Java Execution Model Examples

Franklyn Turbak

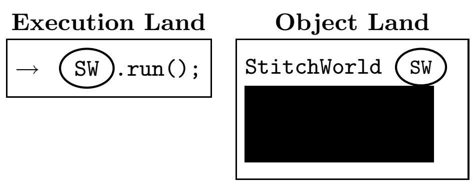
September 18, 2007

1 Void Methods Without Parameters: StitchWorld

As a first example, we consider the `StitchWorld` and `Stitcher` classes on the next page in Fig. 3. The `StitchWorld` class is a subclass of `BuggleWorld` that overrides the `run` method of `BuggleWorld`. This `run` method creates two instances of the `Stitcher` class, which extends the `Buggle` class with five new methods: `stitch4`, `stitch2`, `stitch`, `box`, and `turn180`. All these methods are `void` methods (i.e., they return no results) that take no parameters.

We can understand what the `StitchWorld` `run` method does by carefully drawing a Java Execution Model (JEM) diagram (see Fig. 1) for an invocation of the `run` method. Such a diagram

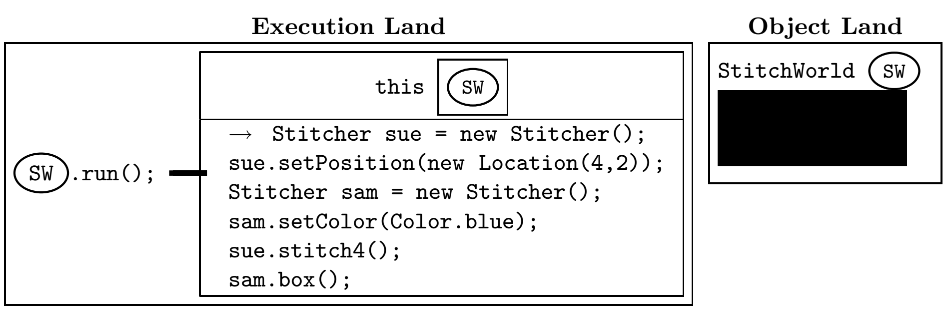
Figure 1: Initial JEM diagram for the `StitchWorld` example.



shows execution frames in Execution Land and objects in Object Land. We assume that `run` is invoked on an instance of `StitchWorld` with object label SW. The `StitchWorld` instance SW is a complex object whose state we do not wish to study in detail, so we shall treat it (and draw it) as a "black box". The **control arrow**, →, indicates which statement is currently being executed.

Invoking the `run` method on SW creates an execution frame (Fig. 2). As always, the `this`

Figure 2: Invoking `run()` creates an execution frame.



variable of the execution frame is the receiver object — the object on which the method was invoked — in this case, SW.

```
public class StitchWorld extends BuggleWorld {

  public void run () {
    Stitcher sue = new Stitcher();
    sue.setPosition(new Location(4,2));
    Stitcher sam = new Stitcher();
    sam.setColor(Color.blue);
    sue.stitch4();
    sam.box();
  }

}

public class Stitcher extends Buggle {

  public void stitch4 () {
    this.stitch2();
    this.stitch2();
  }

  public void stitch2 () {
    this.stitch();
    this.stitch();
  }

  public void stitch () {
    this.forward();
    this.left();
    this.forward();
    this.right();
  }

  public void box () {
    this.stitch();  // draw two sides of box
    this.turn180(); // turn to draw rest of box
    this.stitch();  // draw other two sides of
    this.turn180(); // turn to original heading
    this.stitch();  // move to opposite corner
  }

  public void turn180 () {
    this.left();
    this.left();
  }
}
```

Figure 3: Code for the `StitchWorld` example.

The first statement in the body of the `run` method is the assignment statement

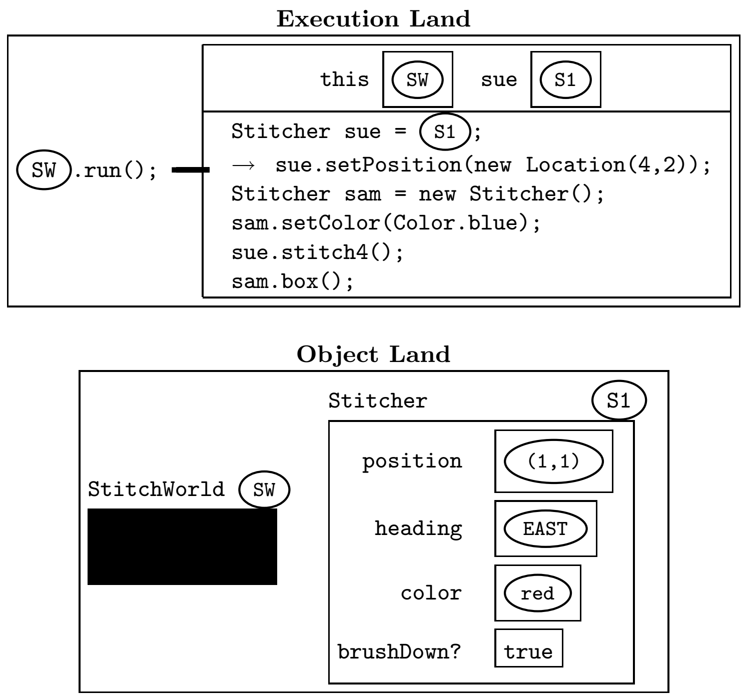
```
Stitcher sue = new Stitcher();
```

Executing this statement takes place in three steps:

1. The variable declaration `Stitcher sue` creates a new local variable named `sue` in the execution frame for `run()`.
2. The instance method invocation `new Stitcher()` creates and returns a new `Stitcher` instance, which we shall assume has object label (S1). A `Stitcher` instance has exactly the same state variables as a buggle.
3. The object label (S1) for the new `Stitcher` instance is stored in the local variable `sue`.

The result of these three steps is depicted in Fig. 4.

Figure 4: JEM after executing the first assignment statement in `run()`.



To cut down on the clutter in Object Land, we use some notational abbreviations for objects in Fig. 4. We use (1,1) as an abbreviation for a new `Location` instance in Object Land (not shown) whose `x` and `y` instance variables both contain the integer 1. Similarly, (EAST) is an abbreviation for a unique `Direction` instance denoting the east direction, and (red) is an abbreviation for a distinguished `Color` instance denoting the red color.

After the execution of the first statement in the `run` method, the control arrow points to the second statement:

```
sue.setPosition(new Location(4,2));
```

Executing this `void` method invocation statement takes place in three steps:

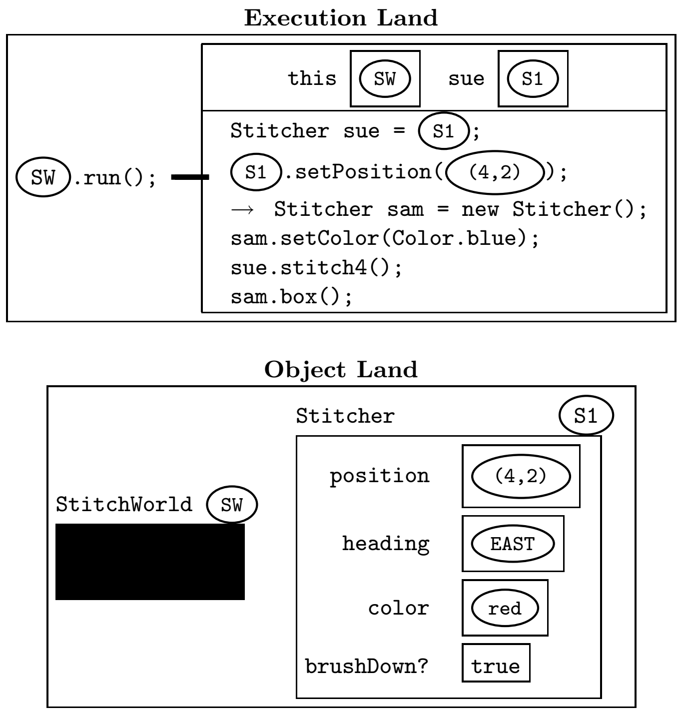
1. The receiver expression `sue` is evaluated. This is a variable reference, whose value is the contents of the local variable named `sue`, which is the object labeled (S1).

2. The argument expression `new Location(4,2)` is evaluated. This creates and returns a new `Location` instance. As above, we will use the abbreviation (4,2) in place of showing the `Location` instance in Object Land.

3. Invoking the `setPosition` method on S1 with the argument (4,2) changes the position of S1 to (4,2).

The result of these three steps is depicted in Fig. 5.

Figure 5: JEM after executing the second statement in the `run` method.



Control now points to the third statement of the `run` method:

```
Stitcher sam = new Stitcher();
```

Executing this assignment statement (1) creates a new local variable named `sam`; (2) creates anotherStitcher instance, which we shall assume has object label S2; and (3) stores S2 the local variable named `sam`. The result of these steps is shown in Fig. 6.

Now control points to the fourth statement of the `run` method:

```
sam.setColor(Color.blue);
```

Executing this `void` method invocation statement (1) evaluates the receiver expression `sam` to S2; (2) evaluates the argument expression `Color.blue` to the object reference blue for an pre-existing `Color` instance in Object Land; and (3) changes the color of S2 to blue. The result of these steps is shown in Fig. 7.

Next comes the execution of the fifth statement in the `run` method, `s1.stitch4();`. This is an invocation of a user-defined method, `stitch4`, on the receiver expression `s1`, whose value is S1. Such an invocation creates a new execution frame whose `this` variable contains the receiver, S1 (Fig. 8). Since Object Land remains unchanged from Fig. 7), we do not show it.

Figure 6: JEM after executing the third statement of the run method.

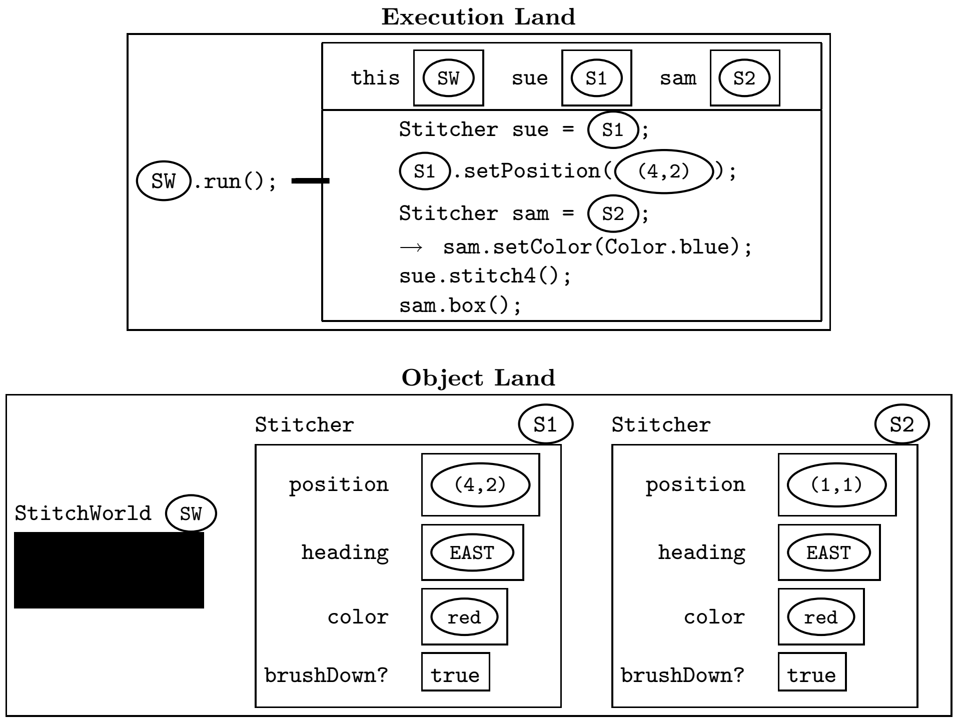


Figure 7: JEM after executing the fourth statement of the run method.

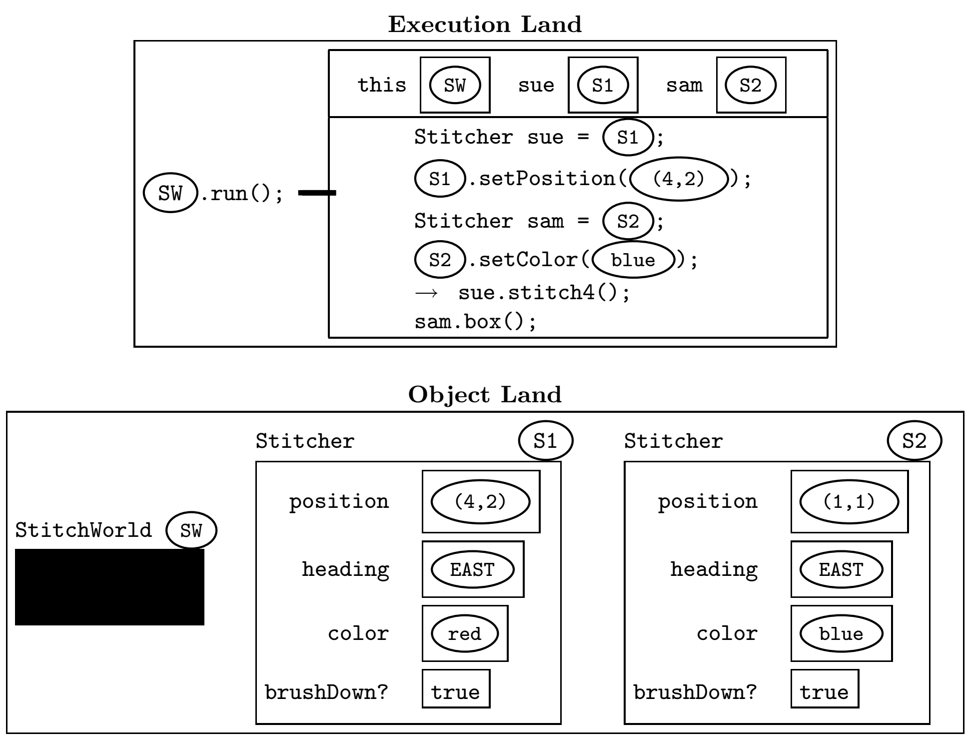
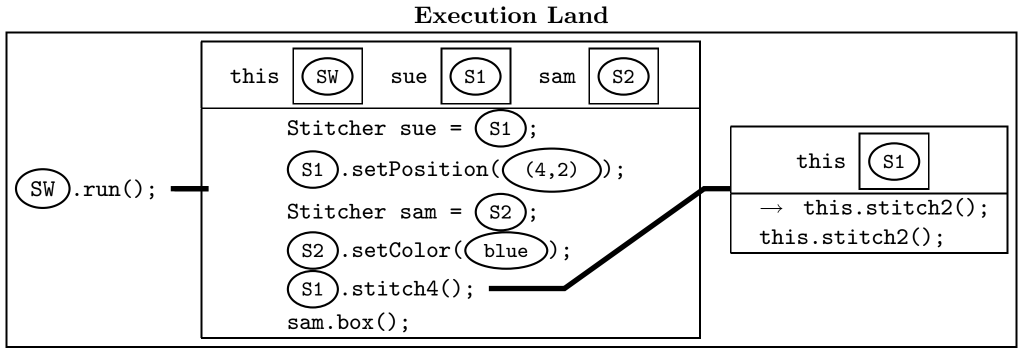
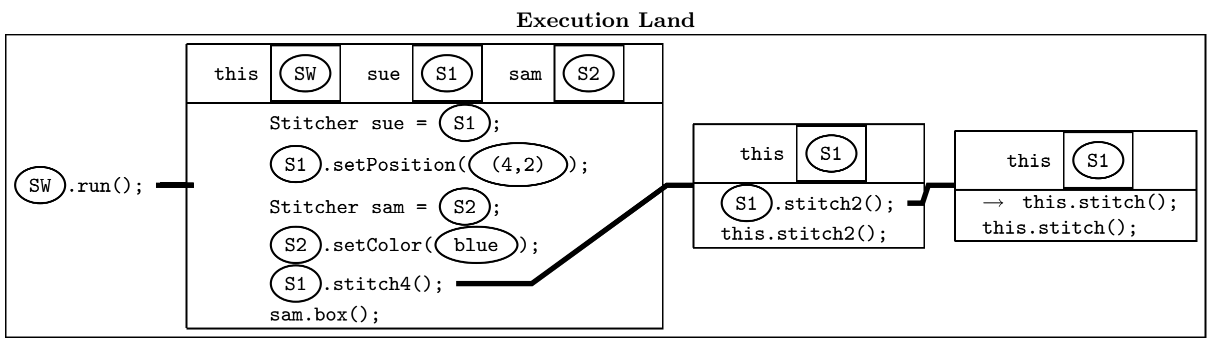


Figure 8: Executing the fifth statement of the `run` method creates an execution frame for `stitch4`.



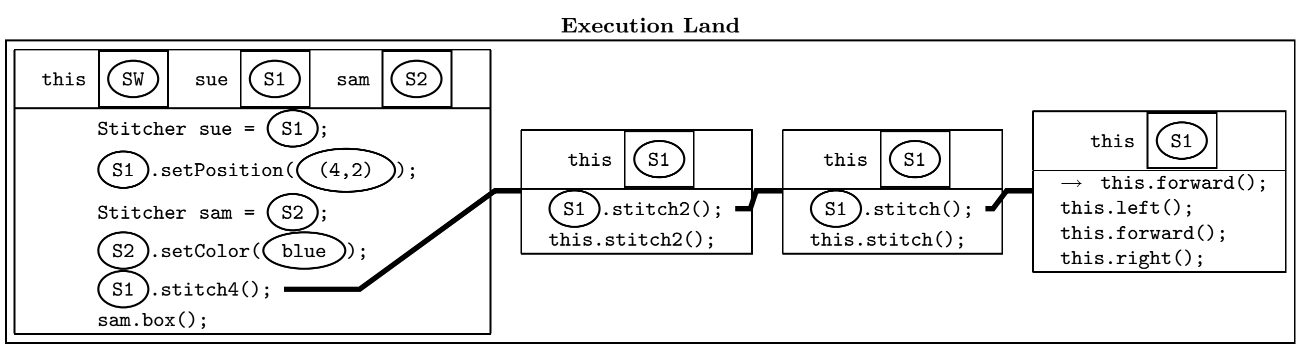
Executing the first statement in the body of the `stitch4` method causes the `stitch2` method to be invoked on S1, which creates another execution frame (Fig. 9).

Figure 9: Executing the first statement of `stitch4` creates an execution frame for `stitch2`.



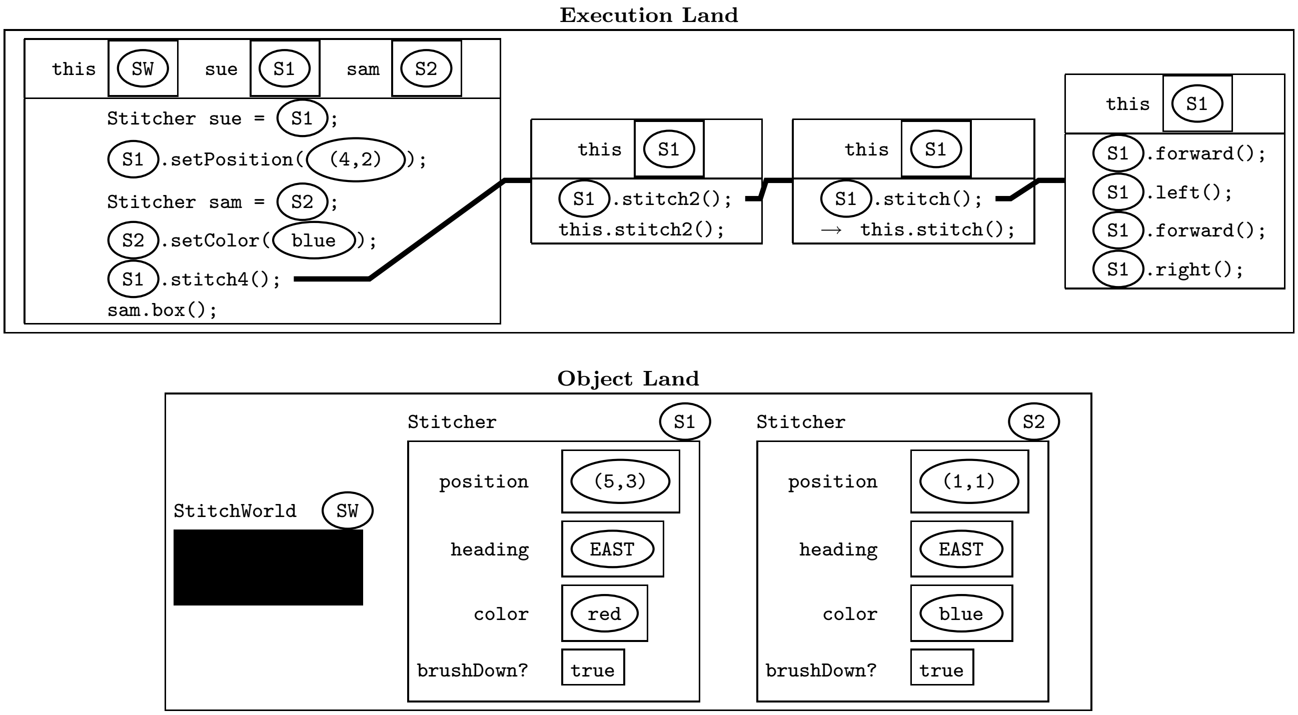
Executing the first statement in the body of the `stitch2` method causes the `stitch` method to be invoked on S1, which creates yet another execution frame (Fig. 10). In the Execution Land of Fig. 10 we omit the initial call to SW`.run()` in order to fit the rest of the execution frames.

Figure 10: Executing the first statement of `stitch2` creates an execution frame for `stitch`.



Executing the four instance method invocations in the body of `stitch` changes Execution Land and Object Land as shown in Fig. 11. No execution frames have been drawn for the instance method invocations of `forward`, `left`, and `right`, because they are considered "primitive"; we are not going to examine the details of how they work.

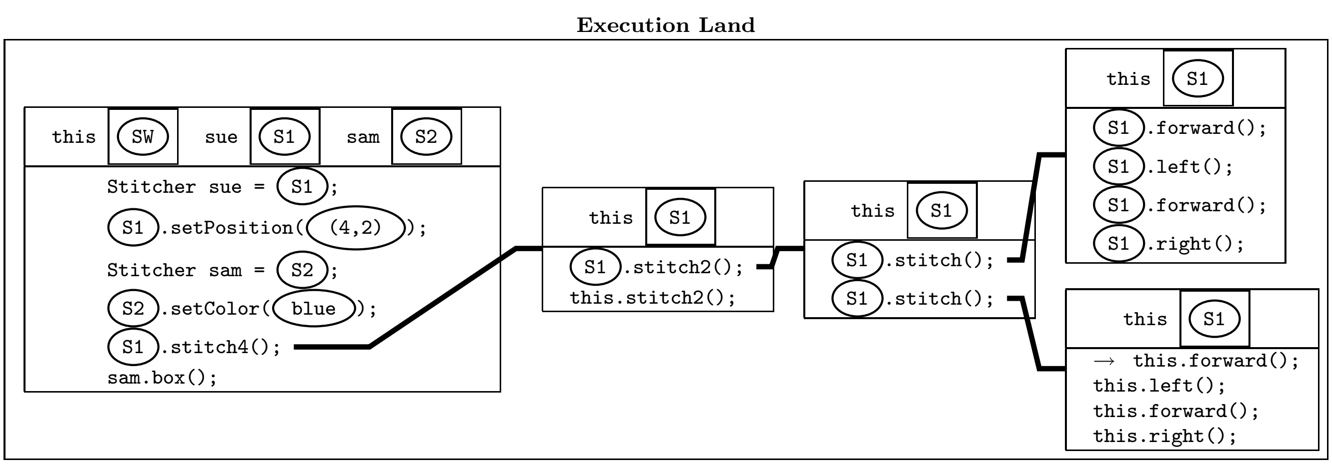
Figure 11: JEM after completing the first invocation of `stitch`.



After executing all four statements in the execution frame for the first `stitch` invocation within `stitch2`, the execution of that invocation is complete, and the control arrow proceeds to execute the second invocation of `stitch` within `stitch2`. In reality, the execution frame for the first `stitch` invocation disappears at this point, but we will continue to display it to show the history of the computation.

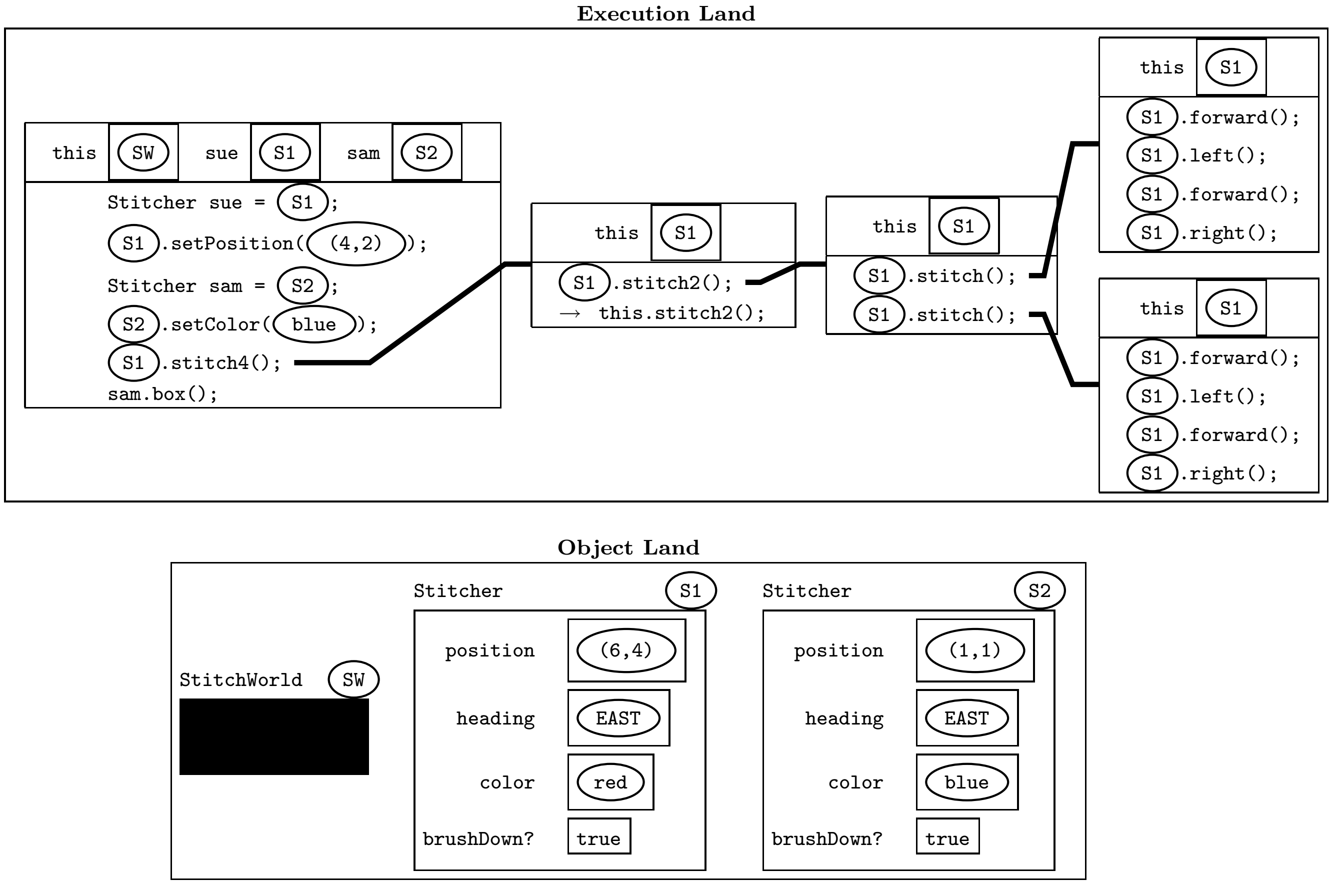
Executing the second invocation of `stitch` within `stitch2` creates a second execution frame for `stitch` (Fig. 12). Performing the four statements in the second `stitch` execution frame yields

Figure 12: The second invocation of `stitch` in `stitch2` creates a second execution frame for `stitch`.



the JEM diagram in (Fig. 13).

Figure 13: JEM after completing the second invocation of `stitch`.



Since the first invocation of `stitch2` is now complete, control proceeds to the second invocation of `stitch2`. Just as in the first invocation of `stitch2`, the second invocation will cause the creation of three new execution frames in the JEM. We shall not show the intermediate steps, just the final state after the execution of the second invocation of `stitch2` (Fig. 14).

The execution of `sam.box` creates six new execution frames, which are shown in Fig. 15 in the final picture of Execution Land for the `StichWorld` example. The final state of Object Land for this example is shown in Fig. 16.

Even though the `StitchWorld` methods are very simple (they take no parameters and return no results), they illustrate some important aspects of JEM diagrams and of programming in general:

- As drawn above, execution frames for buggle programs are arranged in the shape of a rightward growing **tree**. The **root** of the tree is the execution frame for the `run` method. Each execution frame has a number of **children** frames for the invocations of instance methods in the statement section of the frame. A frame is a **leaf** if it has no children frames. In the above examples, the frames for `stitch` and `turn180` are leaves of the tree.

- Certain method invocations (e.g., `forward`, `left`, and `right`) are considered primitive — we do not want or need to look at the details of how they work. It should be clear from the above diagrams that if we wanted to understand every single aspect of even a fairly simple buggle program, we would quickly be mired in a morass of details.

- The fact that a single method (such as `stitch`, `stich2`, or `turn180`) can be invoked multiple times in the same program is a source of great power. Rather than directly writing the

Figure 14: JEM after completing the second invocation of `stitch2`.

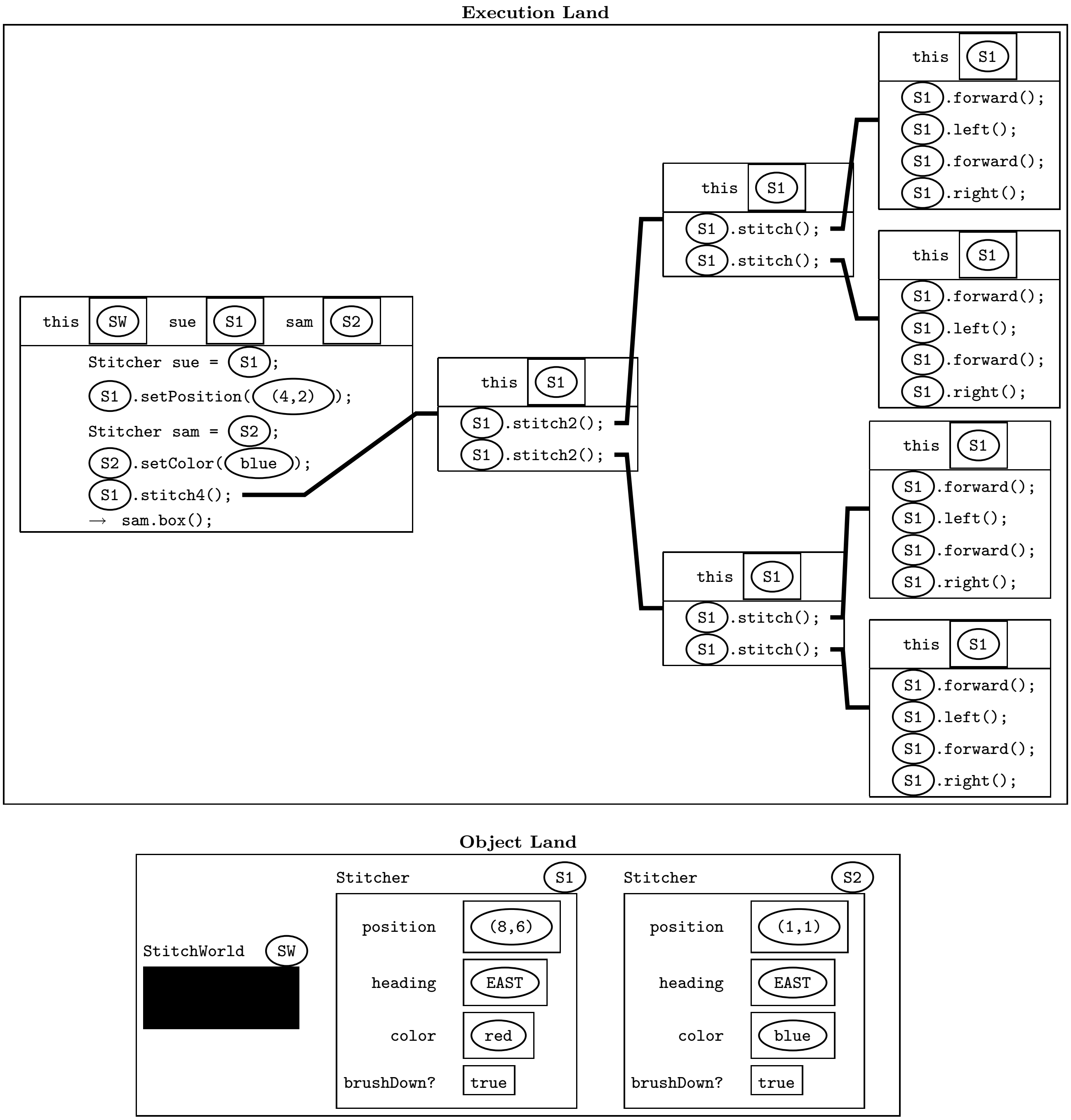


Figure 15: Final state of Execution Land in the StitchWorld example.

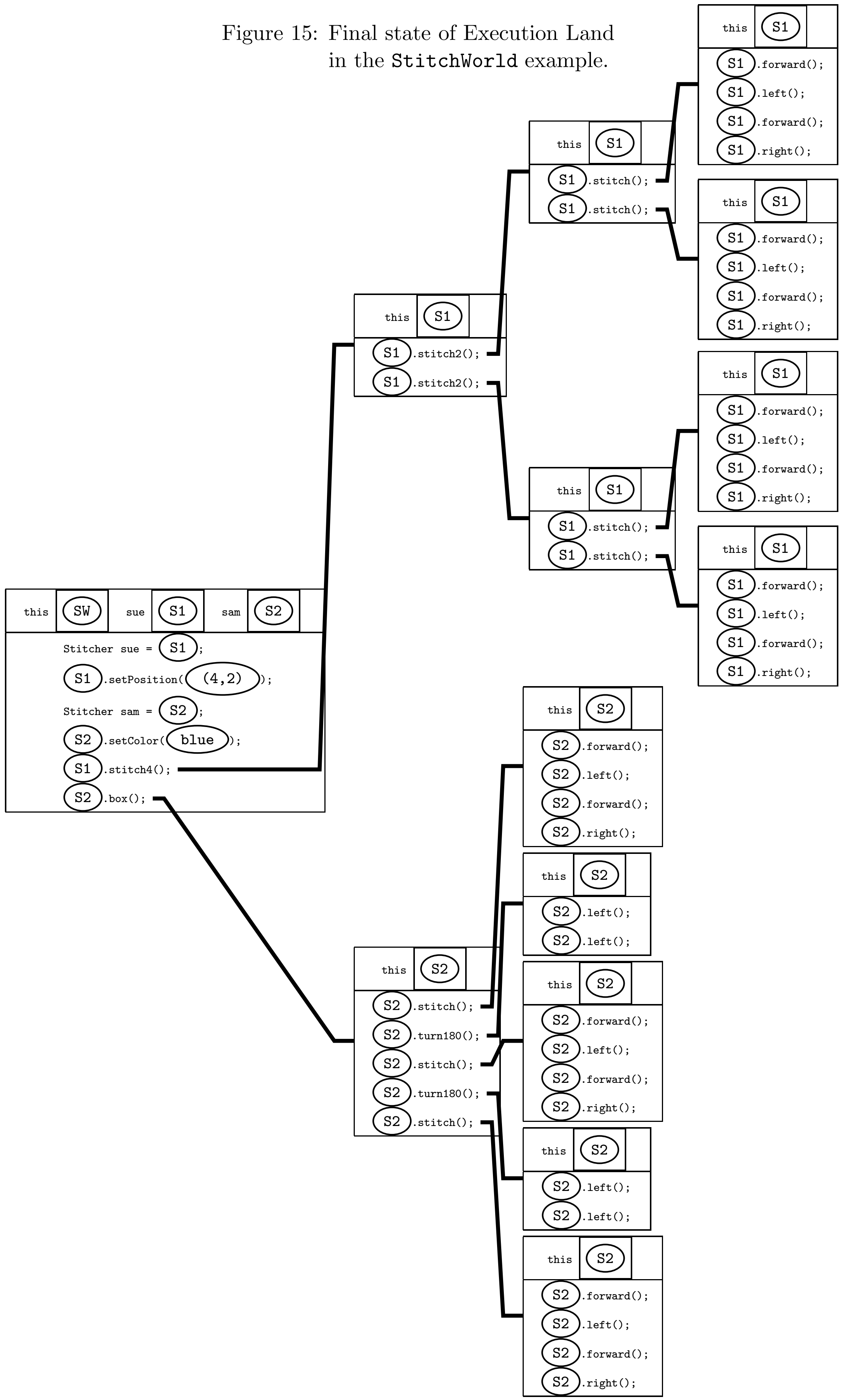
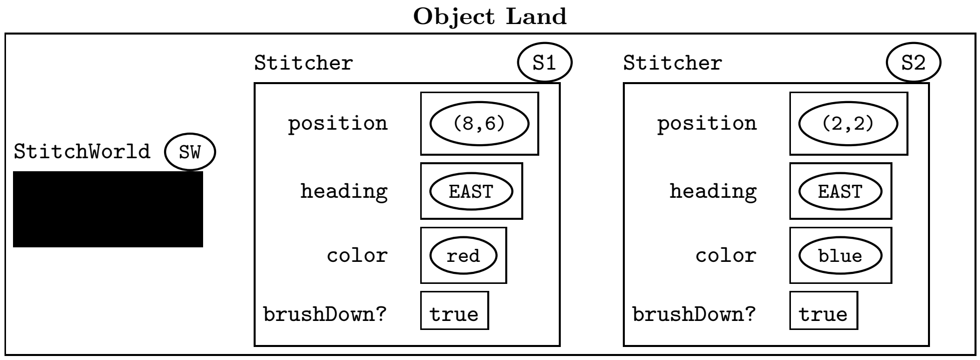


Figure 16: Final state of Object Land in the `StitchWorld` example.



sequence of statements in the leaf frames, we can instead write a few method invocations that "expand" into those statements. For instance, in the above example, each of `stitch4()` and `box()` expands into 16 primitive statements in the leaf frames. In this way, collections of methods give the programmer a way to "amplify" the power of a statement; a single statement can denote a complex sequence of actions.

- The `stitch2` and `stitch4` methods use a **successive doubling** idiom that is a simple but effective way to get a large number of leaf frames with just a few methods. It is easy to use this idiom to define 8, 16, 32, 64, etc. calls to `stitch`. We shall see this idiom many times in the course.

2 Void Methods With Parameters: LineBuggleWorld

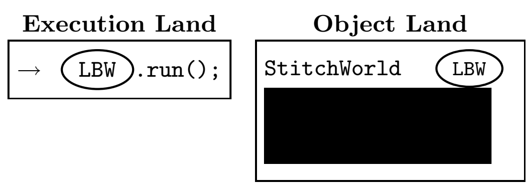
Now we consider a buggle program in which the methods take parameters (but still do not return any results). The code for `LineBuggleWorld` is presented in Fig. 17. We shall use the Java Execution Model to understand this example.

```
public class LineBuggleWorld extends BuggleWorld {

  public void run () {
    LineBuggle liam = new LineBuggle();
    liam.corner(Color.blue, Color.green, 4);
  }
}

public class LineBuggle extends Buggle {

  public void corner (Color c1, Color c2, int n) {
    this.line(c1, n+1);
    this.left();
    this.line(c2, n-1);
  }

  public void line (Color col, int len) {
    this.setColor(col);
    this.forward(len);
  }

}
```

Figure 17: Code for the `LineBuggle` example.

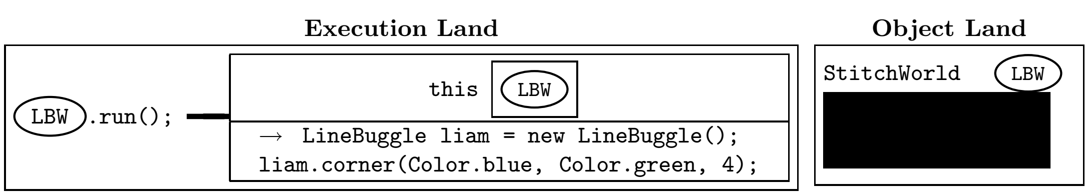
We begin with the method invocation LBW`.run()`, where LBW is assumed to be an instance of the `LineBuggleWorld` class in Object Land (Fig. 18).

Figure 18: Initial JEM diagram for the `LineBuggleWorld` example.



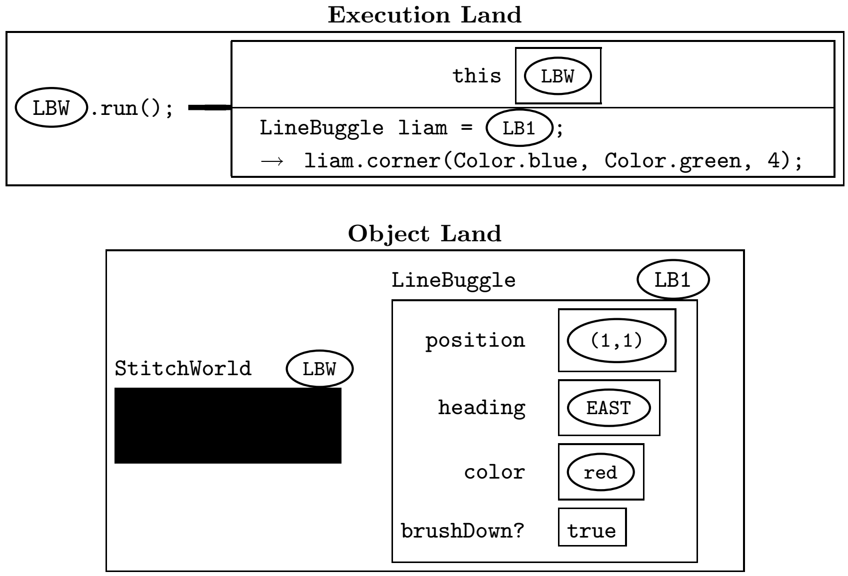
Invoking the `run` instance method on LBW creates a frame (Fig. 19).

Figure 19: Creation of the execution frame for `run`.



Executing the first statement of the `run` frame creates a new `LineBuggle` instance, which we assume has object label LB1, and stores this in a new local variable named `liam` (Fig. 20).

Figure 20: A `LineBuggle` named `liam` is born.

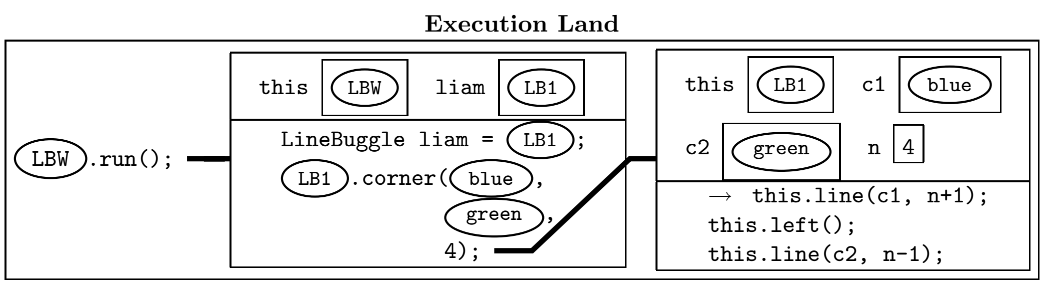


The second statement in the body of `run` is an instance method invocation. Before the method can be invoked, it is necessary to (1) evaluate the receiver expression (in this case, the variable reference `liam`) and (2) evaluate all the argument expressions (in this case, the class constants `Color.blue` and `Color.green` and the integer literal `4`). These evaluations yield the statement:

LB1 `.corner(` blue `,` green `, 4)`

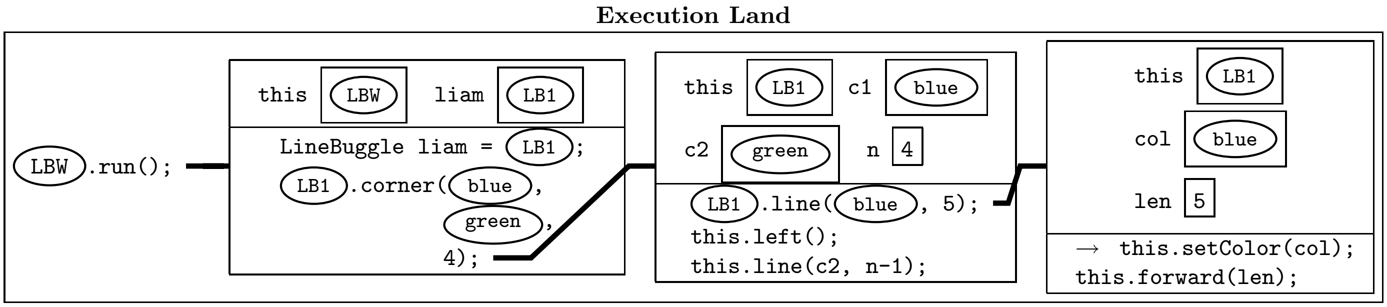
Once the receiver and argument expressions have been evaluated, the execution frame for the invocation of `corner` can be created. In addition to the `this` variable, which always contains the value of the receiver expression, the frame contains one variable for each formal parameter in the method declaration (in this case, `c1`, `c2`, and `n`). These variables are filled with the respective values of the argument expressions, as shown in Fig. 21.

Figure 21: Creation of the execution frame for the `corner` invocation.



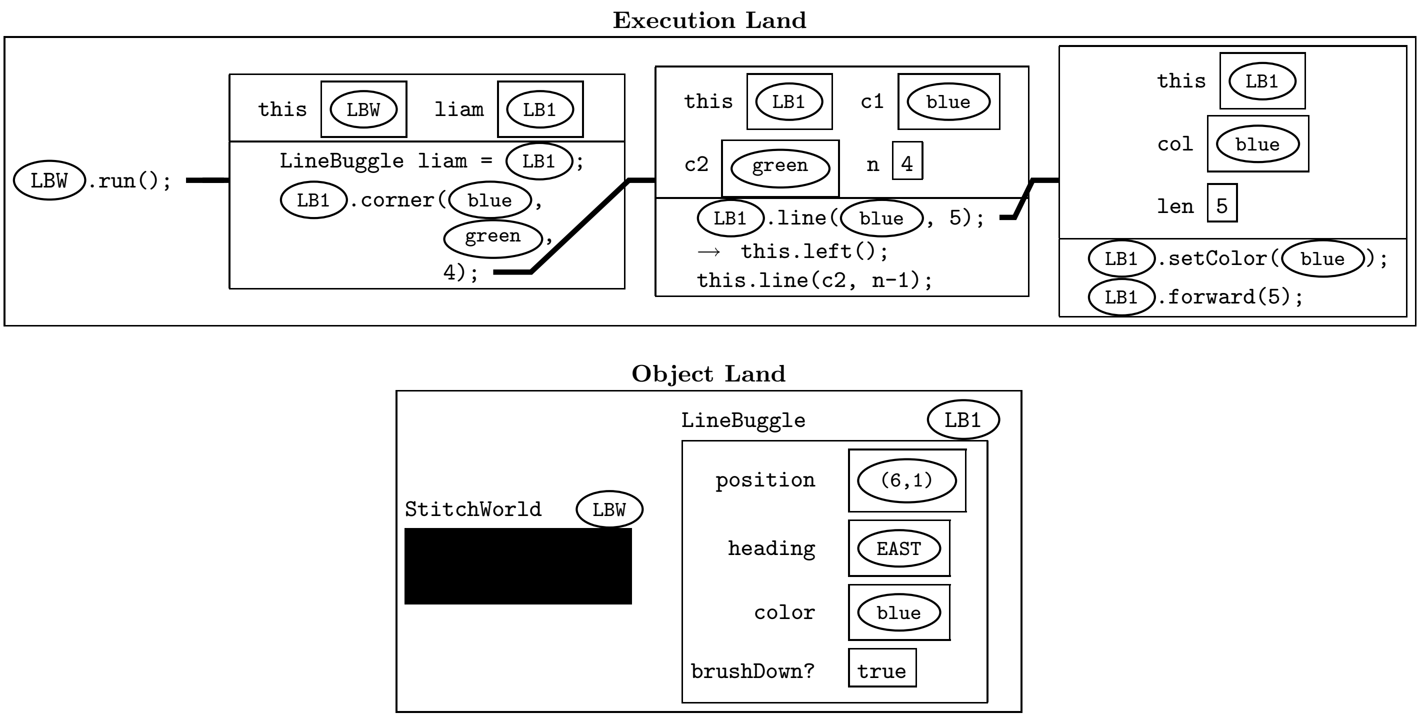
The first statement in the body of corner is an invocation of the `line` instance method. First, we need to evaluate the receiver expression, `this`, whose value is LB1. Next, we evaluate the argument expressions `c1` (whose value is blue) and `n+1` (whose value is 5). Finally, we perform the invocation LB1 `.line(` blue `,5)` by creating a new execution frame (Fig. 22). In addition to the `this` variable, the new frame has one variable for each of the formal parameters, `col` and `len`:

Figure 22: Creation of the execution frame for the first `line` invocation.



Executing the two statements in the body of the `line` method changes Execution Land and Object Land as shown in Fig. 23.

Figure 23: Creation of the execution frame for the first `line` invocation.



Next the `left` method invocation in `corner` is executed, yielding the JEM in Fig. 24.

Finally, the third statement of the `corner` method, `this.line(c2, n-1)`, is executed. To execute this instance method invocation, it is first necessary to evaluate the receiver expression (the variable reference `this`) and the argument expressions (the variable reference `c2` and the binary application `n-1`). The values of these expressions are LB1, green, and `3`, respectively. The invocation LB1`.line(`green`,3)` is performed by creating a new execution frame (Fig. 25).

Executing the two statements in the body of the second `line` frame gives the final Execution Land and Object Land for this example (Fig. 26).

The `LineBuggleWorld` example illustrates the key important fact about methods with parameters: *the variables named by the parameters enable the same method body to do different things for different method invocations.* Consider the `line` method in the above example. The two invocations

Figure 24: JEM after executing the `left` within `corner`.

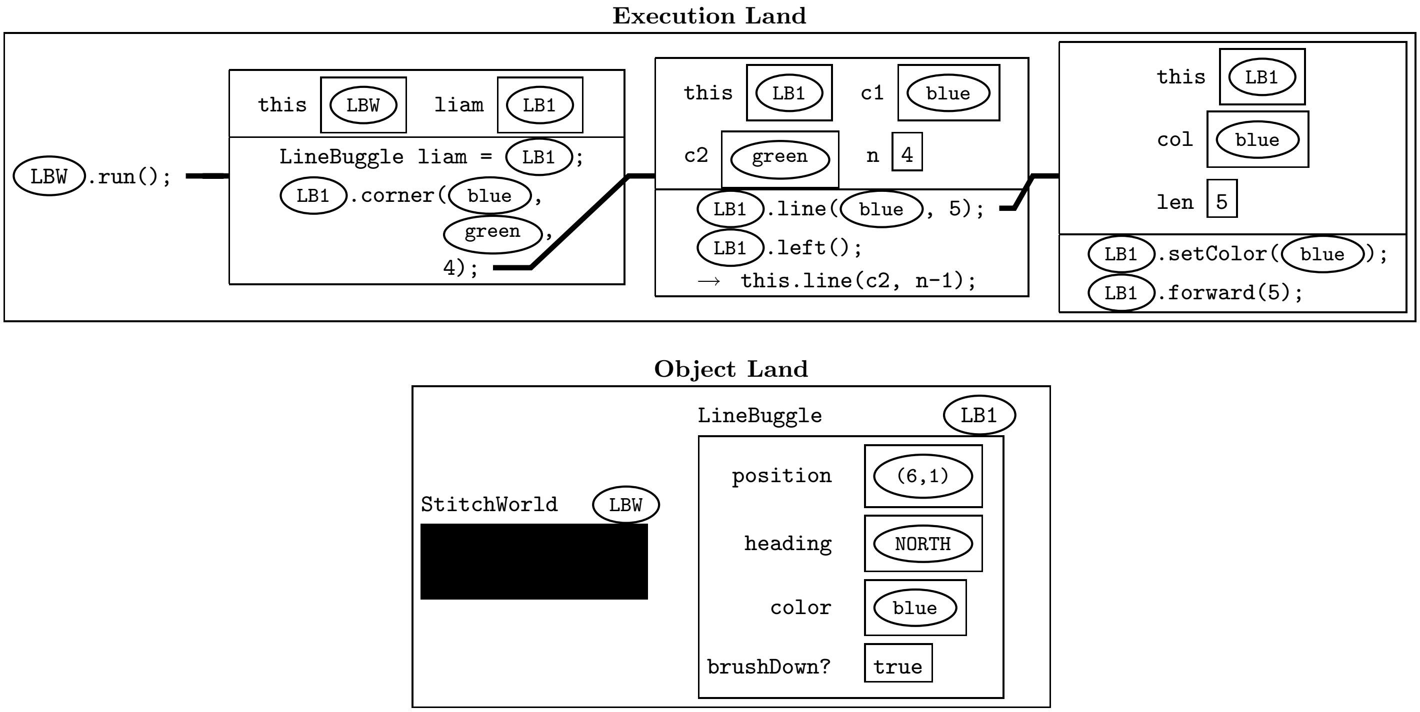


Figure 25: Creation of the execution frame for the second `line` invocation.

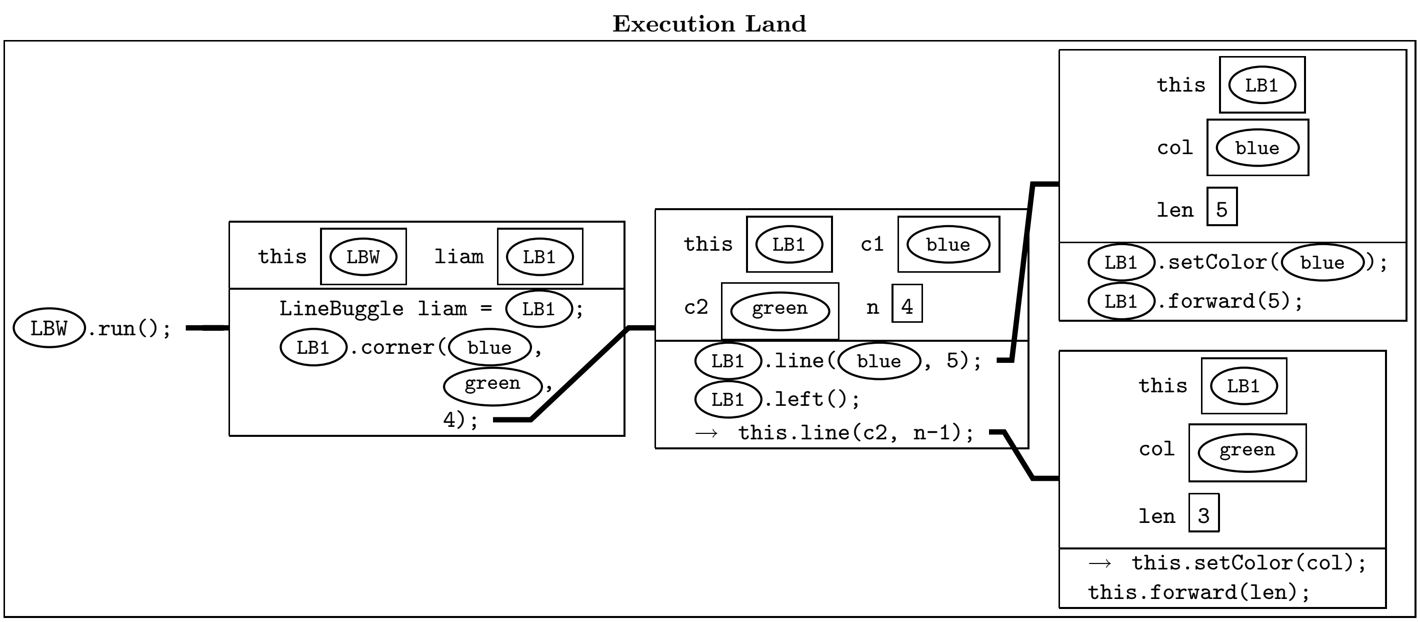
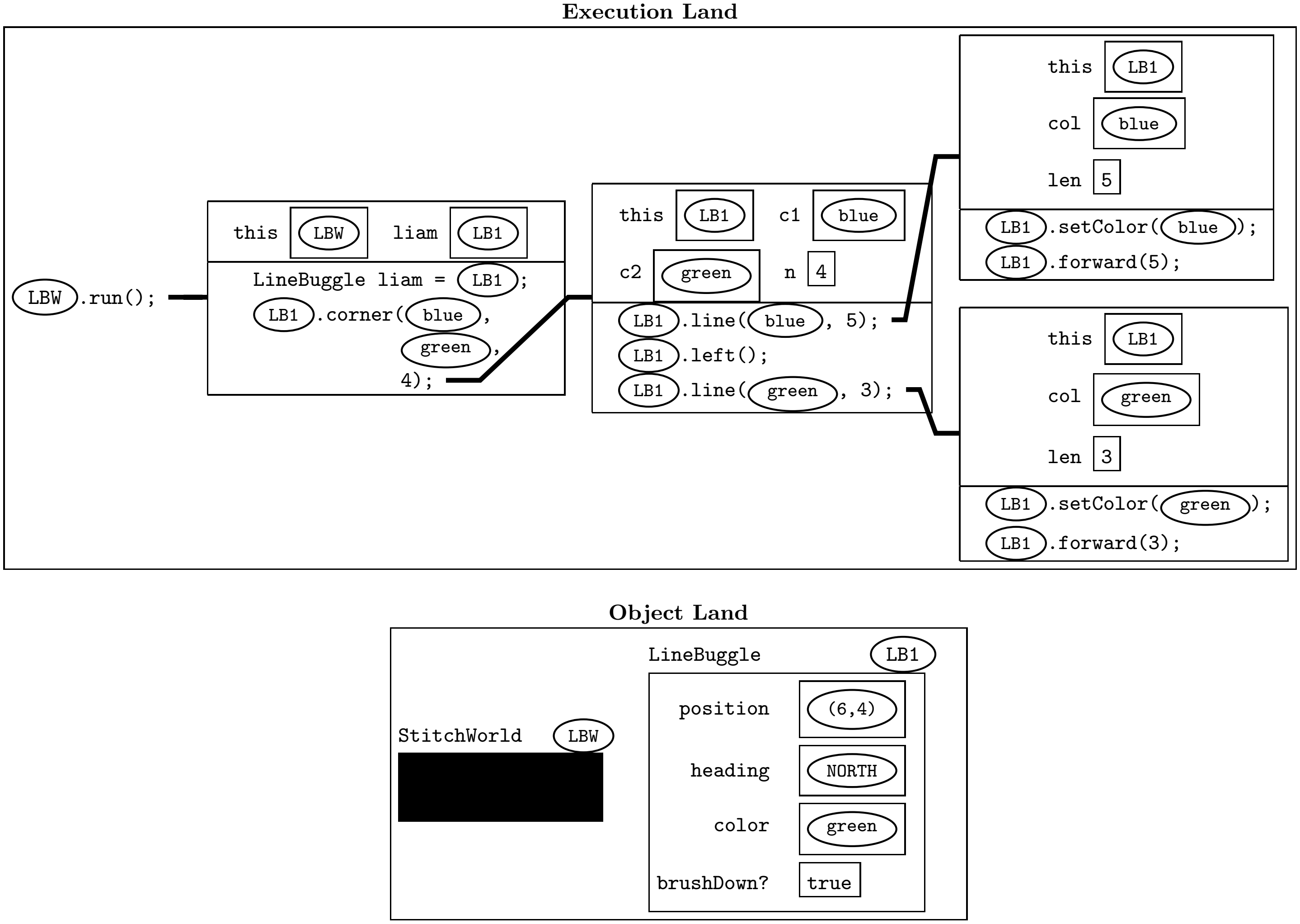


Figure 26: The final JEM diagram for the `LineBuggleWorld` example.



execute *exactly* the same two statements:

```
this.setColor(col);
this.forward(len);
```

Yet, the two invocations have different behaviors. Why? The *only* difference is values stored in the variables named by the parameters. In the first invocation, `col` denotes blue and `len` denotes `5`, while in the second invocation `col` denotes green and `len` denotes `3`. Effectively, the method body acts as a template with "holes" that can be filled in differently for different invocations; the parameters serve as these holes.

Here are a few notes concerning the above example:

- Note that the color variables (`c1`, `c2`, and `col`) are *not* written with a `Color.` in front of them. The expression `Color.c1` would denote the class constant named `c1` within the `Color` class, and, according to the contract for `Color`, there *isn't* such a constant. The only constants are actual color names, as in `Color.red` or `Color.blue`. So `Color.c1` is an error that will be caught by the Java compiler.
- The execution diagram makes it clear that there are two *different* variables named `col` and two *different* variables named `len` — one for each of the two execution frames for `line`.

Although people sometimes get confused by the presence of more than one variable with the same name, Java does not, because it follows a simple rule: when executing the statements in an execution frame, all variable names must refer to the variables *at the top of that frame*. For instance, each of the `line` frames can refer to its own `this`, `col`, and `len` variables, but neither can refer to the `this`, `col`, and `len` variables in the other frame, nor can either refer to the `this`, `c1`, `c2`, and `n` variables in the corner frame. In terms of information flow, this means that if you want a method to use an existing object, you must pass it in as a parameter.[1]

- Because variable names are completely local, they can be consistently renamed without affecting the computation. For instance, we can rename `col` and `len` to be any other two names we want, as long as they are different and not the special name `this`. For instance, we could redefine `line` to be any of the following:

```
public void line (Color color, int length) {
  this.setColor(color);
  this.forward(length);
}

public void line (Color c, int l) {
  this.setColor(c);
  this.forward(l);
}

public void line (Color foogle, int blarg) {
  this.setColor(foogle);
  this.forward(blarg);
}

public void line (Color len, int col) {
  this.setColor(len);
  this.forward(col);
}

public void line (Color c1, int n) {
  this.setColor(c1);
  this.forward(n);
}
```

  Java will treat all of the above in *exactly* the same way, modulo changing the names of variables in the diagrams. This is even true in the last case, where `line` is "reusing" the same parameter names used in `corner`. While human programmers might find this confusing, Java is not confused, as is illustrated by the corresponding Execution Land for this case (Fig. 27). Note that there are now *three* distinct variables named `c1` and `n`, but that this does not change the computation in any way.

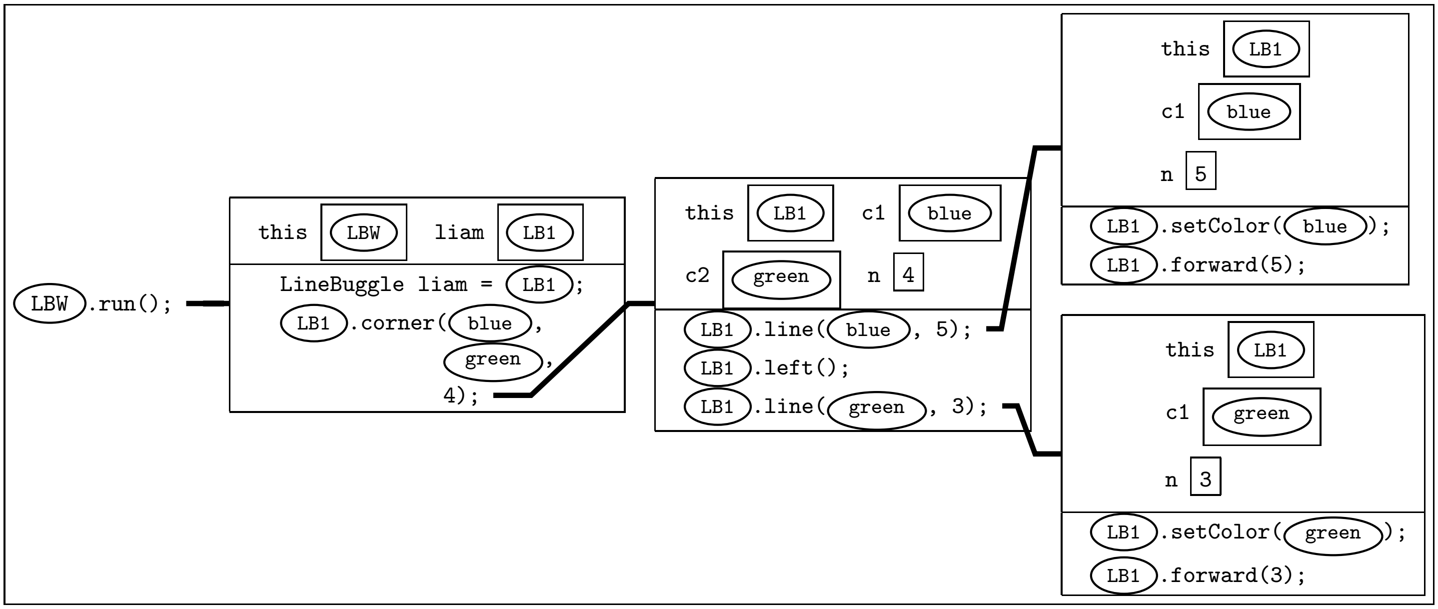
- Ideally, parameter names should serve as comments that indicate the type and purpose of the parameter. From the point of readability, some choices (such as `col`/`len` or `color`/`length`) are better than others. Names like `foogle`/`blarg` may seem funny, but are very unhelpful. Perverse names like `len` for the color variable and `col` for the length variable are particularly confusing for the human programmer.

- Sometimes, new Java programmers try to "set" the values of a method parameter using a local variable declaration, as shown below:

[1] This isn't technically true, since all methods can refer to the same class constants. And we will learn later how objects can be shared through instance variables. But the intuition that objects are shared by passing them as parameters is a good one.

Figure 27: The final JEM diagram for the `LineBuggleWorld` example using `c1` and `n` as the parameter names for the `line` method.

Execution Land




```
// An incorrect way to pass arguments to a method
Color col = Color.blue;
int len = 5;
this.line();
```

The Java Execution Model explains why this does not work. This defines variables named `col` and `int` in the execution frame from which `line` is invoked, and *not* in the execution frame for `line` itself. In fact, since `line` expects two arguments and is not passed any, the Java compiler will complain about the above code.

The above attempt can be repaired as follows:

```
// A working but not very good way to pass arguments to a method
Color col = Color.blue;
int len = 5;
this.line(col, len);
```

This actually works, since the values stored in the variables `col` and `len` in the current execution frame will be passed to the execution frame for `line`. However, this code is clumsy, and suggests that the programmer does not understand how parameter passing works in Java.

The best way to pass parameters is to put them directly in the argument positions of an invocation:

```
// The best way to pass arguments to a method
this.line(Color.blue, 5);
```