Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Interlude: Expressions and Statements

Overview

This interlude explores what you've learned so far about Java expressions and statements. There is a supporting executable, distributed as a part of the on-line supplement to this book, which allows you to experiment with the ideas described here. The goals of this interlude are to show you how expressions and statements can be used in context and simultaneously to give you an opportunity to explore interactions among entities and how these interactions can be used to generate a variety of basic behaviors.

Objectives of this Interlude

1. To increase familiarity with expressions and statements, including return statements and conditionals.
2. To be able to read and write simple sequences of instructions.
3. To appreciate how multiple independent instruction followers can produce behavior through their interactions.
4. To begin to appreciate the differences between parameters, local variables, and fields.

The Problem

This interlude is inspired by a simple child's toy called an Etch A Sketch®. In case you may not be familiar with an Etch A Sketch®, here is a brief description:[Footnote: The Etch A Sketch® product name and the configuration of the Etch A Sketch® product are registered trademarks owned by the The Ohio Art Company. Used by permission.]

An Etch A Sketch® is a rectangular frame (generally red) with a silver screen in the center and two white knobs, one in each of the lower corners. Inside the silver screen is a point of darker grey:

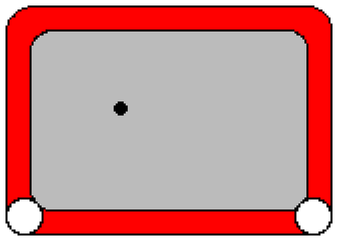

Figure 1: A simple drawing tool.

An Etch A Sketch® is a simple drawing tool. Turning the knobs moves the darker grey point around the screen. As the point moves, it leaves a darker grey trail behind it, showing where it has been. By coordinating the motion of the knobs, you can draw pictures.

On an Etch A Sketch®, each knob controls one direction of motion of the darker grey dot. Rotating the left knob moves the dot from side to side. Rotating the right knob moves the dot up and down. Keeping a knob still prevents the dot from moving in the corresponding direction. So the position of the knob determines the position of the dot on the screen (in one dimension) and changing the knob position moves the dot (in that dimension).

By rotating just one knob -- by leaving the position of the other knob fixed, or constant -- you can draw a straight (horizontal or vertical) line, as in figure 2. By rotating both knobs at appropriately coupled velocities, you can draw diagonal lines of varying slope. Proficient Etch A Sketch® users can draw complex pictures by coordinating the knob postion changes.

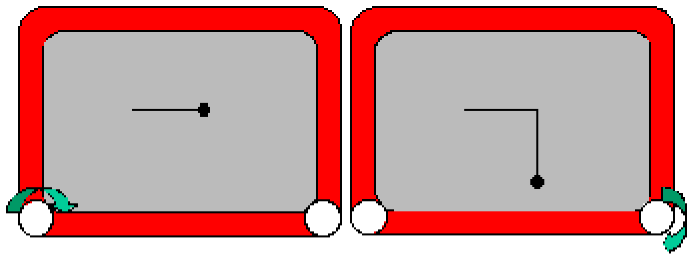

Figure 2: Drawing a straight line by rotating the appropriate knob.

In this interlude, we will explore the instructions required to perform similar operations on a similar (though less brightly colored) display. In our application, we will supply a rule that describes the current position of each knob. For example, the position might be constant -- always the same -- in which case, the dot wouldn't move. Or it might be steadily increasing, in which case the dot would move steadily across the screen.

Each rule will be read (and executed) repeatedly. It is as if, behind the scenes, an instruction-follower were to continually check the position of each knob and update the position of the dot correspondingly. Each time that the instruction follower wants to know what to do with the dot, it will ask our rule. If the rule always gives the same value, it will be as though that knob is stuck in one position. If the rule changes the value, the same change will be made to the knob's position over and over again. So, for example, if the rule says "increase the current position", the dot will move accross the screen until it reaches the edge.

In fact, there will be two instruction-followers, one for each knob. In addition, they're not guaranteed to run at the same speed, or even to take fair turns. So, even if both knobs were to have the same rules, we might discover that our horizontal knob was moving twice as fast as our vertical knob.

Q. What would the resulting picture look like?

Ans. It would slope upwards gradually. @@supply pic.

Or the horizontal knob instruction follower might check its rule three times, then the vertical knob once, then the horizontal knob once, then the vertical knob three times, then both at once.

Since the knobs are being checked by independent instruction-followers, any schedule is possible in principle. By observing the actual behavior of the system, we can try to write rules to explicitly coordinate the behavior of the two knobs. To begin with, we'll just assume that they run at about the same rate.

An interesting feature of our program is that the knob-rules don't have any way to tell which knob they're controlling. The rules just say things like "turn the knob to a higher value" or "turn the knob lower" or "set the knob to the middle" (or "...halfway to the edge").

But on to the details....

Representation

In our application, instead of actually rotating knobs, we will represent a knob position as a number. We will use a standard Cartesian coordinate frame, with (0,0) in the center, increasing horizontal coordinates to the right, and increasing vertical coordinates at the top. [Footnote: In a later chapter, we'll see computer graphics that use a different coordinate system, sometimes known as "screen coordinates".]

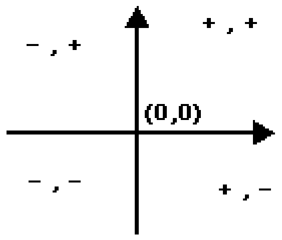

Figure 3: Cartesian coordinates.

The value of the knob position can range between `-maxPos` and `maxPos`. Since (0,0) is dead center on the screen, (0, `maxPos`) is the center top of the screen, while (`-maxPos/2`, 0) is middle height, halfway to the left edge. The actual size of the Etch A Sketch® window (and therefore the value of `maxPos`) will vary as the window is resized. [Footnote: In particular, `maxPos` may have different values in the horizontal and vertical rules. `maxPos` is simply the largest coordinate in whichever dimension the rule controls.]

Q. What four points represent the four corners of the Etch A Sketch® window?

ans. (-maxPos, -maxPos), (-maxPos, maxPos), (maxPos, -maxPos), (maxPos, maxPos). These are lower left, upper left, lower right, and upper right in order. But note that maxPos in the vertical dimension may not have the same value as maxPos in the horizontal dimension, and also that the value of maxPos may change as the window is resized.

Our job will be to write the instructions for the next knob position: a rule that returns the desired dot position. We'll need one rule for the horizontal knob and one for the vertical, of course.

The form of a control rule is a sequence of Java statements ending in a statement of the form

```
    return double;
```

where *`double`* is some Java expression with type `double`. The value returned by your control rule will be used as the new position of the dot. So, for example,

```
    return 0;
```

is a rule that holds the knob in the center, i.e., keeps the dot in the middle of the screen. (Why is it ok to return 0 rather than 0.0?)

Remember, though, that this is just one rule. The other rule might be moving the dot along steadily, or holding it at the edge of the screen, or doing something else entirely. If `return 0;` is the vertical rule, then the dot will remain halfway up the screen, but it could be anywhere along that halfway line.

Q. What rule would keep the dot centered horizontally (halfway between the left and right edges) on the screen? Is this a horizontal- or a vertical-knob control rule?

ans. `return 0;` *as a horizontal-knob control rule. Using this rule, the dot may move up and down (depending on the vertical control rule), but will always be halfway between the two sides of the screen. Its horizontal position is fixed, i.e., constant.*

horizontal rule
`return 0;`

If you use this rule as a vertical control rule, it will keep the dot halfway up the screen. Its side-to-side (horizontal) position would be determined by the horizontal control rule.

vertical rule
`return 0;`

Q. How would you position the dot almost in the upper right-hand corner? The answer should involve a horizontal rule *and* a vertical rule.

ans. The horizontal rule should position it at the far right of the screen. Examining the coordinate system (figure ??), we see that this is at maxPos. So `return maxPos;` *as a horizontal-knob control rule. The vertical control rule should position the dot at the top of the screen, which is maxPos in that dimension. So the vertical rule is also* `return maxPos;`

horizontal rule	**vertical rule**
`return maxPos;`	*`return maxPos;`*

Q. How about the upper left-hand corner? Again, the answer should involve a horizontal rule and a vertical rule.

ans. In this case, the horizontal rule should put the dot at the far left, which (according to the coordinate frame in figure 3) is at -maxPos. So the horizontal rule should be `return -maxPos;` *The vertical rule should put the dot at the top, which is still maxPos, so the horizontal rule should be* `return maxPos;` *as in the previous question.*

horizontal rule	**vertical rule**
`return -maxPos;`	`return maxPos;`

The vertical rule has not changed. But because it is interacting with a different horizontal rule, the on-screen behavior is different.

Q. Assume that you have rules that position the dot in the upper left-hand corner. Now suppose that you swap the horizontal and vertical rules. Where is the dot now? What if you use the horizontal rule for both horizontal and vertical behavior?

ans. This question involves swapping the horizontal and vertical rules from the previous question:

horizontal rule	**vertical rule**
`return maxPos;`	`return -maxPos;`

Note that it is the vertical rule that now returns a negative value. Examining the coordinate frame in figure 3, we see that the horizontal rule puts the dot on the right-hand edge of the frame, while the vertical rule puts the dot at the bottom. So the result will be a dot in the lower right-hand corner of the frame. Swapping the rules -- so that the same two rules play different roles -- changes the behavior of the system, too.

A rule is a sequence of statements including a value returning statement. A rule is executed by following the instructions until a return statement is reached. This kind of rule is essentially the body of a Java method. This application focuses attention on the sequence of statements -- the method body -- but its ideas apply to Java methods as well.

Interacting With The Rules

In introducing this system, we said, "Each rule will be read (and executed) repeatedly." But nothing about the observable behavior of the system has really made this clear. In fact, in the examples above, each rule might be followed only once and the same results would still be produced. Each of the rules that we have used so far has been like setting the Etch A Sketch® knob to a certain position. If you set the knob to the center, it looks the same whether you do it once (and then just leave it there) or do it over and over, repeatedly making sure that the knob is set to the center position. So how can we tell what the Etch A Sketch® is actually doing?

If we had a way to change the position of the dot (without using the rules), we could see that the rules are repeatedly executed. If they

In order to see that the rules are executed repeatedly, we need another (non-rule) way to change the position of the dot. If we could move the dot this way, we would see that continued execution of the rules forces it back to the "stuck knob" position described by the rules. Imagine that both rules say

```
    return 0;
```

So each instruction follower will keep putting the dot right in the middle of the screen. This won't look like much until we move the dot. But if we get the dot to jump to the top of the screen, the next time that the instruction follower executes the rule it will still put the knob back into the position mentioned in the rule -- the center of the screen, in our example. We can see that the rules for our Etch A Sketch® are invoked repeatedly, rather than just run once, by interacting with them.

In fact, in our Etch A Sketch® application, you can move the dot using your mouse. By clicking the mouse at a particular point on the Etch A Sketch® screen, you move the dot there. Almost immediately, the instruction followers go back to their knob-rule-checking. This means that if you position the dot with the mouse, it may not stay there for very long. Fortunately, you can see where the dot has moved because every time that the instruction follower moves the dot, it leaves a (black) trail behind it. Indeed, we will use this feature to draw pictures. So, for example, you can make a line from the upper lefthand portion of the Etch A Sketch® to the center by using the rules

horizontal rule	**vertical rule**
`return 0;`	`return 0;`

and clicking the mouse in the upper lefthand portion of the window:

@@add pic

The mouse click moves the dot momentarily, then the execution of the rules brings the dot back to the center, leaving a trail. You -- the user -- provide a third independent control to this application. By interacting with the dot, you, too, affect its behavior.

Q. Combining these two observations -- leaving trails and "jumping" the dot around using the mouse -- can you figure out how to create an asterisk (a bunch of line segments intersecting in the center)?

ans. If you run the Etch A Sketch® with both rules set at `return 0;` *the dot will return to the middle, no matter where you move it with the mouse:*

horizontal rule	**vertical rule**
`return 0;`	`return 0;`

Now, with the mouse, click anywhere. The dot will jump there, then immediately (well, as soon as the horizontal and vertical rules are sampled) return to the center. This should draw a line from wherever you clicked to the center. Repeat this, clicking somewhere else. Eventually, you'll have a lovely asterisk.

@@add a picture.

Actually, you can make an asterisk around any point in the screen, not just the center of the screen. Try setting the rules to:

horizontal rule	**vertical rule**
`return maxPos/2;`	`return 0;`

Then repeat the clicking around process. Now you should get an asterisk centered at (maxPos/2, 0), i.e., centered vertically but in the right half of the screen.

@@another pic.

Rule bodies as we have presented them here are really just very simple method bodies. The return statement supplies the return value of its containing method. In the rule form that we are using here, the enclosing method declaration and braces are omitted. We will see more of how to write and use methods in the next chapter.

By manipulating the behavior of each method, the role played by each, and how we and the system interact, we can generate a variety of different behaviors even with extremely simple code.

Paying Attention to the World

So far, the rules that we have written return the same value, no matter where the dot starts out. This corresponds to a rule that drives the knob to a certain fixed position, regardless of where it starts out. When you don't move the dot with the mouse, the rule causes the dot to sit still. When you do use the mouse to move the dot, the rule causes the dot to jump back to the same place that it has been. In this section, we will see how our rules can respond to information that they are given.

Each time that an instruction follower goes to execute a rule, the name `pos` is been pre-defined for it to hold the current position of the dot (along the relevant dimension). So, if the dot is half-way between the left side of the screen and the center, `pos` will be `-1/2 maxPos` when the horizontal rule is invoked, while if the dot is all the way at the right side of the screen, `pos` will be `maxPos`.

Q. Using this information, write a rule that causes the dot to stay where it is.

ans. "Where it is" is always pos -- pos is the current position of the dot when the rule is about to be executed. So, no matter where the dot is, we can make it stay there using the rule

```
return pos;
```

If the dot is at 36, pos will be 36 and this rule will return 36. If we click the mouse and move the dot to 78, pos will be 78 the next time that the rule is executed, and we will return pos, or 78. Since pos is always where the dot is, returning pos will keep the dot there.

Note that pos is defined anew each time that the rule is executed. It is, in effect, a parameter to the rule.

Q. What happens when the dot is moved, using the mouse?

ans. When the dot is moved, the rule still says "stay where you are." So each time you click the mouse, the rule adjusts the knob to keep the dot where you've put it.

Note that the horizontal rule and the vertical rule each have their own version of `pos`. So `pos` in the horizontal rule has nothing to do with `pos` in the vertical rule; each gets its own proper position.

Now, using the information that pos is where the dot is, we can cause the dot to move. Each time the knob is checked -- each time the rule is invoked -- the knob should turn just a little bit.

Q. What would such a rule look like?

ans. We could use a rule that says

```
return pos+1;
```

This rule checks where the dot is, then instead of setting the knob there, it moves the knob slightly. The next time the rule is executed, it will move the dot a little bit further over. This will continue to happen until the dot reaches the edge of the screen.

[Why not pos=pos+1? Pos is a local name that this piece of code has for the current position of the dot. It is NOT the "control" for the current position of the dot. When the instruction follower is about to execute the instructions, it creates a new dial -- called pos -- and sets it to the value that represents the current position of the dot. After this happens, there's no additional connection between the value of the pos dial and the position of the dot. Reading the pos dial tells you what the current position of the dot is. But changing the pos dial doesn't move the dot. Returning a value does.

What would happen, then, if we ran with the horizontal rule pos=pos+1? First, we'd get an error. Remember, a rule has to end with return double; So now consider pos=pos+1; return 0; This would be exactly the same as just return 0;, i.e., it would drive the dot to the center. How about pos=pos+1; return pos; ? This works, but it isn't as "nice" as return pos+1; The reason it works is that it first modifies the pos dial to have the value we want to return (pos+1), then returns the value on the dial. There's really no reason to modify the dial; we can just return the value directly.]

Pos is a parameter -- a name whose value is defined before the rule (method) body begins to execute. We can create names (dials) of our own as well. For example, a much more long-winded way of writing the previous rule might be:

```
double velocity = 1;   // how much the position changes by.
   double newPos;         // what the new position will be.

   newPos = pos + velocity;   // compute next position...
   return newPos;             // ...and return it.
```

The names *velocity* and *newPos* here are new local variables we create. Their declarations last until the return statement. Each time that this set of instructions is executed, the declaration line `double velocity = 1;` is executed again, and a new dial called velocity is created. (Yes, that's a lot of wasted dials. Don't worry; Java has facilities to make sure they are recycled.) In a later section, we will see a different kind of name that persists from one rule execution to the next.

Q. What happens when the dot reaches the edge of the screen?

ans. At this point, pos will continue to be increased. But values greater than `maxPos` *aren't allowed. (The application is written so that values greater than maxpos are treated just like* `maxPos`*, so the dot will sit at the edge of the screen.*

Q. How would you make the dot move in the other direction?

ans. With a rule that says

```
return pos-1;
```

This rule makes the knob turn a little bit in the opposite direction. Remember, returning a value is the way to move the dot.

Fancy Dot Tricks

Q. What would happen if you used the rules:

horizontal rule	**vertical rule**
`return pos+1;`	`return 0;`

Assume that the dot starts in the center of the screen.

ans. You would get a horizontal line from the center of the screen to the right hand edge of the screen.

Q. How would this be different if the rules were

horizontal rule	**vertical rule**
`return pos+1;`	`return pos;`

ans. If the dot starts in the center of the screen and you don't click the mouse, you wouldn't be able to tell whether the vertical rule said return 0; or return pos; The value of pos (for the vertical rule) would start out as 0, and since nothing changes it, it would remain 0. The only way to see a difference is to move the dot (using the mouse). If you let the dot move across the screen until it's halfway to the right edge, then click the mouse in the lower left (at the X), here's what you'll see:

@@add picture

Q. What does the dot do if you start in the lower left hand corner of the screen and use the rules

horizontal rule	**vertical rule**
`return pos+1;`	`return pos+1;`

ans. This rule pair would draw a diagonal line from the lower left hand corner of the screen towards the upper right hand corner. [Footnote: Actually, the line would only go towards the corner if the screen were relatively square. This diagonal line has a slope of 1.]

Q. Can you make the dot move from the `maxPos` edge of the screen to the other edge when it gets there?

ans. In order to do this, we need to check whether we've gotten to the `maxPos` *edge. We can do this using an if statement:*

```
if (pos < maxPos)
{
    return pos + 1;
}
else
{
    return -maxPos;
}
```

Q. Can you make the dot move more quickly across the window?

ans. In the previous rules, we've increased pos by 1 each time. If we increase pos by a larger number, it will move more quickly. In fact, this increase to pos is the velocity -- the speed -- of the dot.

We can use this rule to create a sort of barber-shop pole effect -- a slowly climbing spiral around the window. To do this, we use a horizontal wrap-around rule and a vertical wrap-around rule. By setting the horizontal rule to move more quickly than the vertical rule, we get a line with a gradual slope. Since we're using wrap-around rules, the line repeats over and over again as it moves up the screen.

So starting in the lower left hand corner of the screen and executing

horizontal rule	**vertical rule**

```
if (pos < maxPos)
{
    return pos + 5;
}
else
{
    return -maxPos;
}
```
	```
	if (pos < maxPos)
	{
	    return pos + 1;
	}
	else
	{
	    return -maxPos;
	}
	```

produces something like:

@@add pic. a sequence would be better.

For each Etch A Sketch® rule, pos is a name whose value is fresh each time the rule is executed. There is no connection between the value of pos from one invocation of the horizontal rule to the next. The value of pos for the horizontal rule is unrelated to the value of pos for the vertical rule. This behavior is essentially the behavior of a parameter to a Java method. In the next section, we will see a different kind of name.

Remembering State

In the previous section, we saw how to prevent the dot from getting stuck at one edge of the screen by jumping it to the other edge. It might have been nice to have the dot bounce back from the edge -- turning to move in the opposite direction -- instead. It turns out that that behavior requires an additional idea and a corresponding bit of machinery.

Suppose that we wanted to get the dot to turn around. We might start with a rule that looks like the "jump to the other edge rule, trying to detect when we've bumped into the *maxPos* edge:

```
if (pos < maxPos)
{
    return pos + 1;
}
```

This rule seems reasonable enough. It will cause the dot to move along until it reaches maxPos. But what then? When we reach maxPos, the if test will fail and we'll drop through to the else clause. It goes through `maxPos - 2`, `maxPos - 1`, `maxPos`. Now, it needs to go to `maxPos - 1`, (and then to `maxPos - 2`, `maxPos - 3` and so on). So we might try

```
else
{
    return pos - 1;
}
```

Sure enough, the dot's positions will be `maxPos - 2`, `maxPos - 1`, `maxPos`, `maxPos - 1`. But then what? The problem is that when the dot is at `maxPos - 1` and this rule is executed again, the if test will succeed! The next position of the dot will be (`(maxPos - 1) + 1`), or `maxPos`! Then the if test will fail, triggering the else clause: `maxPos - 1`. At this point, the dot will oscillate between `maxPos - 1` and `maxPos` forever.

What went wrong? As always, our errors are informative. The problem is that the condition we're testing -- whether our position is < maxPos -- doesn't really tell us what we need to know -- which direction to move in. Our position might be `maxPos - 1` because we're heading towards `maxPos`, or it might be `maxPos - 1` because we're heading back towards `maxPos - 2`. The if test doesn't give us any way to tell the difference. In fact, nothing about the current rule execution or our current position can answer this question for us. Instead, we need to know something about the *previous* execution, or about where we've been.

Fields

At this point, we need to introduce some new machinery. In our application, there is a special box (for each dimension) where we can enter names that persist from one execution of a rule to the next. These names correspond to fields of instance objects. They are like airport lockers, places that you can leave things when you're executing the rule and find them the next time you come back into town.

There are several different ways we can use airport lockers to solve this problem. The simplest is probably just to remember which direction we're going in. We can do this using a boolean name. In this case, we'll call the boolean increasing. We start with increasing true. So the declaration should say:

fields

```
boolean increasing = true;
```

(Recall that a declaration follows the type-of-thing name-of-thing rule. So this declaration says we have a boolean -- a true-or-false kind of dial -- that is called increasing. Because this is a definition, not just a declaration, it also sets the dial to read true.)

Now, we can write an if statement that says what to do if we're increasing: increase pos, unless we've hit the edge.

```
if (increasing)
{
    if (pos < maxPos)
    {
        //keep going higher -- return the next position
        return pos + 1;
    }
    else //not (pos < maxPos)
    {
        //we've hit the edge -- turn around
        increasing = false;
        return pos;
    }
}
```

The else condition is similar, but with the signs reversed:

```
else //not (increasing)
{
    if (pos > - maxPos)
    {
        //keep going lower -- return the next position
        return pos - 1;
    }
    else //not (pos < maxPos)
    {
        //we've hit the edge -- turn around
        increasing = true;
        return pos;
    }
}
```

Q. Can you write a similar rule that relies on a numeric piece of state -- double previouspos -- instead? What does the declaration of persistent state look like? What happens the first time the rule is executed?

ans. First, declare a field (in the special box):

fields

```
double previousPos;
```

Note that the code starts by checking the boundary cases. If we're at the edge, we need to go inwards. Otherwise, we remember where we were this time (previousPos=pos;) and return a number that continues moving the dot in the appropriate direction.

```
if ( pos >= maxPos )
{
    //we're at the higher edge.
    previousPos = maxPos;
    return maxPos - 1;
}
else if ( pos <= -maxPos )
{
    //we're at the lower edge.
    previousPos = -maxPos;
    return maxPos + 1;
}
else if (pos > previousPos)
{

    //we're moving up: return a higher number.
    previousPos = pos;
    return pos + 1;
}
else // (pos <= previousPos)
{
    //we're moving down; return a lower number.
    previousPos = pos;
    return pos - 1;
}
```

Fields vs. Variables

Consider the previous example: using previousPos to keep track of which direction we were going. Why do we need previousPos here? Why can't we just use pos? There are two reasons. First, pos is already being used for something -- the current position of the dot. But the other reason is that pos gets a new value each time this set of instructions is executed. (Actually, pos gets re-created each time this set of instructions is executed.) So, if we put something we want to remember into pos, it won't be there the next time that these instructions are executed. We need to create a special value -- a field -- to hold things that we want to remember from one execution of these instructions to the next.

Field declarations must be made in the special box. Declarations in the regular code box are allowed, but they do not carry over from one execution to the next. Instead, a name declared in running code is a temporary scratch space. The corresponding dial or label is created each time that the declaration is executed (as a part of following those instructions) and discarded when the return statement is reached.n Such local scratch space is called a local variable.

Contrast this with the use of velocity and newPos in an earlier section:

```
      double velocity = 1;   // how much the position changes by.
         double newPos;         // what the new position will be.

         newPos = pos + velocity;   // compute next position...
         return newPos;             // ...and return it.
```

Velocity and *newPos* here are local variables. They are *not* fields. That is, they are new dials that are created each time the rule is executed -- local scratch space that only exists during a single rule execution -- and they go away when the rule execution is done. Next time the rule is executed, they will be recreated. In contrast, a field -- like previousPos in the rule above -- sticks around from one rule execution to another.

Summary

In this chapter, we have seen simple pieces of Java code that produce behavior. Each short set of instructions is in effect the body of a Java method; a value is returned at the end. The behavior of the system as a whole depends on the particular methods written. In addition, system behavior depends on how those rules are coupled together and how you as a user interact with them.

There are three different kinds of names that can be used in your code. First, you can use names that have been pre-defined for you, like `pos`. These are called ***parameters***. In other chapters, we will see that in a Java method, all parameter names are included in the method declaration.

There are also two kinds of names that are declared using standard declarations or definitions. One kind is a temporary name that can be used during a single application of your rule. These names can be declared anywhere in your code. They are called ***variables***. `newPosition` and `velocity` are examples of variables. Variables can be declared inside a method.

The last kind of name sticks around from one use of your rule to another. These names must be declared in a special box, separate from your rule code, but can be used freely in your rule code. These names are called ***fields***. In this chapter, `increasing` is an example of a field. In the Etch A Sketch®, fields are declared in a separate box. In Java code generally, fields are declared outside of methods (but within an enclosing class).

Suggested Problems

See the text for things marked with a **Q**. Also:

1. Implement constant acceleration. Velocity is the change in position over time. For example, the rule return pos + 1; has a velocity of 1, while the rule return pos - 5; has a velocity of 5 in the opposite direction. Acceleration is the change in velocity over time. For example, if we return pos+1; when the rule is executed the first time and then the next time we return pos+3; when the rule is executed the second time, the change in velocity (i.e., the acceleration) is 2. To implement a constant acceleration, you need to

change the velocity by the same amount each time. This means that the rule can't return pos+a constant; instead, it has to return pos + an amount that changes each time. (Hint: Use a field.)

2. Can you make the dot go in a parabolic path? (Hint: what accelerations does it need?)

3. We have given you a parameter named `otherPos`. Each time that a rule is followed, otherPos begins with the position of the dot along the *other* axis. Using this information, implement a function plotter. Write the code to plot the following:

- *y = x^2;*
- *y = sin(x);*
- *y = 1/x;*

You may want to look at the `Math` library.