Fruitful recursion

Recursion with methods that return values

Friday, October 20, 2006

CS111 Computer Programming

Department of Computer Science
Wellesley College

Recursion

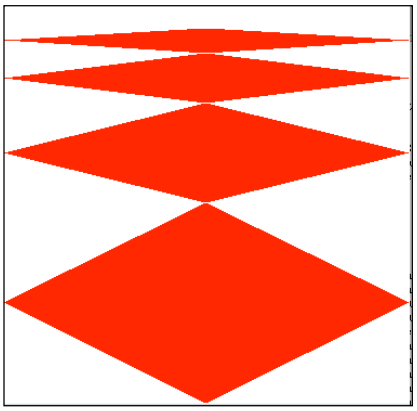
- Recursion is a solution technique that is effective when a problem can be divided into smaller subproblems of the same shape.
- First, we studied cases with no parameters and no returned values.
- Then we studied recursive methods with parameters.
- Today we study recursive methods that return values (and may also take parameters).

```
bagelsToWall();
```

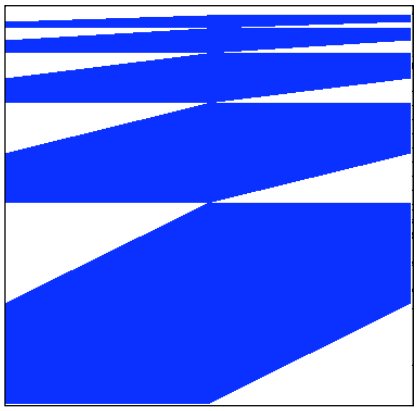
```
tree(4,20,40,0.7);
```

```
cornerPush(4,di);
```

Recursive Pictures: upPush()



`upPush(4,di)`



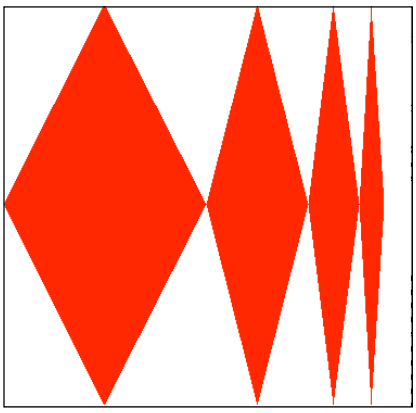
`upPush(5,hex)`

Designing upPush() with an Invocation Tree

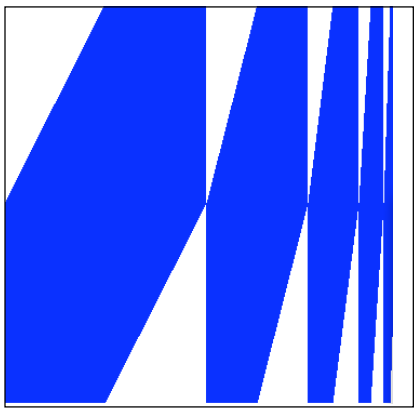
Let's draw an invocation tree for upPush(3,di):

Defining upPush() in Java

Recursive Pictures: rightPush()



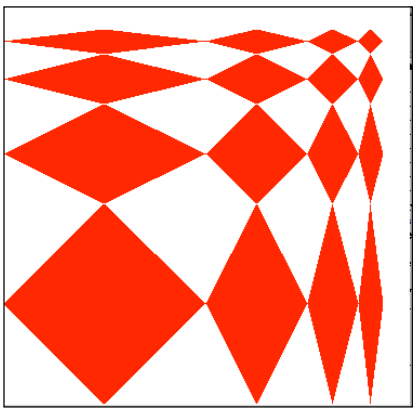
`rightPush(4,di)`



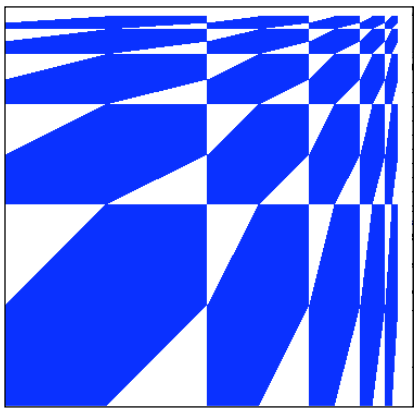
`rightPush(5,hex)`

Defining rightPush() in Java

Recursive Pictures: cornerPush()



`cornerPush(4,di)`



`cornerPush(5,hex)`

Defining cornerPush() in Java

DistanceWorld

Diana buggle wants to find the number of steps to the wall.

- **What is the base case for this problem?**

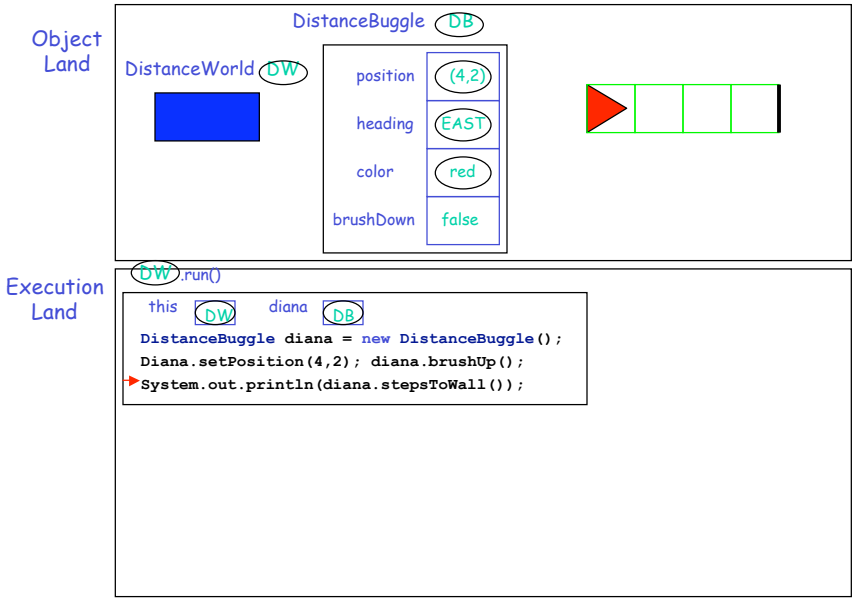
- **In the general case:**
  (1) How is the problem made smaller?

  (2) How is the solution to the whole problem obtained from the solution to the smaller problem?

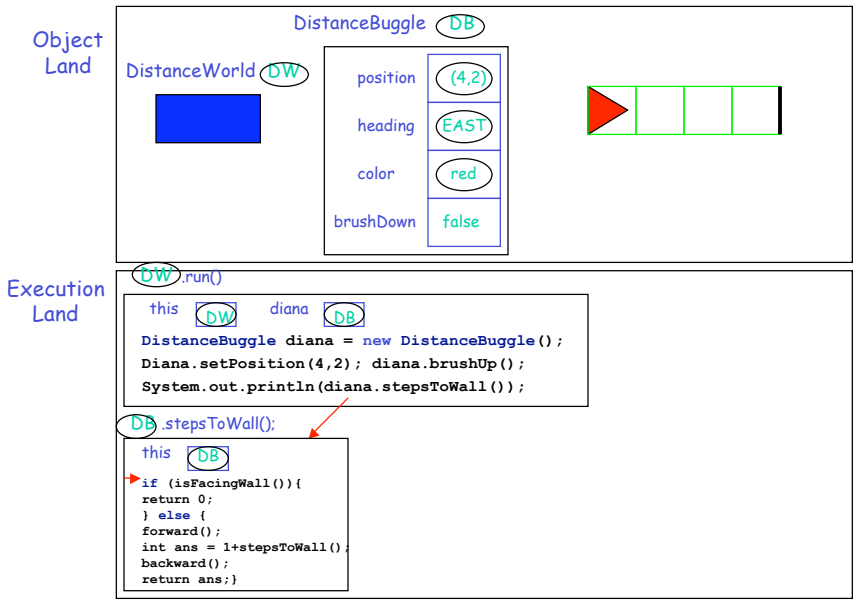
Implementing stepsToWall() in Java

```
// Returns the number of cells between this buggle and the wall it is facing,
// *excluding* the cell it is in.
// Maintains position, heading, color, and brush state of buggle as invariants.
public int stepsToWall() {




}
```

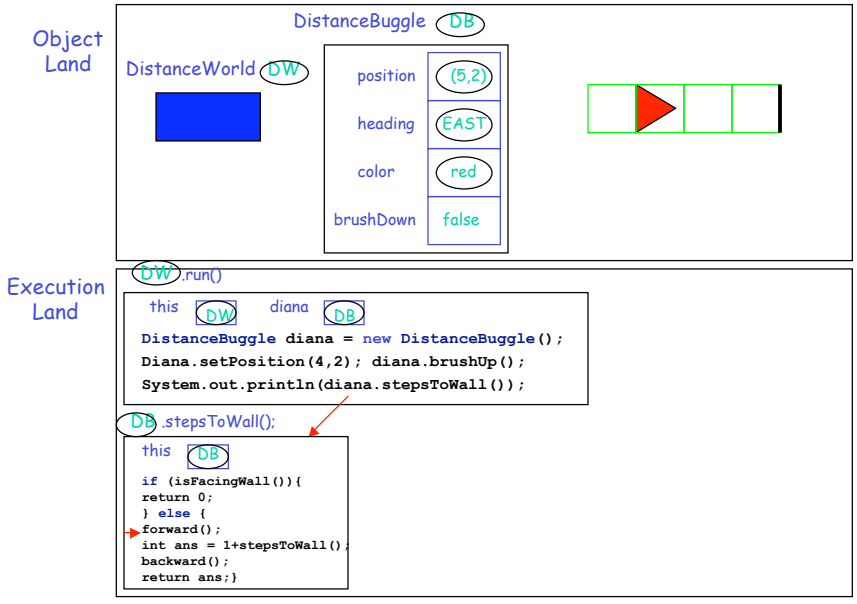

Sample JEM (Joined in Progress)



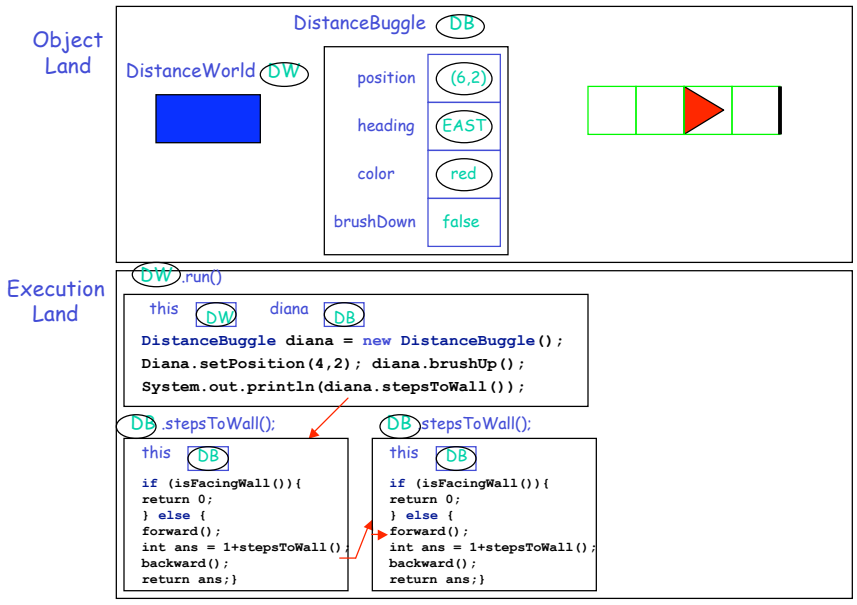
The First Invocation



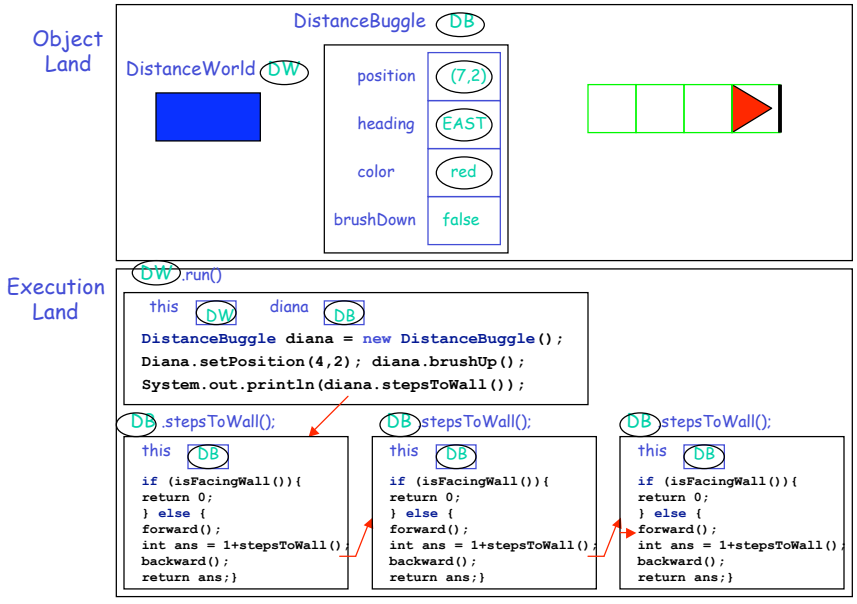
Making the Problem Smaller



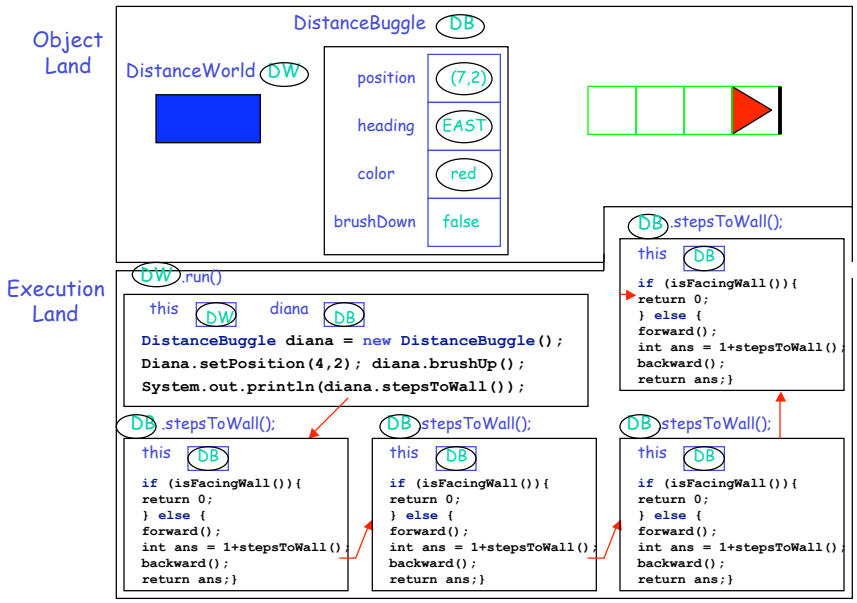
Smaller Again



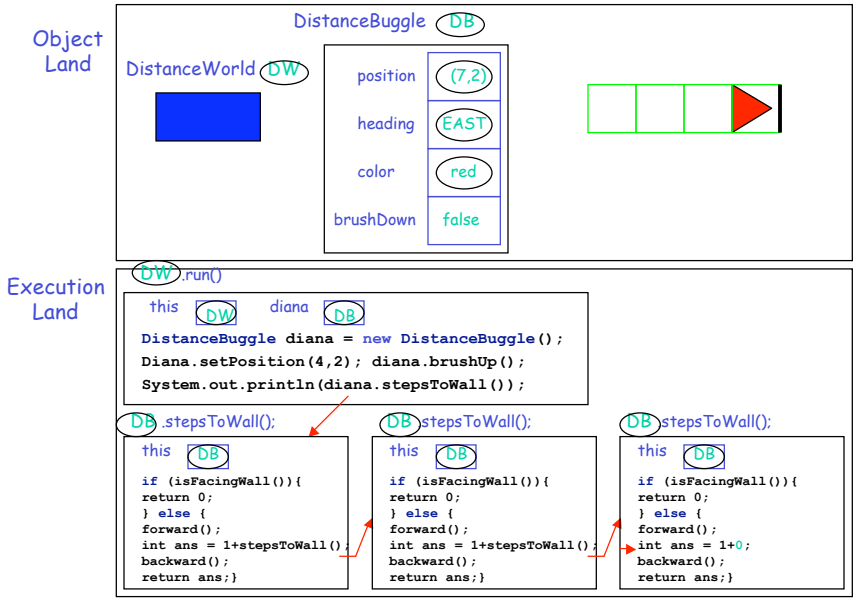
Even Smaller



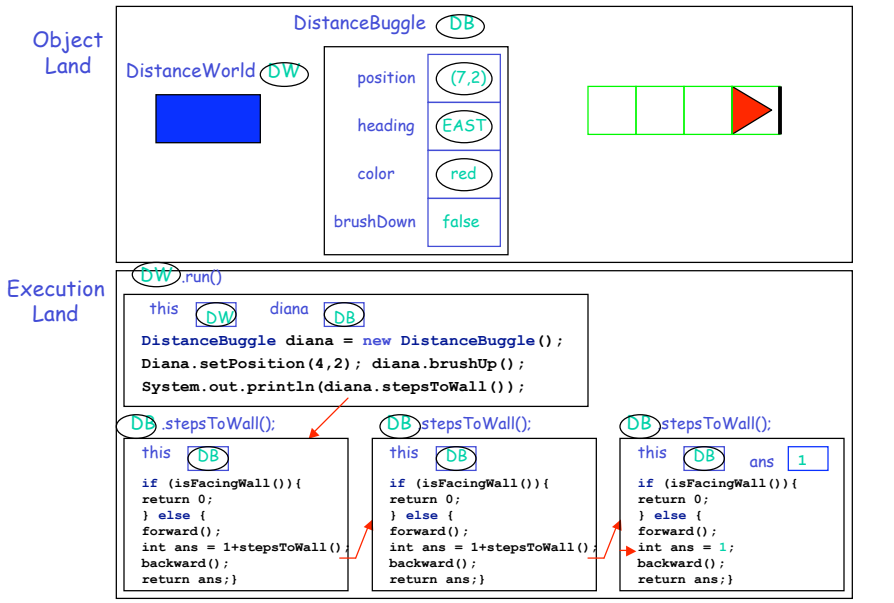
Reaching the Base Case!



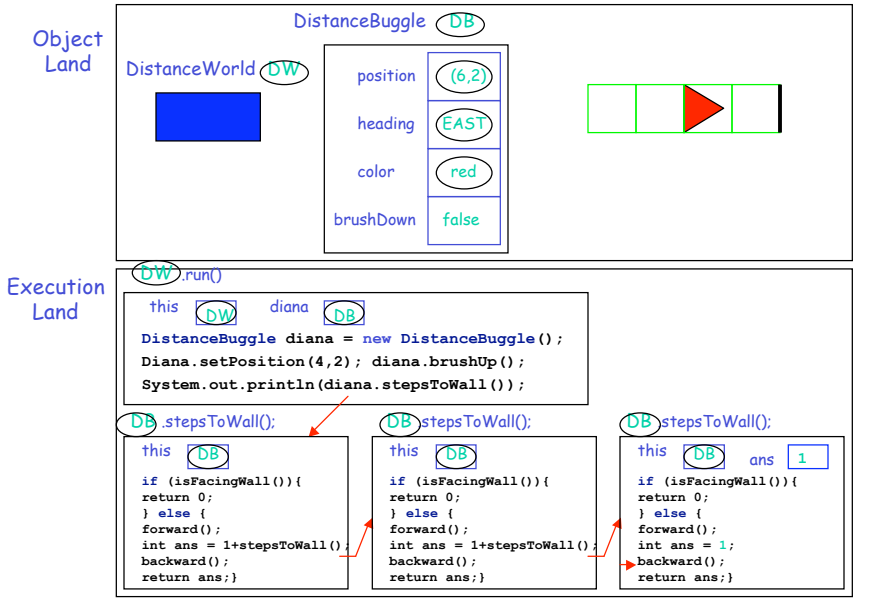
Returning From the Base Case



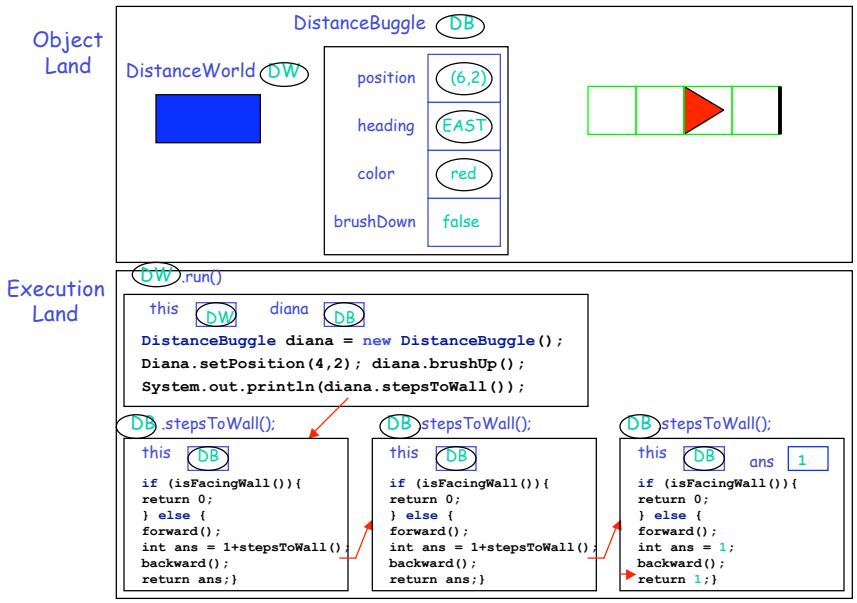
Saving the Answer



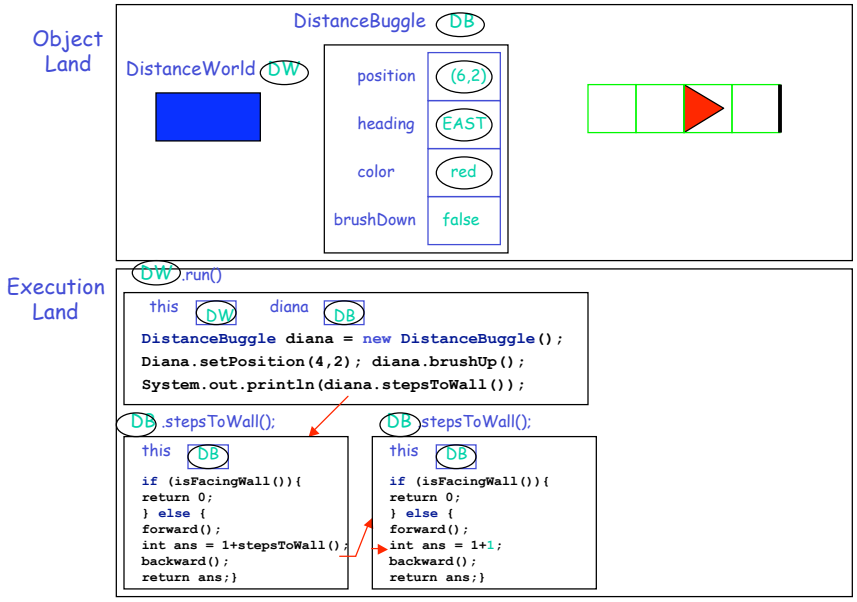
Backing Up



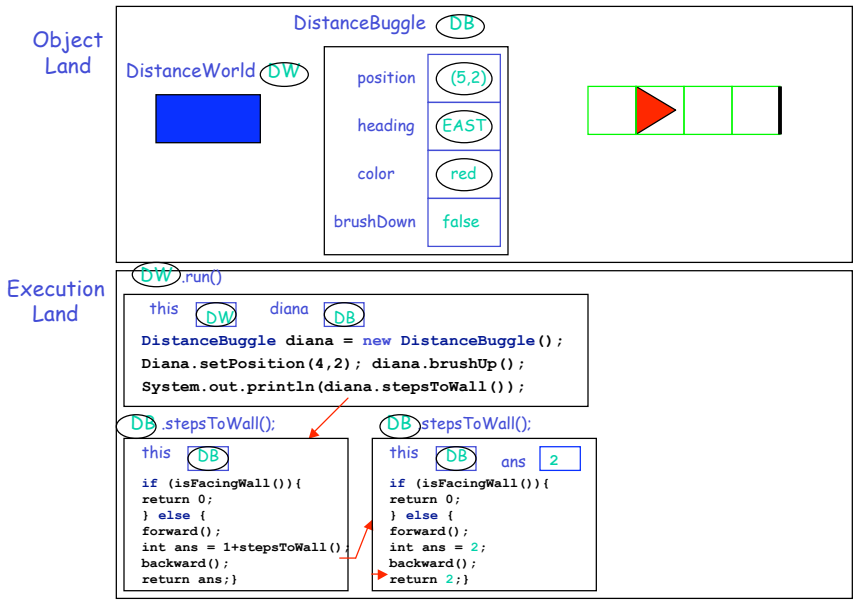
About to Return from the Penultimate Case



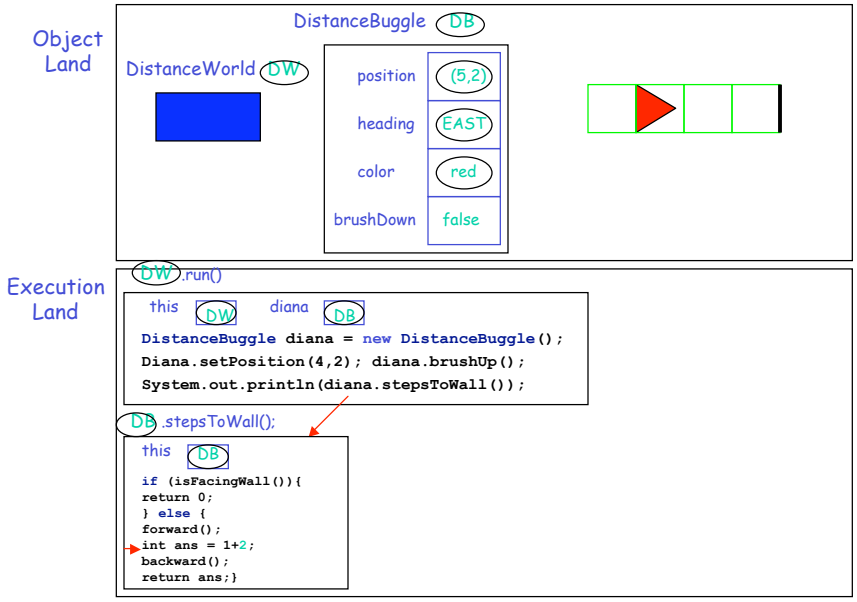
The Story Continues



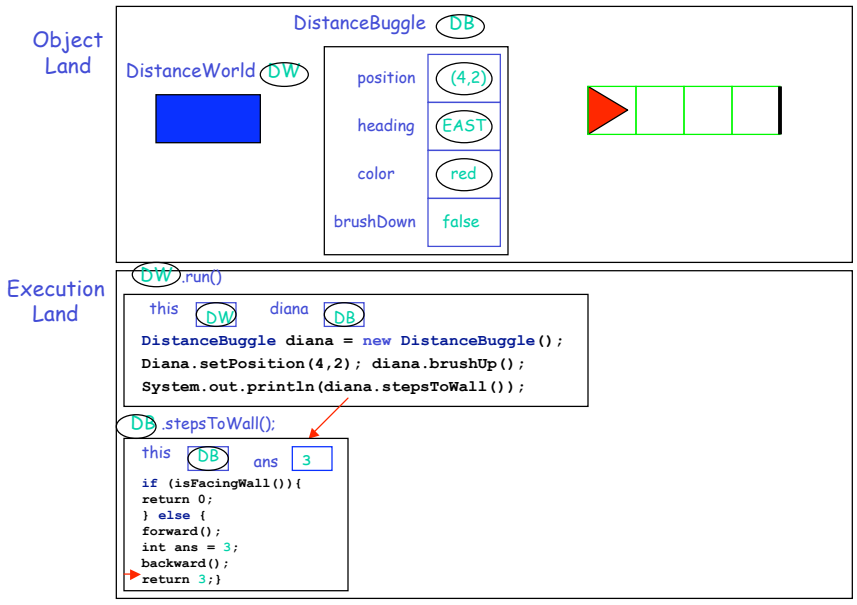
About to Return Again



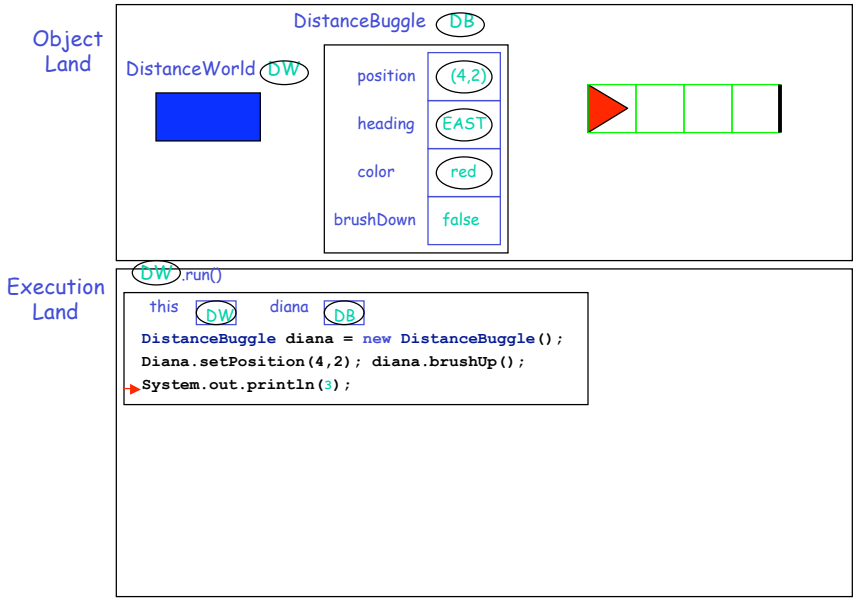
Another Addition



The Final Return



The Answer can be Displayed



Which of the Following Work?

```
public int stepsToWall2() {
  if (isFacingWall()) {
    return 0;
  } else {
    forward();
    int subSteps = stepsToWall2();
    backward();
    return 1+subSteps;
  }
}
```

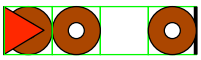
```
public int stepsToWall3() {
  int n = 0;
  if (!isFacingWall()) {
    forward();
    n = 1+stepsToWall3();
    backward();
  }
  return n;
}
```

```
public int stepsToWall4() {
  int n = 0;
  if (!isFacingWall()) {
    forward();
    n = n + 1;
    stepsToWall4();
    backward();
  }
  return n;
}
```

```
public int n = 0; // extra instance variable

public int stepsToWall5() {
  if (!isFacingWall()) {
    forward();
    n = n + 1;
    stepsToWall5();
    backward();
  }
  return n;
}
```


BagelCountingWorld

Suppose a buggle wants to count the number of bagels between it and the wall.



Does it count the bagel under it or not?
We'll consider both choices:

```
public int bagelsInFront();
```
Returns the number of bagels between this buggle and the wall it is facing, *excluding* any bagel that might be under it. Maintains position, heading, color, and brush state of buggle as invariants.

```
public int bagelsUnderAndInFront();
```
Returns the number of bagels between this buggle and the wall it is facing, *including* any bagel that might be under it. Maintains position, heading, color, and brush state of buggle as invariants.

Both solutions are similar to stepsToWall(). Here's a useful helper method:

```
public int bagelsUnder() {
  if (isOverBagel()) return 1; else return 0;
}
```

Defining bagelsInFront()

```
public int bagelsInFront() {

}
```


Defining bagelsUnderAndInFront()

```
public int bagelsUnderAndInFront() {

}
```