computing once we know the final outcome) is valuable and would require a non-standard encoding of binary reduction.

```
 1 | package cheloniidae;
 2 |
 3 | import java.util.Collection;
 4 | import java.util.LinkedList;
 5 | import java.util.List;
 6 |
 7 | public class Disjunction<T> implements Predicate<T> {
 8 |   public final List<Predicate<T>> disjuncts = new LinkedList<Predicate<T>> ();
 9 |   public Disjunction (final Predicate<T> ... _disjuncts)
10 |     {for (final Predicate<T> p : _disjuncts) disjuncts.add (p);}
11 |   public Disjunction (final Collection<Predicate<T>> _disjuncts) {disjuncts.addAll (_disjuncts);}
12 |
13 |   public boolean matches (final T value) {
14 |     for (final Predicate<T> p : disjuncts) if (p.matches (value)) return true;
15 |     return false;
16 |   }
17 |
18 |   public Predicate<T> map (final Transformation<Predicate<T>> t) {
19 |     final Predicate<T> newPredicate = t.transform (this);
20 |     if (newPredicate == this) {
21 |       final List<Predicate<T>> newDisjuncts = new LinkedList<Predicate<T>> ();
22 |       for (final Predicate<T> p : disjuncts) newDisjuncts.add (p.map (t));
23 |       return new Disjunction<T> (newDisjuncts);
24 |     } else return newPredicate;
25 |   }
26 | }
```

### 4.2.4 Type refinement

When you have a group of generic objects and want to measure a property that only some of those objects possess, you're using an abstraction called type refinement. The idea is that you ignore all

of the objects that don't have the property you're looking for, and run a more specialized predicate on the objects that do support it. So, for example, if you have these classes:

```
class Point                {public double x, y;}
class Point3 extends Point {public double z;}
```

you could write a predicate to find positive *z*-coordinates like this:

```
class ZPositive implements Predicate<Point> {
  boolean matches (final Point p) {
    if (p instanceof Point3) return ((Point3) p).z > 0;
    else                     return false;
  }
}
```

Simpler, though, is to use a type refinement:

```
class ZPositive implements Predicate<Point3> {
  public boolean matches (final Point3 p) {
    return p.z > 0;
  }
}
```

```
Predicate<Point> zplus = new CastableTo<Point, Point3> (Point3.class, new ZPositive ());
```

```
 1 | package cheloniidae.predicates;
 2 |
 3 | import cheloniidae.Predicate;
 4 | import cheloniidae.Transformation;
 5 |
 6 | public class CastableTo<T, U extends T> extends AtomicPredicate<T> {
 7 |   public final Class<U>      type;
 8 |   public final Predicate<U> predicate;
 9 |
10 |   public CastableTo (final Class<U> _type) {type = _type; predicate = null;}
11 |   public CastableTo (final Class<U> _type, final Predicate<U> _predicate)
12 |     {type = _type; predicate = _predicate;}
13 |
14 |   public boolean matches (final T value) {
15 |     return type.isAssignableFrom (value.getClass ()) &&
16 |            (predicate == null || predicate.matches (type.cast (value)));
17 |   }
18 | }
```

## 4.3 Attributes

While you could write a predicate that matched a particular property of an object, it's often easier to put a tag on the object instead. The attribute framework allows you to do just that; all turtles, for example, are required to support attribute lists.

There are a few supporting interface definitions involved. First, the `Attribute` itself:

```
 1 | package cheloniidae;
 2 | public interface Attribute
 3 |         extends Transformable<Attribute>, HasEquivalenceClass<Attribute> {}
```

Of note are the `HasEquivalenceClass` and `Comparable` definitions. `HasEquivalenceClass` indicates that we can construct a predicate to compare two attributes (this is explained further in section 4.4). `Comparable` is required so that attributes can be put into a `TreeSet` object – this is the default set used by `BasicTurtle`. The comparison function can be structurally inconsistent, but must be referentially consistent.

```
1  package cheloniidae;
2  import java.util.Set;
3  public interface HasAttributes {
4    public Set<Attribute> attributes ();
5  }
```

### 4.3.1  AtomicAttribute

Some attributes are atomic in the sense that they have no capacity to store other attributes. For these we can make some assumptions that ease implementation:

```
1  package cheloniidae.attributes;
2  import cheloniidae.Attribute;
3  import cheloniidae.Transformation;
4  public abstract class AtomicAttribute implements Attribute {
5    public Attribute map (final Transformation<Attribute> t) {return t.transform (this);}
6  }
```

Notice the implementation of `compareTo`; the output will be meaningless but referentially consistent. This is acceptable, however.

### 4.3.2  NamedAttribute

One kind of atomic attribute is a named attribute, which is simply a tag with a name on it. This can be used to mark objects as being inside named sets, or to identify them uniquely. This particular attribute is useful when differentiating turtles; you might want different turtles to follow different instructions. Used in conjunction with the `When` command (section 8.3), you can write the equivalent of an `if` statement in pure turtle command language.

```
1  package cheloniidae.attributes;
2
3  import cheloniidae.Attribute;
4  import cheloniidae.Predicate;
5
6  import cheloniidae.predicates.CastableTo;
7  import cheloniidae.predicates.AtomicPredicate;
8
9  public class Named extends AtomicAttribute {
10    public final String name;
11    public Named (final String _name) {name = _name;}
12
13    public Predicate<Attribute> projectivePredicate () {
14      return new CastableTo<Attribute, Named> (Named.class, new AtomicPredicate<Named> () {
15        public boolean matches (final Named value) {return value.name.equals (name);}
16      });
17    }
18  }
```

## 4.4  Equivalence Classes

How should attributes be compared? For instance, suppose you add a named attribute to a turtle:

```
t.attribute (new Named ("foo"));
```

If you then try to ask whether my turtle has that name:

```
t.attributes ().contains (new Named ("foo"))
```

you get `false` because the attribute set uses comparison functions instead of (inferred) structural equality. This is where you need some way to identify whether two instances are really the same thing, and each attribute provides a *projective predicate* to determine this. A projective predicate projects the attribute into an equivalence class where attributes that share semantics are considered equal. So, for example:

```
new Named ("foo").projectivePredicate ().matches (someAttribute)
```

would correctly determine whether `someAttribute` matches `Named ("foo")`.

```
1  package cheloniidae;
2  public interface HasEquivalenceClass<T> {
3    public Predicate<T> projectivePredicate ();
4  }
```

# Chapter 5

# Transformation

Cheloniidae provides a way to take an object and run it through a transformer to produce a variant. Because most things in Cheloniidae are immutable, transformations return a modified copy, leaving the original intact.

The `Transformable` interface defines a single `map` method. This allows the object being transformed to invoke the transformation on its children, if it has any, and reconstruct itself with modified child copies. Here is the signature for the `Transformable` interface:

```
1  package cheloniidae;
2  public interface Transformable<T> {
3    public T map (Transformation<T> transformation);
4  }
```

Correspondingly, there is a `Transformation` interface that defines a `transform` method:

```
1  package cheloniidae;
2  public interface Transformation<T> {
3    public T transform (T input);
4  }
```

Note that the type is fixed on transformations; so, for example, a `Transformation<Turtle>` must both take and return only instances of `Turtle`. This isn't the most general form; transformations could be defined as mapping across types, but for the sake of simplicity I've chosen to fix the type.

The `transform` method is very straightforward; this transformation, for instance, will append `"foo"` to a string:

```
class AppendFoo implements Transformation<String> {
  public String transform (final String input) {return input + "foo";}
}
```

By default, strings don't supply a `map` method, so this transformation would have to be invoked directly on a string: `t.transform (s)`.

## 5.1   The `map` Method

In its simplest form (i.e. for atomic objects), `map` simply returns the result of applying a transformation to `this`. The `AtomicPredicate` (section 4.1.1) and `AtomicAttribute` (section 4.3.1) classes illustrate this case. More complex objects require that `map` iterate through their children as well.

For example, suppose we are writing an arithmetic expression library and we want to write a transformation to reduce an expression tree. The class hierarchy could look like this:

```
abstract class Expression {}

class Constant extends Expression {
  public final double x;
  public Constant (final double _x) {x = _x;}
}

abstract class BinaryExpression extends Expression {
  public final Expression e1;
  public final Expression e2;
  public BinaryExpression (final Expression _e1, final Expression _e2)
    {e1 = _e1; e2 = _e2;}
}

class Sum extends BinaryExpression {
  public Sum (final Expression _e1, final Expression _e2)
    {super (_e1, _e2);}
}

class Product extends BinaryExpression {
  public Product (final Expression _e1, final Expression _e2)
    {super (_e1, _e2);}
}
```

Now we want to make these expressions transformable. We need a `map` method for each one, since the transformation shouldn't have to know anything about how the expressions store sub-expressions. After some factoring, it might be implemented like this.

```
abstract class Expression implements Transformable<Expression> {}

class Constant extends Expression {
  public final double x;
  public Constant (final double _x) {x = _x;}
  public Expression map (final Transformation<Expression> t) {
    // Simple case: Map only over this.
    return t.transform (this);
  }
}

abstract class BinaryExpression extends Expression {
  public final Expression e1;
  public final Expression e2;
  public BinaryExpression (final Expression _e1, final Expression _e2)
    {e1 = _e1; e2 = _e2;}
```

```
  // Should return whatever type this expression is. There are multiple types of
  // binary expressions, so this should construct an instance of whichever child
  // class is currently subclassing.
  public abstract BinaryExpression create (final Expression _e1, final Expression _e2);

  public Expression map (final Transformation<Expression> t) {
    // Complex case: Map over this, and then see whether we got something different.
    // If the transformation modifies this binary expression, then we don't descend,
    // since we wouldn't know what to do with the transformations of e1 and e2.
    // However, if it left this binary expression alone, then we create a new binary
    // expression of this type with the transformations of e1 and e2.

    final Expression firstResult = t.transform (this);
    if (firstResult == this) return create (e1.map (t), e2.map (t));
    else                     return firstResult;
  }
}

class Sum extends BinaryExpression {
  public Sum (final Expression _e1, final Expression _e2)
    {super (_e1, _e2);}
  public BinaryExpression create (final Expression _e1, final Expression _e2)
    {return new Sum (_e1, _e2);}
}

class Product extends BinaryExpression {
  public Product (final Expression _e1, final Expression _e2)
    {super (_e1, _e2);}
  public BinaryExpression create (final Expression _e1, final Expression _e2)
    {return new Product (_e1, _e2);}
}
```

Now let's write the transformation. The strategy is simple: Each time we arrive at a binary expression with two constant children, we perform the operation on those children. There is no post-order traversal in the map function we defined, so we'll have to keep running the transformation on the expression tree until we get just a constant out of it.

```
class Evaluator implements Transformation<Expression> {
  public Expression transform (final Expression e) {
    if (e instanceof BinaryExpression) {
      final BinaryExpression be = (BinaryExpression) e;
      if (be.e1 instanceof Constant && be.e2 instanceof Constant)
        if      (be instanceof Sum)     return new Constant (be.e1.x + be.e2.x);
        else if (be instanceof Product) return new Constant (be.e1.x * be.e2.x);
        else                            throw new Exception ("Unsupported operator");
      else
```

```
        return e;
    } else
        return e;
    }
}
```

To invoke the transformation on an expression:

```
Expression      e  = ...;
final Evaluator ev = new Evaluator ();
while (! e instanceof Constant) e = e.map (ev);
```

So the map method serves two purposes. One is to provide a top-level interface for transforming things (you shouldn't use the transformation's transform method directly because its purpose is to act on only one object), and the other is to give transformations a way to transform the pieces of complex objects in addition to transforming the objects themselves.

## 5.2   Basic Transformations

### 5.2.1   Identity

This transformation just maps objects to themselves.

```
1  package cheloniidae.transformations;
2  import cheloniidae.*;
3  public class Identity<T> implements Transformation<T> {
4    public Identity () {}
5    public T transform (final T x) {return x;}
6  }
```

### 5.2.2   Scale

Slightly more complex is the transformation that finds Move and Jump turtle commands and scales their distances by a given factor. This is primarily useful for recursion (section 8.4). This transformation makes sense only for turtle commands.

```
1  package cheloniidae.transformations;
2
3  import cheloniidae.*;
4  import cheloniidae.commands.*;
5
6  public class Scale implements Transformation<TurtleCommand> {
7    public final double  factor;
8    public final boolean scaleLineSizes;
9
10   public Scale (final double _factor)                          {this (_factor, false);}
11   public Scale (final double _factor, final boolean _scaleLineSizes) {factor = _factor; scaleLineSizes = _scaleLineSizes;}
12
13   public TurtleCommand transform (final TurtleCommand c) {
14     if                      (c instanceof Move)     return new Move     (((Move) c).value    * factor);
15     else if                 (c instanceof Jump)     return new Jump     (((Jump) c).value    * factor);
16     else if (scaleLineSizes && c instanceof LineSize) return new LineSize (((LineSize) c).value * factor);
17     else                                           return c;
18   }
19 }
```

## 5.3 Composition

There are a couple of wrappers for transformations. The first one is `Compose`, which takes a bunch of transformations and applies each of them to an input. *Note!* This composition runs backwards from a mathematical perspective; that is, $(f_1 \circ f_2 \circ \cdots \circ f_n)(x)$ evaluates to $f_n(\cdots(f_2(f_1(x)))\cdots)$, not $f_1(f_2(\cdots(f_n(x))\cdots))$.

```
1   package cheloniidae.transformations;
2   import cheloniidae.*;
3   public class Compose<T extends Transformable<T>> implements Transformation<T> {
4     public final Transformation<T>[] transformations;
5     public Compose (final Transformation<T> ... _transformations) {transformations = _transformations;}
6
7     public T transform (final T c) {
8       T immediate = c;
9       for (final Transformation<T> t : transformations) immediate = immediate.map (t);
10      return immediate;
11    }
12  }
```

## 5.4 Predicated Transformations

Another wrapper for transformations is one which conditionally applies a given transform. It uses a predicate to decide whether each object it runs across should be transformed; if it should, then the transformation is applied; otherwise, it applies the identity transformation leaving the object unchanged.

```
1   package cheloniidae.transformations;
2   import cheloniidae.*;
3   public class PredicatedTransformation<T extends Transformable<T>> implements Transformation<T> {
4     public final Predicate<T>        predicate;
5     public final Transformation<T> transformation;
6
7     public PredicatedTransformation (final Predicate<T> _predicate, final Transformation<T> _transformation)
8       {predicate = _predicate; transformation = _transformation;}
9
10    public T transform (final T input) {
11      if (predicate.matches (input)) return input.map (transformation);
12      else                           return input;
13    }
14  }
```

# Chapter 6

# Replication

Similar to the concept of transformation (chapter 5) is replication. The goal of replication is to create a transformed clone of a mutable object such as a turtle. Since the replication framework was written with turtles in mind, I'll sacrifice generality for simplicity by making the assumption that turtles are the only things we would want to replicate this way.

## 6.1 Motivation

### 6.1.1 Creating a tree

So, why would you want to replicate a turtle? Well, one reason is to draw some sort of recursive structure. For example, you could draw a tree like this (pseudocode):

```
define stack as a stack of turtle states
define tree (turtle, recursion limit) as
  move some distance
  if recursion limit > 0 then
    repeat 2
      push turtle state onto stack
      do some random setup
      tree (turtle, recursion limit - 1)
      pop turtle state from stack
```

However, this implementation is a bit clunky and the imperative structure obscures the meaning of what's going on. What we'd like to say is this:

```
define tree (turtle, recursion limit) as
  move some distance
  if recursion limit > 0 then
    repeat 2
      with a copy of turtle
        do some random setup
        tree (the copy of turtle, recursion limit - 1)
```

So at each point the turtle splits off into two new turtles, and each one performs the next iteration. This design pattern is called splitting, and is described in section 6.5. For a more advanced use, see section 7.3.

### 6.1.2 Making a circle of things

Let's suppose you have a square initially:

```
define square (turtle, distance) as
  repeat 4 turtle move distance turn 90
```

and you decide you want a ring of them. This isn't too hard to do, since drawing a square leaves the turtle where it started, but perhaps you want to change the square to a half-circle at some point in the future and don't want to worry about breaking your code. So you insulate calls to `square` by using a temporary turtle state:

```
define protected (shape, turtle, <stuff>) as
  define state as turtle serialize in
    shape (turtle, <stuff>)
    turtle deserialize state
```

So now you make a circle of squares like this:

```
define circle of (shape, turtle, <stuff>) as
  repeat 10
    turtle jump 100 turn 36
    protected (shape, turtle, <stuff>)
```

And this works fairly well. However, the design pattern can be abstracted away just a bit. The pattern here is called *inductive replication* because each turtle copy is computed from the previous one by an inductive step. The replicator for this is described in section 6.4.

## 6.2 Replicable

Unfortunately, replication isn't quite as simple as it could be. We don't want a complete copy of a replicated turtle, since the copy shouldn't inherit the lines of the original. Also, we want an object of the same type as the original, so we need to provide a per-subclass method to create an instance of that subclass. In this case, it's called `create`.

Because replication requires clonability, I'm using the Java notion of a `clone` method. The `Turtle` interface (section 2.1) inherits from Java's `Cloneable` interface to indicate that the `clone` method does what it should. The default implementation provided by the `Replicable` interface should always work (the semantics are correct), but it may be necessary to make modifications in subclasses.

```
1  package cheloniidae;
2  import cheloniidae.commands.*;
3  public abstract class Replicable<T extends Turtle> implements Turtle<T>, TurtleCommand {
4    public abstract T create ();
5
```

```
 6      public T clone () {
 7        final T result = this.create ();
 8        this.applyTo (result);
 9        return result;
10      }
11    }
```

## 6.3  Replicator

The `Replicable` class isn't strictly required for replication, but it provides the correct semantics for `clone`.  Replicators should work on any object that implements `Cloneable`, though they're restricted to turtles for the purposes of Cheloniidae.[1]  The idea is that a replicator takes one turtle and returns a collection of new turtles with some initial configuration.

```
1    package cheloniidae;
2    public interface Replicator extends NonDistributiveTurtleCommand {
3      public TurtleGroup<Turtle> replicate (Turtle base);
4    }
```

## 6.4  Inductive Replication

The simplest way to get a bunch of turtles is follow a system like this:

```
define copy (turtle, configuration command, n) as
  if n > 0 then
    a copy of turtle configured by the configuration command, along with
      copy (the copy of turtle, configuration command, n - 1)
  else
    an empty turtle group
```

so that each new copy is a transformed version of the previous copy.  The Cheloniidae class to implement this is:

```
 1    package cheloniidae.replicators;
 2
 3    import cheloniidae.*;
 4    import cheloniidae.commands.*;
 5
 6    public class InductiveReplicator<T extends Turtle> implements Replicator {
 7      public final TurtleCommand step;
 8      public final int          copies;
 9      public final Sequence      actions;
10      public InductiveReplicator (final int _copies, final TurtleCommand _step, final TurtleCommand ... _actions)
11        {step = _step; copies = _copies; actions = new Sequence (_actions);}
12
13      public TurtleGroup<Turtle> replicate (final Turtle turtle) {
14        final TurtleGroup<Turtle> result   = new TurtleGroup<Turtle> ();
15        Turtle                    previous = turtle;
16        for (int i = 0; i < copies; ++i) result.turtles ().add (previous = previous.clone ().run (step));
17        return result;
18      }
19
20      public TurtleCommand applyTo (final Turtle turtle) {
```

---

[1]This may change in the future.

```
21      final TurtleGroup<Turtle> copies = this.replicate (turtle);
22      turtle.window ().add (copies);
23      copies.run (actions);
24      return this;
25    }
26
27    public TurtleCommand map (final Transformation<TurtleCommand> t) {
28      final TurtleCommand newCommand = t.transform (this);
29      if (newCommand == this) return new InductiveReplicator (copies, step.map (t), actions.map (t));
30      else                    return newCommand;
31    }
32  }
```

## 6.5  Splitting

Often you'll just want a single copy of a turtle instead of a series of transformed ones. An easy
way to do this is to use the inductive replicator to obtain a single copy:

```
t.run (new InductiveReplicator (1, new NullCommand (), <commands>));
```

Doing this will run <commmands> on a copy of t. The `NullCommand` command is a no-op; it does
nothing, and is useful for situations like this. `CoreCommands` (section 11.4) provides a convenient
shorthand for split operations, allowing this code to be used instead:

```
t.run (copy (<commands>));
```

# Chapter 7

# Triangles

Unlike most regular turtles, turtles in Cheloniidae can produce triangles. This isn't as straight-forward as producing lines because it usually involves a split of some sort, but there are several different models that simplify the process.

## 7.1  Using the `Triangle` **Turtle Command**

This is the simplest way to produce a triangle. The command takes two other turtle commands and works like this:

1. Take the current turtle position – this is the first vertex of the triangle.

2. Copy the turtle and run the first turtle command on the copy. The location of the copy is the second vertex of the triangle.

3. Copy the original and run the second turtle command on the copy. The location of this copy is the third vertex of the triangle.

Alternately, you can pass in two other turtles and the command will generate a triangle connecting them.

Internally there is some finagling to make this work; a new preloaded turtle is created (see below), and this new turtle contains the triangle. Also, the original turtle must have a location that makes sense in Cartesian space (indicated by implementing the `SupportsPosition` interface).

```java
package cheloniidae;

import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

public class PreloadedTurtle<T extends PreloadedTurtle> extends BasicTurtle<T> {
  public final Set<RenderAction> actions;
  public PreloadedTurtle (final Set<RenderAction> _actions) {actions = _actions;}

  public SortedSet<RenderAction> actions (final Viewport v) {
    final SortedSet<RenderAction> result = new TreeSet<RenderAction> (new PerspectiveComparator (v));
    for (final RenderAction a : actions) if (v.shouldCancel ()) break;
                                         else                    result.add (a);
```

```
15        return result;
16      }
17
18      public TurtleState serialize   ()                    {return null;}
19      public T            deserialize (final TurtleState t) {return (T) this;}
20
21      public T              create  ()                  {return (T) this;}
22      public T              clone   ()                  {return (T) this;}
23      public TurtleCommand applyTo (final Turtle t) {return this;}
24    }
```

```
1   package cheloniidae.commands;
2
3   import java.awt.Color;
4   import java.util.Set;
5   import java.util.HashSet;
6
7   import cheloniidae.*;
8
9   public class Triangle implements TurtleCommand {
10    public final SupportsPosition p1;
11    public final SupportsPosition p2;
12
13    public final TurtleCommand c1;
14    public final TurtleCommand c2;
15
16    public Triangle (final SupportsPosition _p1, final SupportsPosition _p2) {
17      p1 = _p1; p2 = _p2;
18      c1 = c2 = null;
19    }
20
21    public Triangle (final TurtleCommand _c1, final TurtleCommand _c2) {
22      p1 = p2 = null;
23      c1 = _c1; c2 = _c2;
24    }
25
26    public TurtleCommand applyTo (final Turtle t) {
27      if (t instanceof SupportsPosition && t instanceof SupportsLineColor) {
28        final SupportsPosition p3       = (SupportsPosition) t;
29        final Color            color    = ((SupportsLineColor) t).lineColor ();
30        final Set<RenderAction> actions = new HashSet<RenderAction> ();
31
32        if (p1 != null) actions.add (new CartesianTriangle (p1.position (), p2.position (), p3.position (), color));
33        else {
34          // Construct positions by creating temporary clones of the turtle.
35          final Turtle clone1 = t.clone ().run (c1);
36          final Turtle clone2 = t.clone ().run (c2);
37
38          actions.add (new CartesianTriangle (((SupportsPosition) clone1).position (),
39                                              ((SupportsPosition) clone2).position (),
40                                              p3.position (), color));
41        }
42
43        t.window ().add (new PreloadedTurtle (actions));
44      }
45
46      return this;
47    }
48
49    public TurtleCommand map (final Transformation<TurtleCommand> t) {
50      final TurtleCommand newCommand = t.transform (this);
51      if (newCommand == this && c1 != null) return new Triangle (c1.map (t), c2.map (t));
52      else                                  return newCommand;
53    }
54  }
```

## 7.2 PathwiseTriangleConnector

The pathwise triangle connector connects a group of turtles together and constructs quads between them. It is a low-level interface (a higher-level interface more suited to standard use is the BandSplitReplicator in section 7.3). The idea is fairly straightforward: Given *n* turtles, track their positions and draw virtual lines between them. Each time emit() is called, draw a new set of virtual lines between each pair of turtles and construct quads to connect the last set of virtual lines with the new set.

This interface is one of the most counterintuitive in Cheloniidae. For a more intuitive presentation, take a look at section 7.3 and the example using it in the bandsplit.java file.

The TriangleEmitter interface provides turtle commands to initialize and emit triangles.

```
1   package cheloniidae;
2
3   import cheloniidae.commands.AtomicCommand;
4
5   public interface TriangleEmitter<T extends Turtle> {
6     public static class Start extends AtomicCommand implements NonDistributiveTurtleCommand {
7       public Start () {}
8       public TurtleCommand applyTo (final Turtle t) {
9         if (t instanceof TriangleEmitter) ((TriangleEmitter) t).start ();
10        return this;
11      }
12    }
13
14    public static class Emit extends AtomicCommand implements NonDistributiveTurtleCommand {
15      public Emit () {}
16      public TurtleCommand applyTo (final Turtle t) {
17        if (t instanceof TriangleEmitter) ((TriangleEmitter) t).emit ();
18        return this;
19      }
20    }
21
22    public T start ();
23    public T emit ();
24  }
```

A quick note about pathwise triangle connectors: There is some seriously gnarly stuff that has to happen to get sequences and repeats to work properly. All of the logic to do this is in the Parallelize transformation implemented in section 8.5, and this is ultimately why the SerialTurtleCommandComposition interface exists.

```
1   package cheloniidae;
2
3   import cheloniidae.transformations.Parallelize;
4
5   import java.util.ArrayList;
6   import java.util.Collection;
7   import java.util.LinkedList;
8   import java.util.List;
9   import java.util.SortedSet;
10  import java.util.TreeSet;
11
12  public class PathwiseTriangleConnector<T extends EuclideanTurtle>
13        extends TurtleGroup<T>
14    implements TriangleEmitter<PathwiseTriangleConnector<T>> {
15
16    public final List<Vector>           points    = new ArrayList<Vector> ();
17    public final List<CartesianTriangle> triangles = new LinkedList<CartesianTriangle> ();
18
19    public PathwiseTriangleConnector (final T ...        _turtles) {super (_turtles); start ();}
20    public PathwiseTriangleConnector (final Collection<T> _turtles) {super (_turtles); start ();}
21
```

```java
22    public SortedSet<RenderAction> actions (final Viewport v) {
23      final SortedSet<RenderAction> result = super.actions (v);
24      for (final CartesianTriangle t : triangles) if (v.shouldCancel ()) break;
25                                                  else                     result.add (t);
26      return result;
27    }
28
29    public PathwiseTriangleConnector<T> create () {
30      final PathwiseTriangleConnector<T> result = new PathwiseTriangleConnector<T> ();
31      for (final T t : turtles) result.add ((T) t.clone ());
32      result.start ();
33      return result;
34    }
35
36    public PathwiseTriangleConnector<T> add (final T turtle) {
37      super.add (turtle);
38      points.add (turtle.position ().clone ());
39      return this;
40    }
41
42    public PathwiseTriangleConnector start () {
43      // Keep track of all of the locations of all of the turtles.
44      points.clear ();
45      for (final EuclideanTurtle t : turtles) points.add (t.position ().clone ());
46      return this;
47    }
48
49    public PathwiseTriangleConnector emit () {
50      // Emit triangles connecting all of the turtles. For n turtles, there will be O(n^2) triangles.
51      for (int i = 0; i < turtles.size () - 1; ++i)
52        for (int j = i + 1; j < turtles.size (); ++j) {
53          final EuclideanTurtle t1  = turtles.get (i);
54          final EuclideanTurtle t2  = turtles.get (j);
55          final Vector         tv1 = t1.position ();
56          final Vector         tv2 = t2.position ();
57          final Vector         sv1 = points.get (i);
58          final Vector         sv2 = points.get (j);
59
60          // If the triangle's area is below a certain amount, then we don't add the triangle.
61          // This is to prevent degenerate triangles from slowing the scene render operations,
62          // which might occur if the user moved one turtle but not the other.
63          if (tv1.clone ().subtract (sv1).cross (sv2.clone ().subtract (sv1)).lengthSquared () > 1.0e-10)
64            triangles.add (new CartesianTriangle (tv1, sv1, sv2, t1.color ()));
65
66          if (tv1.clone ().subtract (tv2).cross (sv2.clone ().subtract (tv2)).lengthSquared () > 1.0e-10)
67            triangles.add (new CartesianTriangle (tv1, tv2, sv2, t2.color ()));
68        }
69
70      return start ();
71    }
72
73    public PathwiseTriangleConnector<T> run (final TurtleCommand c) {
74      super.run (c.map (new Parallelize ()));
75      return this;
76    }
77  }
```

## 7.3  BandSplitReplicator

This class hides a lot of the practical difficulties that arise when using
PathwiseTriangleConnectors. Instead of taking two pre-existing turtles, it operates on a
single turtle, performs a split, initializes the copy, and then runs a series of commands. This
enables you to use it inline, as in the example file bandsplit.java.

```
1   package cheloniidae.replicators;
2
3   import cheloniidae.*;
4   import cheloniidae.commands.*;
5
6   public class BandSplitReplicator implements Replicator, TurtleCommand {
7     public final TurtleCommand firstTurtlePrimer;
8     public final Sequence       actions;
9
10    public BandSplitReplicator (final TurtleCommand _firstTurtlePrimer, final TurtleCommand ... _actions)
11      {firstTurtlePrimer = _firstTurtlePrimer; actions = new Sequence (_actions);}
12
13    public TurtleGroup<Turtle> replicate (final Turtle turtle) {
14      if (turtle instanceof EuclideanTurtle)
15        // We have to wrap the pathwise connector in a new turtle group because of type annotations.
16        // Specifying a generic with <EuclideanTurtle> makes it unassignable to anything specified
17        // with <Turtle> because Java doesn't know whether the type will be in a contravariant position.
18        // (In this case it isn't, but Java can't prove that.) The best we can do is wrap it.
19        return new TurtleGroup<Turtle> (
20              new PathwiseTriangleConnector<EuclideanTurtle> ()
21                .add ((EuclideanTurtle) turtle.clone ().run (firstTurtlePrimer))
22                .add ((EuclideanTurtle) turtle.clone ()));
23      else return null;
24    }
25
26    public TurtleCommand applyTo (final Turtle turtle) {
27      final TurtleGroup<Turtle> copies = this.replicate (turtle);
28      turtle.window ().add (copies);        // Very important! Otherwise the copies will have no way of producing visible lines.
29      copies.run (actions);
30      return this;
31    }
32
33    public TurtleCommand map (final Transformation<TurtleCommand> t) {
34      final TurtleCommand newCommand = t.transform (this);
35      if (newCommand == this) return new BandSplitReplicator (firstTurtlePrimer.map (t), actions.map (t));
36      else                    return newCommand;
37    }
38  }
```

45

# Chapter 8

# Commands 2

Most of the commands we've seen so far are straightforward; they generate some form of output immediately and don't really change the behavior of things very much. The commands in this chapter are geared more towards transformation of other commands, doing things like simulating control flow and leveraging transformation (chapter 5) to modify pieces of the command structure.

## 8.1   Sequencing

Just as most imperative programming languages execute one statement after another, Cheloniidae provides a construct that executes one turtle command after another. It has shown up in some classes already, often to bundle a sequence of commands into a single `TurtleCommand` object.

Notice the `SerialTurtleCommandComposition` interface. This is a flag indicating that sequences string together a bunch of logical operations. Section 8.5 explains why this is important.

```java
1  package cheloniidae.commands;
2
3  import cheloniidae.SerialTurtleCommandComposition;
4  import cheloniidae.Turtle;
5  import cheloniidae.TurtleCommand;
6  import cheloniidae.Transformation;
7
8  public class Sequence implements TurtleCommand, SerialTurtleCommandComposition {
9    protected TurtleCommand intersperse = null;
10
11   public final TurtleCommand[] commands;
12   public Sequence (final TurtleCommand ... _commands) {commands = _commands;}
13
14   public Sequence applyTo (final Turtle t) {if (intersperse != null) t.run (intersperse);
15                                             for (TurtleCommand c : commands) {
16                                               t.run (c);
17                                               if (intersperse != null) t.run (intersperse);
18                                             }
19                                             return this;}
20
21   public Sequence interspersing (final TurtleCommand c) {
22     intersperse = intersperse == null || c == null ? c : new Sequence (c, intersperse);
23     return this;
24   }
25
26   public TurtleCommand map (final Transformation<TurtleCommand> t) {
27     final TurtleCommand newCommand = t.transform (this);
```

46

```
28        if (newCommand == this) {
29          final TurtleCommand[] cs = new TurtleCommand[commands.length];
30          for (int i = 0; i < commands.length; ++i) cs[i] = commands[i].map (t);
31          return new Sequence (cs).interspersing (intersperse != null ? intersperse.map (t) : null);
32        } else return newCommand;
33      }
34 }
```

## 8.2    Repetition

The Repeat command lets you reduce a series of repeated commands:

```
move (100), turn (90),
move (100), turn (90)
```

to a single repetition invocation:

```
repeat (2, move (100), turn (90))
```

It doesn't expand the commands in memory. Rather, when run on a turtle it loops over its body a number of times; this saves memory for large repeat counts.

```
1   package cheloniidae.commands;
2
3   import cheloniidae.SerialTurtleCommandComposition;
4   import cheloniidae.Turtle;
5   import cheloniidae.TurtleCommand;
6   import cheloniidae.Transformation;
7
8   public class Repeat implements TurtleCommand, SerialTurtleCommandComposition {
9     public final int       repetitions;
10    public final Sequence body;
11
12    public Repeat (final int _repetitions, final TurtleCommand ... _body)
13      {repetitions = _repetitions; body = new Sequence (_body);}
14
15    public Repeat applyTo (final Turtle t) {
16      for (int i = 0; i < repetitions; ++i) t.run (body);
17      return this;
18    }
19
20    public TurtleCommand map (final Transformation<TurtleCommand> t) {
21      final TurtleCommand newCommand = t.transform (this);
22      if (newCommand == this) return new Repeat (repetitions, body.map (t));
23      else                    return newCommand;
24    }
25 }
```

## 8.3    Conditional Execution

Commands can be predicated on turtles. This can be useful if you have a group of turtles and you want only some of them to follow a command path. The way to do this is to use the When command on a turtle predicate:

```
group.run (new When (someTurtlePredicate, <commands>));
```

```
1  package cheloniidae.commands;
2
3  import cheloniidae.Predicate;
4  import cheloniidae.Transformation;
5  import cheloniidae.Turtle;
6  import cheloniidae.TurtleCommand;
7
8  public class When implements TurtleCommand {
9    public final Predicate<Turtle> predicate;
10   public final TurtleCommand      action;
11
12   public When (final Predicate<Turtle> _predicate, final TurtleCommand ... _actions)
13     {predicate = _predicate; action = new Sequence (_actions);}
14
15   public TurtleCommand applyTo (final Turtle t) {
16     if (predicate.matches (t)) t.run (action);
17     return this;
18   }
19
20   public TurtleCommand map (final Transformation<TurtleCommand> t) {
21     final TurtleCommand newCommand = t.transform (this);
22     if (newCommand == this) return new When (predicate, action.map (t));
23     else                    return newCommand;
24   }
25 }
```

## 8.4   Recursion

This is where things get complicated. Turtle commands weren't really designed with recursive expansion in mind, so a few hacks are required to implement recursion. What we are ultimately trying to achieve is analogous to recursion in a functional language:

```
factorial (0) = 1
factorial (n) = n * factorial (n - 1)
```

which yields:

```
factorial (5) = 5 * factorial (4)
              = 5 * 4 * factorial (3)
              = 5 * 4 * 3 * factorial (2)
              = 5 * 4 * 3 * 2 * factorial (1)
              = 5 * 4 * 3 * 2 * 1 * factorial (0)
              = 5 * 4 * 3 * 2 * 1 * 1
```

In Cheloniidae, recursion looks a little bit different. Instead of being a computational construct, it ends up looking more like a rewrite system and provides an interface that lets you customize how the terms are rewritten. The recursive construct of a square would look like this:

```
new RecursiveExpansion ("square", new Move (100), new Turn (90),
                                   new RecursiveExpansion.Marker (
                                     "square", 3,
                                     new Identity<TurtleCommand> (), new NullCommand ()))
```

The `Marker` instance tells the `RecursiveExpansion` that an expansion should be inserted there. Each marker stores the name of the code to be expanded, enabling multiple-level recursion. The 3 indicates that the marker should survive for three more expansions before turning into a base case. This should yield a total of four move-turn sequences.

Each marker also has the ability to transform its expansion in some way; this lets you do things traditionally done by passing parameters into the recursive invocation. In this case, we use the identity transformation because we don't want to change either the move or the turn; however, we could just as easily make the other sides of the square decrease in length exponentially:

```
new RecursiveExpansion.Marker (
  "square", 3,
  new Scale (0.5), new NullCommand ())
```

The final parameter is the value that the expansion should take when its expansion count reaches zero. So here is the rewriting that happens when a turtle executes our square definition:

```
move, turn,
  marker ("square", 3, identity, null)

move, turn, sequence (move, turn,
  marker ("square", 2, identity, null)).map (identity)

move, turn, sequence (move, turn, sequence (move, turn,
  marker ("square", 1, identity, null)).map (identity))

move, turn, sequence (move, turn, sequence (move, turn, sequence (move, turn, null)))
```

Since the null command does nothing, the turtle executes four moves and four turns, producing a square as expected.

```java
1   package cheloniidae.commands;
2
3   import cheloniidae.*;
4   import cheloniidae.transformations.*;
5
6   public class RecursiveExpansion implements TurtleCommand {
7     public static class Marker implements TurtleCommand {
8       public final String                    name;
9       public final int                       remainingExpansions;
10      public final Transformation<TurtleCommand> inductiveTransformation;
11      public final TurtleCommand             base;
12      public       TurtleCommand             inductiveExpansion = null;
13
14      public Marker (final String _name, final int _remainingExpansions,
15                 final Transformation<TurtleCommand> _inductiveTransformation, final TurtleCommand _base)
16        {name                    = _name;                        remainingExpansions = _remainingExpansions;
17         inductiveTransformation = _inductiveTransformation; base                    = _base;}
18
19      public Marker inductiveExpansion (final TurtleCommand _inductiveExpansion) {
20        inductiveExpansion = _inductiveExpansion;
21        return this;
22      }
23
24      public TurtleCommand applyTo (final Turtle t) {
25        if (remainingExpansions <= 0) t.run (base);
26        else {
```

```
27        final TurtleCommand transformedExpansion =
28          inductiveExpansion.map (inductiveTransformation).map (new DecrementTransformation (name));
29        t.run (transformedExpansion.map (new ExpansionPopulator (name, transformedExpansion)));
30      }
31      return this;
32    }
33
34    public TurtleCommand map (final Transformation<TurtleCommand> t) {
35      final TurtleCommand newCommand = t.transform (this);
36      if (newCommand == this)
37        return new Marker (name, remainingExpansions,
38                            inductiveTransformation, base.map (t)).inductiveExpansion (inductiveExpansion);
39      else
40        return newCommand;
41    }
42  }
43
44  public static class DecrementTransformation implements Transformation<TurtleCommand> {
45    public final String name;
46    public DecrementTransformation (final String _name) {name = _name;}
47
48    public TurtleCommand transform (final TurtleCommand c) {
49      if (c instanceof Marker && ((Marker) c).name.equals (name)) {
50        final Marker cprime = (Marker) c;
51        return new Marker (cprime.name, cprime.remainingExpansions - 1, cprime.inductiveTransformation, cprime.base);
52      } else return c;
53    }
54  }
55
56  public static class ExpansionPopulator implements Transformation<TurtleCommand> {
57    public final String         name;
58    public final TurtleCommand expansion;
59    public ExpansionPopulator (final String _name, final TurtleCommand _expansion)
60      {name = _name; expansion = _expansion;}
61
62    public TurtleCommand transform (final TurtleCommand c) {
63      if (c instanceof Marker && ((Marker) c).name.equals (name)) ((Marker) c).inductiveExpansion (expansion);
64      return c;
65    }
66  }
67
68  public final String         name;
69  public final TurtleCommand body;
70
71  public RecursiveExpansion (final String _name, final TurtleCommand _body) {name = _name; body = _body;}
72
73  public TurtleCommand applyTo (final Turtle t) {
74    t.run (body.map (new ExpansionPopulator (name, body)));
75    return this;
76  }
77
78  public TurtleCommand map (final Transformation<TurtleCommand> t) {
79    final TurtleCommand newCommand = t.transform (this);
80    if (newCommand == this) return new RecursiveExpansion (name, body.map (t));
81    else                      return newCommand;
82  }
83 }
```

## 8.5   Parallelize

This isn't a turtle command *per se*; rather, it transforms turtle commands to work better with the
PathwiseTriangleConnector (). The fundamental problem that needs addressing is
subtle and arises from the way TurtleGroup handles sequences and repetition.

To see the problem, imagine you have two people, each holding the end of a chalkline, standing

next to each other and facing the same direction. All of the commands you give to these people go through a proxy that issues your command first to the first person, and then to the second one. The commands you give to the proxy are "move" and "create a line," so you can get a series of lines fairly easily:

```
proxy.run (move (10))
     .run (createALine ())
     .run (move (10))
     ...
```

The difficulty arises when you specify any complex command such as a repeat or sequence. In this case, the proxy receives the command as usual:

```
proxy.run (repeat (2, move (10), createALine ()))
```

and proceeds to translate it like this:

```
person1.run (repeat (2, move (10), createALine ()));
person2.run (repeat (2, move (10), createALine ()));
```

With this arrangement, all of the movement will be done by one person before the other person starts moving, resulting in a bunch of diagonal lines instead of the desired parallel lines. The way to fix it is to rewrite the sequences (including Repeat) as NonDistributiveTurtleCommands that apply themselves in parallel to all of the people/turtles managed by the proxy.

```
1  package cheloniidae;
2  public interface SerialTurtleCommandComposition extends TurtleCommand {}
```

```
1   package cheloniidae.transformations;
2
3   import cheloniidae.*;
4   import cheloniidae.predicates.*;
5
6   public class Parallelize implements Transformation<TurtleCommand> {
7     // Sometimes the order of execution of turtle commands matters. One example of this is when they're
8     // operating in parallel constructing triangles using a pathwise triangle connector. In this case,
9     // each turtle should move one step and then the emitter should be run. However, normally serial
10    // turtle command compositions distribute across turtle groups such as connectors. To fix this, we
11    // wrap those commands inside of non-distributive proxies.
12    //
13    // There's a slight catch here. When we grab a serial composition, we need to transform the
14    // original. Well, we can't just run it back through this transformation, since that would result in
15    // an infinite loop. Instead, we create a new child instance of this transformation that is primed
16    // to ignore the thing we're wrapping. This is why we have a private constructor.
17
18    private final TurtleCommand commandBeingWrapped;
19
20    public  Parallelize ()                                    {commandBeingWrapped = null;}
21    private Parallelize (final TurtleCommand _commandBeingWrapped) {commandBeingWrapped = _commandBeingWrapped;}
22
23    public TurtleCommand transform (final TurtleCommand c) {
24      if (c instanceof SerialTurtleCommandComposition && c != commandBeingWrapped)
25        return new NonDistributiveProxy (c.map (new Parallelize (c)));
26      else return c;
27    }
28  }
```

# Part II

# Implementation

# Chapter 9

# Displaying Scenes

These components, in descending order of dependency, make up Cheloniidae's rendering framework:

1. `TurtleWindow` – section 9.5

2. `Viewport` – section 9.6

3. `Renderable` – section 9.1

4. `RenderAction` – section 9.2; implementations include `CartesianLine` (section 10.2) and `CartesianTriangle` (section 10.3)

5. `HasPerspectiveProjection` – section 9.3

6. `PerspectiveComparator` – section 9.4

## 9.1 Renderable

Objects that provide visual output should implement the `Renderable` interface. Turtles do this because they draw stuff to the screen.

```
1  package cheloniidae;
2  import java.util.SortedSet;
3  public interface Renderable {
4    public SortedSet<RenderAction> actions (Viewport v);
5  }
```

## 9.2 RenderAction

Turtles display their output by providing a set of `RenderAction`s. A render action must support two basic operations:

1. It must be able to draw itself to a Java graphics context.

2. It must provide its distance from the camera.

The second requirement exists so that objects can be rendered from back-to-front in the viewspace (see section 9.3).

The two render actions currently defined are CartesianLine (section 10.2) and CartesianTriangle (section 10.3). Some turtles also define render actions to display themselves; these can be found in the definition of those turtle classes in chapter 2. (All of these render actions, by convention, are nested classes called View.)

```
1  package cheloniidae;
2  public interface RenderAction extends HasPerspectiveProjection {
3    public void render (Viewport viewport);
4  }
```

## 9.3    HasPerspectiveProjection

Depth-sorting ensures that objects are rendered in a reasonable order. Particularly, we want objects that are far away from the camera to be rendered first, followed by objects that are close to the camera. This is important because it allows closer objects to occlude more distant ones, which is what happens in real life. The viewport provides methods to transform points into camera-space (see section 9.6).

```
1  package cheloniidae;
2  public interface HasPerspectiveProjection {
3    public double depth (Viewport v);
4  }
```

## 9.4    PerspectiveComparator

This is the wrapper necessary to construct a SortedSet of render actions, sorted in reverse by depth.

```
1   package cheloniidae;
2
3   import java.util.Comparator;
4
5   public final class PerspectiveComparator implements Comparator<HasPerspectiveProjection> {
6     public final Viewport v;
7     public PerspectiveComparator (final Viewport _v) {v = _v;}
8
9     public final int compare (final HasPerspectiveProjection p1, final HasPerspectiveProjection p2) {
10      // A couple of notes about this. First, notice that we're comparing in reverse. This is intentional.
11      // We want to sort such that the objects farthest away from the point of view are at the beginning,
12      // since they should be rendered first. Second, notice that we never return equality. This is because
13      // a TreeSet is a proper set, not just a bag of elements. Items that compare equal belong to the same
14      // equivalence class, so they would be considered duplicates. Thus we make sure that the comparison
15      // never returns equality. (This was the cause of a weird bug.)
16
17      return p2.depth (v) < p1.depth (v) ? -1 : 1;
18    }
19  }
```

## 9.5  TurtleWindow

This is where all of the graphical stuff happens.  The structure is not too complex, either.  Each
RenderAction provided by a turtle is given the graphics context of the window, and is allowed to
draw on it arbitrarily. So the turtle window is really just a shell that delegates the actual drawing
to the render actions specified by its turtles.

```java
1   package cheloniidae;
2
3   import java.awt.image.BufferedImage;
4   import java.awt.event.*;
5
6   import java.awt.Color;
7   import java.awt.Frame;
8   import java.awt.Graphics;
9   import java.awt.Graphics2D;
10  import java.awt.RenderingHints;
11
12  import java.util.SortedSet;
13  import java.util.Iterator;
14  import java.util.Random;
15
16  public class TurtleWindow<T extends Turtle> extends Frame implements Viewport {
17    public class RenderOperation extends Thread {
18      protected final Viewport viewport;
19      public RenderOperation (final Viewport v) {viewport = v;}
20
21      public void run () {
22        final Graphics2D c = context ();
23        c.setColor (getBackground ());
24        c.fillRect (0, 0, getWidth (), getHeight ());
25
26        final SortedSet<RenderAction> actions       = turtles.actions (viewport);
27        final Iterator<RenderAction>  actionIterator = actions.iterator ();
28
29        int objectsDrawnSoFar = 0;
30        while (! shouldCancel && actionIterator.hasNext ()) {
31          actionIterator.next ().render (viewport);
32          if (++objectsDrawnSoFar % drawingRefreshInterval == 0) repaint ();
33        }
34
35        if (! shouldCancel) {
36          if (shouldShowObjectCount) setTitle ("Cheloniidae (" + objectsDrawnSoFar + " objects)");
37          repaint ();
38        }
39      }
40    }
41
42    public class IntermediateRenderOperation extends Thread {
43      public void run () {
44        final Graphics2D c         = context ();
45        final int        pointColor = (~ getBackground ().getRGB ()) & 0xffffff;
46        c.setColor (getBackground ());
47        c.fillRect (0, 0, getWidth (), getHeight ());
48
49        for (int i = 0; ! shouldCancel && i < intermediatePointCloud.length; ++i)
50          if (intermediatePointCloud[i] != null) {
51            final Vector tp = transformPoint (intermediatePointCloud[i]);
52            if (tp.z > 0.0) {
53              final Vector pp = projectPoint (tp);
54              if (pp.x >= 0.0 && pp.x < getWidth () && pp.y >= 0.0 && pp.y < getHeight ())
55                offscreen.setRGB ((int) pp.x, (int) pp.y, pointColor);
56            }
57          }
58
59        repaint ();
60      }
```

```
 61  |    }
 62  |
 63  |    protected int          drawingRefreshInterval   = 1000;
 64  |    protected boolean       shouldShowObjectCount    = true;
 65  |
 66  |    protected BufferedImage offscreen                = null;
 67  |    protected Graphics2D    cachedContext            = null;
 68  |    protected TurtleGroup<T> turtles                 = new TurtleGroup<T> ();
 69  |
 70  |    protected Vector        virtualPOV               = new Vector (0, 0, -500.0);
 71  |    protected Vector        virtualPOVUp             = new Vector (0, 1, 0);
 72  |    protected Vector        virtualPOVForward        = new Vector (0, 0, 1);
 73  |
 74  |    protected Vector        minimumExtent            = new Vector (0, 0, 0);
 75  |    protected Vector        maximumExtent            = new Vector (0, 0, 0);
 76  |
 77  |    protected Vector[]      intermediatePointCloud   = new Vector[5000];
 78  |
 79  |    protected int           mouseDownX               = 0;
 80  |    protected int           mouseDownY               = 0;
 81  |    protected boolean       mouseDown                = false;
 82  |    protected Thread        graphicsRequestRunner    = null;
 83  |    protected boolean       graphicsRequestCancelFlag = false;
 84  |    protected boolean       fisheye3D                = false;
 85  |    protected long          lastChange               = 0;
 86  |    protected boolean       shouldCancel             = false;
 87  |
 88  |    public TurtleWindow () {initialize ();}
 89  |
 90  |    public int        drawingRefreshInterval     () {return drawingRefreshInterval;}
 91  |    public TurtleWindow drawingRefreshInterval     (final int _drawingRefreshInterval)
 92  |      {drawingRefreshInterval = _drawingRefreshInterval; return this;}
 93  |
 94  |    public boolean    shouldShowObjectCount      () {return shouldShowObjectCount;}
 95  |    public TurtleWindow shouldShowObjectCount      (final boolean _shouldShowObjectCount)
 96  |      {shouldShowObjectCount = _shouldShowObjectCount; return this;}
 97  |
 98  |    public int        intermediatePointCloudSize () {return intermediatePointCloud.length;}
 99  |    public TurtleWindow intermediatePointCloudSize (final int _intermediatePointCloudSize)
100  |      {intermediatePointCloud = new Vector[_intermediatePointCloudSize]; return this;}
101  |
102  |    protected void handleResize () {
103  |      offscreen     = new BufferedImage (super.getWidth (), super.getHeight (), BufferedImage.TYPE_3BYTE_BGR);
104  |      cachedContext = null;
105  |      enqueueGraphicsRefreshRequest (new RenderOperation (this));
106  |    }
107  |
108  |    protected void initialize () {
109  |      final TurtleWindow t = this;
110  |
111  |      super.addWindowListener (new WindowListener ()
112  |        {public void windowClosing     (final WindowEvent e) {dispose ();}
113  |         public void windowActivated    (final WindowEvent e) {}
114  |         public void windowClosed       (final WindowEvent e) {}
115  |         public void windowDeactivated  (final WindowEvent e) {}
116  |         public void windowDeiconified  (final WindowEvent e) {}
117  |         public void windowIconified    (final WindowEvent e) {}
118  |         public void windowOpened       (final WindowEvent e) {}});
119  |
120  |      super.addComponentListener (new ComponentListener ()
121  |        {public void componentResized (final ComponentEvent e) {handleResize ();}
122  |         public void componentMoved    (final ComponentEvent e) {}
123  |         public void componentHidden   (final ComponentEvent e) {}
124  |         public void componentShown    (final ComponentEvent e) {}});
125  |
126  |      super.addMouseListener (new MouseListener () {public void mouseReleased (final MouseEvent e)
127  |                                                    {mouseDown = false;
128  |                                                     lastChange = System.currentTimeMillis ();
129  |                                                     enqueueGraphicsRefreshRequest (new RenderOperation (t));}
```

```
130
131                                                   public void mousePressed  (final MouseEvent e)
132                                                     {mouseDown = true;
133                                                      mouseDownX = e.getX ();
134                                                      mouseDownY = e.getY ();}
135
136                                                   public void mouseClicked  (final MouseEvent e) {}
137                                                   public void mouseEntered  (final MouseEvent e) {}
138                                                   public void mouseExited   (final MouseEvent e) {}});
139
140      super.addMouseMotionListener (new MouseMotionListener () {
141        public void mouseDragged (final MouseEvent e) {
142          if (mouseDown) {
143            final Vector virtualPOVRight = virtualPOVForward.cross (virtualPOVUp);
144            final Vector center          = new Vector (maximumExtent).multiply (0.5).
145                                                          addScaled (minimumExtent, 0.5);
146            final double factor          = e.isControlDown () ? 0.1 : 1.0;
147
148            // A normal drag translates the view locally.
149            if (! e.isShiftDown ())
150              virtualPOV.addScaled (virtualPOVUp,     factor * (mouseDownY - e.getY ())).
151                         addScaled (virtualPOVRight, factor * (e.getX () - mouseDownX));
152            else {
153              final double pitchAngle = (e.getY () - mouseDownY) * factor;
154              final double turnAngle  = (e.getX () - mouseDownX) * factor;
155
156              virtualPOV = virtualPOV.subtract (center).rotatedAbout (virtualPOVUp,     turnAngle).
157                                                        rotatedAbout (virtualPOVRight, pitchAngle).add (center);
158
159              virtualPOVForward = virtualPOVForward.rotatedAbout (virtualPOVUp,     turnAngle).
160                                                    rotatedAbout (virtualPOVRight, pitchAngle).normalize ();
161              virtualPOVUp      = virtualPOVUp.rotatedAbout      (virtualPOVRight, pitchAngle).normalize ();
162            }
163
164            lastChange = System.currentTimeMillis ();
165
166            mouseDownX = e.getX ();
167            mouseDownY = e.getY ();
168            enqueueGraphicsRefreshRequest (new IntermediateRenderOperation ());
169          }
170        }
171        public void mouseMoved (final MouseEvent e) {}
172      });
173
174      super.addMouseWheelListener (new MouseWheelListener  () {
175        public void mouseWheelMoved (final MouseWheelEvent e) {
176          virtualPOV.addScaled (virtualPOVForward, (e.isControlDown () ? -1 : -10) * e.getWheelRotation ());
177          lastChange = System.currentTimeMillis ();
178          enqueueGraphicsRefreshRequest (new IntermediateRenderOperation ());
179        }});
180
181      super.setSize       (600, 372);
182      super.setTitle      ("Cheloniidae");
183      super.setVisible    (true);
184      super.setBackground (Color.WHITE);
185
186      handleResize ();
187    }
188
189    public void update (final Graphics g) {paint (g);}
190    public void paint  (final Graphics g) {g.drawImage (offscreen, 0, 0, null);}
191
192    public TurtleWindow add (final T t) {turtles.turtles ().add (t); t.window (this); return this;}
193
194    public void enqueueGraphicsRefreshRequest (final Thread t) {
195      if (offscreen != null) {
196        if (graphicsRequestRunner != null && graphicsRequestRunner.isAlive ()) {
197          cancel ();
198          try {graphicsRequestRunner.join ();}
```

```
199          catch (final InterruptedException e) {}
200        }
201
202        shouldCancel = false;
203        (graphicsRequestRunner = t).start ();
204      }
205    }
206
207    public TurtleWindow pause (final long milliseconds) {
208      enqueueGraphicsRefreshRequest (new RenderOperation (this));
209      try {Thread.sleep (milliseconds);}
210      catch (final InterruptedException e) {}
211      return this;
212    }
213
214    public TurtleWindow cancel () {shouldCancel = true; return this;}
215
216    public boolean shouldCancel () {return shouldCancel;}
217    public long    lastChange  () {return lastChange;}
218    public double  scaleFactor () {return getHeight ();}
219
220    public TurtleWindow representativePoint (final Vector v) {
221      final int index = Math.abs (new Random ().nextInt ()) % intermediatePointCloud.length;
222      if (intermediatePointCloud[index] == null || new Random ().nextDouble () > 0.97) intermediatePointCloud[index] = v;
223      minimumExtent.componentwiseMinimum (v);
224      maximumExtent.componentwiseMaximum (v);
225      return this;
226    }
227
228    public Vector transformPoint (final Vector v)
229      {return new Vector (v).subtract (virtualPOV).toCoordinateSpace (
230        virtualPOVUp.cross (virtualPOVForward), virtualPOVUp, virtualPOVForward);}
231
232    public Vector projectPoint (final Vector v)
233      {return (fisheye3D ? new Vector (v).normalize () :
234                          new Vector (v).divide (v.z)).multiply (
235                            super.getHeight ()).add (new Vector (super.getWidth () >> 1, super.getHeight () >> 1, 0));}
236
237    public Graphics2D context () {
238      if (cachedContext != null) return cachedContext;
239      else {
240        final Graphics2D    g  = (Graphics2D) offscreen.getGraphics ();
241        final RenderingHints rh = g.getRenderingHints ();
242        rh.put (rh.KEY_ANTIALIASING, rh.VALUE_ANTIALIAS_ON);
243        g.setRenderingHints (rh);
244        return g;
245      }
246    }
247 }
```

## 9.6 Viewport

A subset of the TurtleWindow is used to interface with render actions; this is called a Viewport. It's an abstraction to allow generalization of the user interface, one of the options I'm considering taking advantage of with Cheloniidae 4.

Most of these methods should be self-explanatory (see section 10.4 for an explanation of lastChange), but one deserves some attention. The representativePoint method appears to do nothing; it just returns the viewport. However, it is provided to update the viewport's state. It is intended to let render actions specify points that they consider to be significant; this enables a faster point-cloud display to be used for intermediate renders instead of rendering a subset of actions from the turtles. The TurtleWindow class (see section 9.5) also uses these points to compute

a bounding box for the scene, enabling the camera to rotate around the middle of the scene instead of just around the origin.

```java
1   package cheloniidae;
2
3   import java.awt.Graphics2D;
4
5   public interface Viewport {
6     public boolean    shouldCancel        ();
7     public Graphics2D context             ();
8     public double     scaleFactor         ();
9     public Vector     transformPoint      (Vector v);
10    public Vector     projectPoint        (Vector v);
11    public Viewport   representativePoint (Vector v);
12    public long       lastChange          ();
13  }
```

# Chapter 10

# Render Actions

## 10.1 Vector

A vector is Cheloniidae's mechanism for representing a three-dimensional point. It includes a bunch of helpful vector mathematics methods, most of which modify the point somehow and then return it. A notable exception to this rule is `rotatedAbout`, which creates a new vector. In general, methods which are phrased in the present tense will modify the vector, and methods phrased in past tense will return a new one.

```java
package cheloniidae;

import java.io.Serializable;

public final class Vector implements Serializable, Cloneable, Comparable<Vector> {
  public double x = 0.0;
  public double y = 0.0;
  public double z = 0.0;

  public Vector ()                                              {}
  public Vector (final Vector existing)                         {this (existing.x, existing.y, existing.z);}
  public Vector (final double all)                              {this (all, all, all);}
  public Vector (final double x, final double y, final double z) {this.assign (x, y, z);}

  public Vector clone     ()                      {return new Vector (this);}
  public int    compareTo (final Vector other) {return Double.compare (lengthSquared (), other.lengthSquared ());}
  public String toString  ()                      {return "<" + Double.toString (x) + ", " +
                                                          Double.toString (y) + ", " +
                                                          Double.toString (z) + ">";}

  public Vector assign (final Vector v)                                          {return this.assign (v.x, v.y, v.z);}
  public Vector assign (final double nx, final double ny, final double nz) {x = nx; y = ny; z = nz; return this;}
  public Vector center ()                                                        {return this.assign (0, 0, 0);}

  public Vector addScaled (final Vector other, final double factor) {x += other.x * factor;
                                                                     y += other.y * factor;
                                                                     z += other.z * factor;
                                                                     return this;}

  public Vector add      (final Vector other)  {x += other.x; y += other.y; z += other.z; return this;}
  public Vector subtract (final Vector other)  {x -= other.x; y -= other.y; z -= other.z; return this;}
  public Vector multiply (final Vector other)  {x *= other.x; y *= other.y; z *= other.z; return this;}
  public Vector multiply (final double factor) {x *= factor;  y *= factor;  z *= factor;  return this;}
  public Vector divide   (final Vector other)  {x /= other.x; y /= other.y; z /= other.z; return this;}
  public Vector divide   (final double factor) {x /= factor;  y /= factor;  z /= factor;  return this;}
```

```java
36
37     public Vector componentwiseMinimum (final Vector other) {return this.assign (other.x < this.x ? other.x : this.x,
38                                                                                  other.y < this.y ? other.y : this.y,
39                                                                                  other.z < this.z ? other.z : this.z);}
40     public Vector componentwiseMaximum (final Vector other) {return this.assign (other.x > this.x ? other.x : this.x,
41                                                                                  other.y > this.y ? other.y : this.y,
42                                                                                  other.z > this.z ? other.z : this.z);}
43
44     public double dot (final Vector other) {return x * other.x + y * other.y + z * other.z;}
45     public double length ()                 {return Math.sqrt (x*x + y*y + z*z);}
46     public double lengthSquared ()          {return x*x + y*y + z*z;}
47
48     public Vector proj (final Vector base) {return new Vector (base).multiply (this.dot (base) / base.lengthSquared ());}
49     public Vector orth (final Vector base) {return this.proj (base).multiply (-1.0).add (this);}
50
51     public Vector normalize () {return divide (length ());}
52
53     public double distanceFrom (final Vector other) {return Math.sqrt ((x - other.x) * (x - other.x) +
54                                                                        (y - other.y) * (y - other.y) +
55                                                                        (z - other.z) * (z - other.z));}
56
57     public Vector cross (final Vector other) {return new Vector (y * other.z - z * other.y,
58                                                                  z * other.x - x * other.z,
59                                                                  x * other.y - y * other.x);}
60
61     public Vector toCoordinateSpace (final Vector v1, final Vector v2, final Vector v3)
62       {return new Vector (this.dot (v1) / v1.length (), this.dot (v2) / v2.length (), this.dot (v3) / v3.length ());}
63
64     public Vector fromCoordinateSpace (final Vector v1, final Vector v2, final Vector v3)
65       {return new Vector (v1).multiply (x).addScaled (v2, y).addScaled (v3, z);}
66
67     public Vector rotatedAbout (final Vector v, final double degrees) {
68       final Vector b1 = new Vector (v).normalize ();
69       final Vector b2 = this.orth (b1).normalize ();
70       final Vector b3 = b1.cross (b2);                    // The cross product of orthogonal unit vectors is a unit vector.
71       final double l  = this.orth (b1).length ();
72
73       return this.proj (b1).addScaled (b2, Math.cos (degrees * Math.PI / 180.0) * l).
74                             addScaled (b3, Math.sin (degrees * Math.PI / 180.0) * l);
75     }
76   }
```

## 10.2   CartesianLine

This object represents a Euclidean line whose endpoints have Cartesian coordinates. It knows how to approximate the incidence angle relative to the viewpoint (this can be inferred by using the viewport to transform the endpoints and midpoint), and uses this to shade the line according to the calculations for thickness-based shading (see section 10.5).

```java
1   package cheloniidae;
2
3   import java.awt.BasicStroke;
4   import java.awt.Color;
5   import java.awt.Graphics2D;
6
7   public class CartesianLine extends ViewportCaching implements HasPerspectiveProjection, RenderAction {
8     public final Vector v1;
9     public final Vector v2;
10    public final double width;
11    public final Color  color;
12    public final Vector midpoint;
13
14    public CartesianLine (final Vector _v1, final Vector _v2, final double _width, final Color _color) {
15      v1        = new Vector (_v1);
```

```
16        v2       = new Vector (_v2);
17        width    = _width;
18        color    = _color;
19        midpoint = v1.clone ().multiply (0.5).addScaled (v2, 0.5);
20     }
21
22     public double computeDepth (final Viewport v) {return v.transformPoint (midpoint).length ();}
23
24     public void render (final Viewport v) {
25       final Vector tv1 = v.transformPoint (v1);
26       final Vector tv2 = v.transformPoint (v2);
27
28       // If either point is behind the POV, then solve for z = 1.0. If both points are behind,
29       // then the line does not get rendered.
30       if (tv1.z > 0 || tv2.z > 0) {
31         v.representativePoint (midpoint).representativePoint (v1).representativePoint (v2);
32
33         if (tv1.z <= 0) {
34           final double factor = (1.0 - tv1.z) / (tv2.z - tv1.z);
35           tv1.multiply (1.0 - factor).addScaled (tv2, factor);
36         } else if (tv2.z <= 0) {
37           final double factor = (1.0 - tv2.z) / (tv1.z - tv2.z);
38           tv2.multiply (1.0 - factor).addScaled (tv1, factor);
39         }
40
41         final double     thickness = 2.0 * v.scaleFactor () / (tv1.z + tv2.z);
42         final Vector     pv1       = v.projectPoint (tv1);
43         final Vector     pv2       = v.projectPoint (tv2);
44         final Graphics2D g         = v.context ();
45
46         final Vector transformedMidpoint = v.transformPoint (midpoint);
47         final Vector transformedNormal   = tv1.clone ().subtract (tv2);
48         final Color  newColor            = IncidentAngleComputation.adjustForThickness (color,
49                                              IncidentAngleComputation.cylindricalThickness (
50                                                transformedNormal, transformedMidpoint));
51
52         g.setStroke (new BasicStroke ((float) Math.abs (thickness * width)));
53         g.setColor  (newColor);
54         g.drawLine  ((int) pv1.x, (int) pv1.y, (int) pv2.x, (int) pv2.y);
55       }
56     }
57 }
```

## 10.3 CartesianTriangle

This represents a Euclidean triangle whose vertices are represented by Cartesian coordinates. Like the CartesianLine (section 10.2), it is shaded according to the perceived incidence angle.

```
1  package cheloniidae;
2
3  import java.awt.Color;
4  import java.awt.Graphics2D;
5  import java.awt.Polygon;
6
7  public class CartesianTriangle extends ViewportCaching implements HasPerspectiveProjection, RenderAction {
8    public final Vector v1;
9    public final Vector v2;
10   public final Vector v3;
11   public final Color  color;
12   public final Vector midpoint;
13
14   public CartesianTriangle (final Vector _v1, final Vector _v2, final Vector _v3, final Color _color) {
15     v1       = new Vector (_v1);
16     v2       = new Vector (_v2);
17     v3       = new Vector (_v3);
```

```
18      color   = _color;
19      midpoint = v1.clone ().multiply (1.0 / 3.0).addScaled (v2, 1.0 / 3.0).addScaled (v3, 1.0 / 3.0);
20    }
21
22    public double computeDepth (final Viewport v) {return v.transformPoint (midpoint).length ();}
23
24    public void render (final Viewport v) {
25      final Vector tv1 = v.transformPoint (v1);
26      final Vector tv2 = v.transformPoint (v2);
27      final Vector tv3 = v.transformPoint (v3);
28
29      v.representativePoint (midpoint).representativePoint (v1).representativePoint (v2).representativePoint (v3);
30
31      // Render the triangle only when all three points are in front of the camera.
32      // I'm being lazy here -- there is a way to solve for the points' projections to
33      // z = 1 like we did for Cartesian lines, but the number of cases is much larger.
34      if (tv1.z > 0 && tv2.z > 0 && tv3.z > 0) {
35        final Vector     pv1 = v.projectPoint (tv1);
36        final Vector     pv2 = v.projectPoint (tv2);
37        final Vector     pv3 = v.projectPoint (tv3);
38        final Graphics2D g   = v.context ();
39
40        final int[] xs = new int[] {(int) pv1.x, (int) pv2.x, (int) pv3.x};
41        final int[] ys = new int[] {(int) pv1.y, (int) pv2.y, (int) pv3.y};
42
43        // Render only real triangles. If they become degenerate when projected into 2D,
44        // then we ignore them.
45        if (! (xs[0] == xs[1] && ys[0] == ys[1] ||
46               xs[1] == xs[2] && ys[1] == ys[2] ||
47               xs[0] == xs[2] && ys[0] == ys[2])) {
48          final Vector transformedMidpoint    = v.transformPoint (midpoint);
49          final Vector transformedNormal      = tv1.clone ().subtract (tv2).cross (tv3.clone ().subtract (tv2));
50          final Color  newColor               = IncidentAngleComputation.adjustForThickness (color,
51                                                  IncidentAngleComputation.planarThickness (
52                                                    transformedNormal, transformedMidpoint));
53          g.setColor (newColor);
54          g.fill    (new Polygon (xs, ys, 3));
55        }
56      }
57    }
58 }
```

## 10.4   Viewport Caching

This is a hack for optimization purposes. When the render actions are depth-sorted, $O(n \log n)$ comparisons will be done between render actions' computed depths.[1] If done naïvely, this would result in $O(n \log n)$ distance computations for $n$ objects. However, the distance of an object is unlikely to change during a depth-sort operation, so the naïve algorithm will be a factor of $\log n$ slower than it could be.

This is where viewport caching comes in. The viewport maintains a timestamp of the last state change (only state changes that would affect the depth-sorting are considered), and objects that inherit from the ViewportCaching class have the optimization that they recompute their distance only if the viewport's timestamp is after their last recomputation.

```
1  package cheloniidae;
2
3  public abstract class ViewportCaching {
4    private Viewport cachedViewport   = null;
5    private long     cachedLastChange = 0;
```

---

[1]See the HasPerspectiveProjection interface in section 9.3.

```
 6    private double   cachedDepth      = 0.0;
 7
 8    public abstract double computeDepth (Viewport v);
 9
10    public double depth (final Viewport v) {
11      final double result = (cachedViewport == v && v.lastChange () == cachedLastChange) ?
12                          cachedDepth : (cachedDepth = computeDepth (v));
13      cachedViewport  = v;
14      cachedLastChange = v.lastChange ();
15      return result;
16    }
17  }
```

## 10.5   Incidence Angles

Incident angle computation is about finding out how much light passes through a translucent material based on the angle. Here's why the angle matters. Suppose you have a one-inch thick sheet of smoked glass:
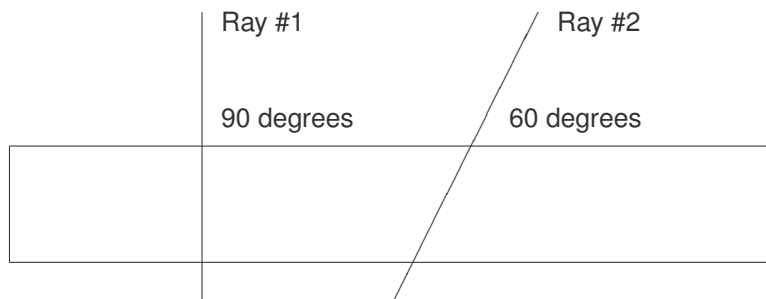


Looking through it vertically (ray #1) results in the attenuation from one inch of smoked glass. However, if we look through it from a different angle (ray #2), the ray will pass through more glass. In each case, the amount of glass traveled through is the secant of the ray's incident angle, so for ray 1 it would be $\sec 90° = 1$, and for ray 2 it would be $\sec 60° = \frac{2}{\sqrt{3}}$. Conveniently, the secant of the angle is simply the reciprocal of the dot product of the ray's direction and the surface normal of the glass. (That is, $\sec \theta = (\vec{r} \cdot \vec{N})^{-1}$, where $\theta$ is the incidence angle, $\vec{r}$ is the ray's direction, and $\vec{N}$ is the surface normal vector.)

Slightly more difficult is figuring out how much light is left after traveling through $x$ amount of material. Doing this requires a separable differential equation:

$$\frac{dL}{dx} = -oL$$

where $L(x)$ is the amount of light after traveling through $x$ amount of glass, and $o$ is the opacity of the glass. Solving yields:

$$L(x) = Ce^{-ox}$$

Solving for $C$ is simple if we make the assumption that ray #1 should perceive $o$ opacity from one inch of glass. So we can solve with that constraint:

$$1 - o = Ce^{-o} \quad \Rightarrow \quad C = \frac{1 - o}{e^{-o}}$$

```java
package cheloniidae;

import java.awt.Color;
import static java.lang.Math.*;

public abstract class IncidentAngleComputation {
  public static final double lightTransmission (final double thickness, final double opacity) {
    final double c = (1.0 - opacity) / exp (-opacity);
    return c * exp (-thickness * opacity);
  }

  public static final Color adjustForThickness (final Color original, final double thickness) {
    final double originalOpacity  = original.getAlpha () / 255.0;
    final double perceivedOpacity = 1.0 - lightTransmission (thickness, originalOpacity);
    final double clippedOpacity   = perceivedOpacity > 1.0 ? 1.0 :
                                    perceivedOpacity < 0.0 ? 0.0 : perceivedOpacity;

    return new Color (original.getRed (), original.getGreen (), original.getBlue (), (int) (clippedOpacity * 255.0));
  }

  public static final double planarThickness (final Vector surfaceNormal, final Vector v) {
    // This is a simplified way to compute the secant of the scalar angle between the two vectors.
    return surfaceNormal.length () * v.length () / abs (surfaceNormal.dot (v));
  }

  public static final double cylindricalThickness (final Vector axis, final Vector v) {
    final double cosineTheta = axis.dot (v) / (axis.length () * v.length ());
    final double sineTheta   = sqrt (1.0 - cosineTheta * cosineTheta);
    return 1.0 / sineTheta;
  }
}
```

# Chapter 11

# Frames

These classes exist to simplify turtle scene construction.

## 11.1   Resources

Over time, I would like to build up a standard library of resources for Cheloniidae. One of the more straightforward of these is a text rendering interface, included here. Given a string it will return a sequence of commands that render each letter. Right now it supports lowercase letters and spaces, though it can easily be extended.

Because the resulting sequence is split on letters and not on lines, you can do fun things with it; check out the `text.java` example file for some ideas.

```java
 1   package cheloniidae.resources;
 2
 3   import cheloniidae.*;
 4   import cheloniidae.commands.*;
 5
 6   import static cheloniidae.frames.CoreCommands.*;
 7
 8   import java.util.*;
 9
10   public class BasicTextRenderer {
11     public static final double diagonal = Math.sqrt (2);
12
13     public TurtleCommand line     (final double length) {return move (length);}
14     public TurtleCommand lateral (final double right)  {return sequence (turn (90), jump (right), turn (-90));}
15     public TurtleCommand right    ()                    {return sequence (turn ( 45), move (diagonal), turn ( 45));}
16     public TurtleCommand left     ()                    {return sequence (turn (-45), move (diagonal), turn (-45));}
17
18     public TurtleCommand wind     (final int ... xs) {
19       // 0 = change direction. By default, the direction is clockwise, or right.
20       final List<TurtleCommand> cs   = new LinkedList<TurtleCommand> ();
21       TurtleCommand              last = null;
22       boolean                    right = true;
23
24       for (final int x : xs) if (x == 0) right ^= true;
25                              else if (x > 0) {
26                                cs.add (line (x));
27                                cs.add (last = right ? right () : left ());
28                              } else {
29                                cs.add (jump (-x));
30                                cs.add (last = right ? right () : left ());
```

```java
31                                        }
32
33        if (last != null) cs.remove (last);
34        return seq (cs.toArray (new TurtleCommand[0]));
35      }
36
37      public TurtleCommand topLeft      () {return new Sequence (new Jump (9), new Turn (180));}
38      public TurtleCommand topRight     () {return new Sequence (new Jump (9), new Turn (90), new Jump (6), new Turn (90));}
39      public TurtleCommand bottomLeft   () {return new NullCommand ();}
40      public TurtleCommand bottomRight  () {return new Sequence (new Turn (90), new Jump (6), new Turn (-90));}
41
42      public TurtleCommand seq (final TurtleCommand ... commands) {return new Sequence (commands);}
43
44      public TurtleCommand drawCharacter (final char c) {
45        switch (c) {
46          case 'a': return seq (wind (-5, 4, 4, 4, 1, 5), lateral (3), jump (1), turn (-90));
47          case 'b': return seq (topLeft (), wind (0, 8, 4, 4, 5), lateral (-6), turn (180), jump (7), turn (-90));
48          case 'c': return seq (lateral (6), jump (5), left (), wind (0, 4, 4, 4), left (), jump (-1), lateral (1));
49          case 'd': return seq (topRight (), wind (8, 4, 4, 5), lateral (6), jump (1), turn (-90));
50          case 'e': return seq (jump (3), turn (90), wind (0, 5, 1, 4, 4, 4), left (), jump (-1), lateral (1));
51          case 'f': return seq (wind (8, 4), right (), jump (3), turn (90), jump (1), line (5), lateral (-5), turn (90),
52                                line (7));
53          case 'g': return seq (jump (-2), turn (180), left (), wind (0, 4, 7, 4, 4, 5), jump (1), turn (-90));
54          case 'h': return seq (topLeft (), line (9), jump (-5), turn (180), right (), wind (4, 5), turn (180),
55                                lateral (1));
56          case 'i': return seq (turn (90), line (6), jump (-3), turn (-90), wind (0, 5, 1), lateral (-6), turn (90),
57                                lateral (5));
58          case 'j': return seq (jump (-3), turn (90), wind (0, 2, 8), turn (-90), line (2), lateral (-6), turn (90),
59                                lateral (5));
60          case 'k': return seq (line (6), jump (-3), turn (45), line (3 * diagonal), jump (-3 * diagonal), turn (90),
61                                line (3 * diagonal), turn (-135), lateral (3));
62          case 'l': return seq (turn (90), line (6), jump (-3), turn (-90), wind (0, 8, 1), turn (90), jump (-9),
63                                lateral (6));
64          case 'm': return seq (wind (5, 1, 5), turn (180), wind (5, 1, 5), turn (180), lateral (1));
65          case 'n': return seq (wind (5, 4, 5), turn (180), lateral (1));
66          case 'o': return seq (jump (1), wind (4, 4, 4, 4), right (), jump (-1), lateral (7));
67          case 'p': return seq (jump (-3), wind (8, 4, 4, 5), turn (90), lateral (7));
68          case 'q': return seq (jump (-3), lateral (6), wind (0, 8, 4, 4, 5), turn (-90), lateral (1));
69          case 'r': return seq (wind (5, 4), turn (-90), jump (-6), lateral (2));
70          case 's': return seq (turn (90), wind (0, 5, 1, 0, 4, 1, 5), turn (-90), jump (-6), lateral (1));
71          case 't': return seq (lateral (3), line (6), turn (90), jump (-3), line (6), lateral (6),
72                                turn (-90), lateral (1));
73          case 'u': return seq (jump (6), turn (180), wind (0, 5, 4, 5), jump (-6), lateral (1));
74          case 'v': return seq (jump (6), turn (180), line (3), turn (-45), line (3 * diagonal), turn (-90),
75                                line (3 * diagonal), turn (-45), line (3), jump (-6), lateral (1));
76          case 'w': return seq (jump (6), turn (180), wind (0, 5, 1, 5), turn (180), wind (0, 5, 1, 5), jump (-6),
77                                lateral (1));
78          case 'x': return seq (turn (45), line (6 * diagonal), turn (-135), jump (6), turn (-135), line (6 * diagonal),
79                                turn (-135), lateral (1));
80          case 'y': return seq (jump (6), turn (180), wind (0, 5, 5), turn (-90), jump (6), turn (180), wind (8, 5),
81                                turn (90), jump (3), lateral (7));
82          case 'z': return seq (jump (6), turn (90), line (6), turn (135), line (6 * diagonal), turn (-135), line (6),
83                                turn (-90), lateral (1));
84
85          case '␣': return lateral (7);
86          default:  return new NullCommand ();
87        }
88      }
89
90      public Sequence drawText (final String text) {
91        final List<TurtleCommand> cs = new LinkedList<TurtleCommand> ();
92        for (int i = 0; i < text.length (); ++i) cs.add (drawCharacter (text.charAt (i)));
93        return new Sequence (cs.toArray (new TurtleCommand[0]));
94      }
95  }
```

## 11.2  SingleTurtleScene

This automates the logic behind creating a scene with one turtle. There is some parameterization to allow a turtle group to be used as the turtle, but it turns out to be somewhat difficult to use.

```
1   package cheloniidae.frames;
2
3   import cheloniidae.*;
4
5   public abstract class SingleTurtleScene<T extends Turtle> {
6     protected final TurtleWindow window;
7     protected final T           turtle;
8
9     protected final TurtleStack  stack = new TurtleStack ();
10
11    public abstract TurtleCommand commands ();
12
13    public SingleTurtleScene () {
14      window = createWindow ();
15      turtle = createTurtle ();
16      initialize ();
17      run ();
18    }
19
20    public T            createTurtle () {return (T) new StandardRotationalTurtle ();}
21    public TurtleWindow createWindow () {return new TurtleWindow ();}
22
23    public SingleTurtleScene<T> initialize () {
24      window.add (turtle).setVisible (true);
25      return this;
26    }
27
28    public SingleTurtleScene<T> run () {
29      turtle.run (this.commands ());
30      window.pause (0);
31      return this;
32    }
33  }
```

## 11.3  DarkSingleTurtleScene

This is just like `SingleTurtleScene`, except that it defaults to a dark background and light-colored lines.

```
1   package cheloniidae.frames;
2
3   import cheloniidae.Turtle;
4   import static cheloniidae.frames.CoreCommands.color;
5
6   import java.awt.Color;
7
8   public abstract class DarkSingleTurtleScene<T extends Turtle> extends SingleTurtleScene<T> {
9     public DarkSingleTurtleScene initialize () {
10      window.setBackground (new Color (0.05f, 0.06f, 0.08f));
11      turtle.run (color (new Color (0.8f, 0.8f, 0.9f, 0.3f)));
12      super.initialize ();
13      return this;
14    }
15  }
```

## 11.4 CoreCommands

These are constructors written for convenience. Instead of constructing turtle commands manually, e.g. new Move (100), you can drop the new and write move (100). There are also shorthands to save typing when specifying colors and vectors, as the constructors for these can be implicitly inferred.

```
1   package cheloniidae.frames;
2
3   import cheloniidae.*;
4   import cheloniidae.attributes.*;
5   import cheloniidae.commands.*;
6   import cheloniidae.predicates.*;
7   import cheloniidae.replicators.*;
8   import cheloniidae.transformations.*;
9
10  import java.awt.Color;
11
12  import java.util.Random;
13
14  public abstract class CoreCommands {
15    private static final Random rng = new Random ();
16
17    public static Move move (final double distance) {return new Move (distance);}
18    public static Jump jump (final double distance) {return new Jump (distance);}
19
20    public static Turn  turn  (final double angle) {return new Turn (angle);}
21    public static Bank  bank  (final double angle) {return new Bank (angle);}
22    public static Pitch pitch (final double angle) {return new Pitch (angle);}
23
24    public static LineSize  size    (final double size)     {return new LineSize (size);}
25    public static LineColor color   (final Color color)     {return new LineColor (color);}
26    public static Visible   visible (final boolean visible) {return new Visible (visible);}
27
28    public static LineColor color   (final double r, final double g, final double b)
29      {return color (r, g, b, 1.0);}
30    public static LineColor color   (final double r, final double g, final double b, final double a)
31      {return color (new Color ((float) r, (float) g, (float) b, (float) a));}
32
33    public static Position  position  (final Vector position)  {return new Position (position);}
34    public static Direction direction (final Vector direction) {return new Direction (direction);}
35    public static DirectionComplement directionComplement (final Vector directionComplement)
36      {return new DirectionComplement (directionComplement);}
37
38    public static Position  position  (final double x, final double y, final double z)
39      {return position  (new Vector (x, y, z));}
40
41    public static Direction direction (final double x, final double y, final double z)
42      {return direction (new Vector (x, y, z));}
43
44    public static DirectionComplement directionComplement (final double x, final double y, final double z)
45      {return directionComplement (new Vector (x, y, z));}
46
47    public static NullCommand pass ()                        {return new NullCommand ();}
48    public static Debug       debug (final String text) {return new Debug (text);}
49
50    public static When    when    (final Predicate<Turtle> decisional, final TurtleCommand ... commands) {
51      return new When (decisional, commands);
52    }
53
54    public static When    unless  (final Predicate<Turtle> decisional, final TurtleCommand ... commands) {
55      return when (new Negation<Turtle> (decisional), commands);
56    }
57
58    public static Repeat   repeat   (final int repetitions, final TurtleCommand ... commands)
59      {return new Repeat (repetitions, commands);}
60
```

```java
   public static Sequence sequence (final TurtleCommand ... commands) {return new Sequence (commands);}

   public static Triangle triangle (final SupportsPosition p1, final SupportsPosition p2) {return new Triangle (p1, p2);}
   public static Triangle triangle (final TurtleCommand    c1, final TurtleCommand    c2) {return new Triangle (c1, c2);}

   public static InductiveReplicator<StandardRotationalTurtle> inductiveReplicator
       (final int copies, final TurtleCommand inductiveStep, final TurtleCommand ... replicatedActions) {
     return new InductiveReplicator<StandardRotationalTurtle> (copies, inductiveStep, replicatedActions);
   }

   public static InductiveReplicator<StandardRotationalTurtle> copy (final TurtleCommand ... copiedActions) {
     return inductiveReplicator (1, pass (), copiedActions);
   }

   public static BandSplitReplicator bandSplitReplicator (final TurtleCommand firstTurtlePrimer,
                                                          final TurtleCommand ... replicatedActions) {
     return new BandSplitReplicator (firstTurtlePrimer, replicatedActions);
   }

   public static RecursiveExpansion recursiveBlock (final String name, final TurtleCommand ... body) {
     return new RecursiveExpansion (name, sequence (body));
   }

   public static RecursiveExpansion.Marker recurse (final String name, final int remainingExpansions,
                                                    final Transformation<TurtleCommand> inductiveTransformation,
                                                    final TurtleCommand ... baseCommands) {
     return new RecursiveExpansion.Marker (name, remainingExpansions, inductiveTransformation, sequence (baseCommands));
   }

   public static Scale scale (final double factor)                          {return new Scale (factor);}
   public static Scale scale (final double factor, final boolean scaleLineSize) {return new Scale (factor, scaleLineSize);}
   public static Identity identity ()                                       {return new Identity ();}

   public static <T extends Transformable<T>> Compose<T> compose (final Transformation<T> ... transformations)
     {return new Compose<T> (transformations);}

   public static double random ()                          {return rng.nextDouble ();}
   public static double random (final double scale)        {return random () * scale;}
   public static double random (final double min, final double max) {return random (max - min) + min;}

   public static Predicate<Turtle> hasAttribute (final Predicate<Attribute> predicate) {
     return new HasAttribute (predicate);
   }

   public static AddAttribute addAttribute (final Attribute a) {return new AddAttribute (a);}
   public static Named        named       (final String name) {return new Named (name);}
   public static Pause        pause       (final long time)   {return new Pause (time);}

   // Can't make this plural because then it would atomize a sequence and not the command that we want to atomize.
   // To compensate, we could wrap each subcommand with a non-distributive proxy, but that sounds like a lot of
   // work for such a simple function.
   public static NonDistributiveProxy atomic (final TurtleCommand c) {return new NonDistributiveProxy (c);}

   public static TriangleEmitter.Start triangleStart () {return new TriangleEmitter.Start ();}
   public static TriangleEmitter.Emit  triangleEmit  () {return new TriangleEmitter.Emit  ();}
}
```