# Fruitful recursion

## Recursion with methods that return values

Tuesday, October 16, 2007

**CS111 Computer Programming**

Department of Computer Science
Wellesley College

## A New Wrinkle on 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.

```
bagelsToWall();
```

- Then we studied recursive methods with parameters.

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

- Today we study recursion using both parameters and returned values. We'll see examples from BuggleWorld, TurtleWorld, and PictureWorld.

```
triangleDesign(4,red,blue);
```

# DistanceWorld

Diana buggle wants to find the number of steps to the wall and go back to her original position without leaving a trail.

For simplicity, assume Diana's brush is up before she starts.

```
public class DistanceWorld extends BuggleWorld {
   public void run() {
      DistanceBuggle diana = new DistanceBuggle();
      diana.brushUp();
      System.out.println(diana.stepsToWall());
   }
}
public class DistanceBuggle extends Buggle {
   public int stepsToWall() { … }
}
```



# stepsToWall(): Case Analysis

o What is the base case for this problem?

o In the general case:
(1) **Divide:** how is the problem made smaller?

(2) **Conquer:** how are the smaller problems solved?

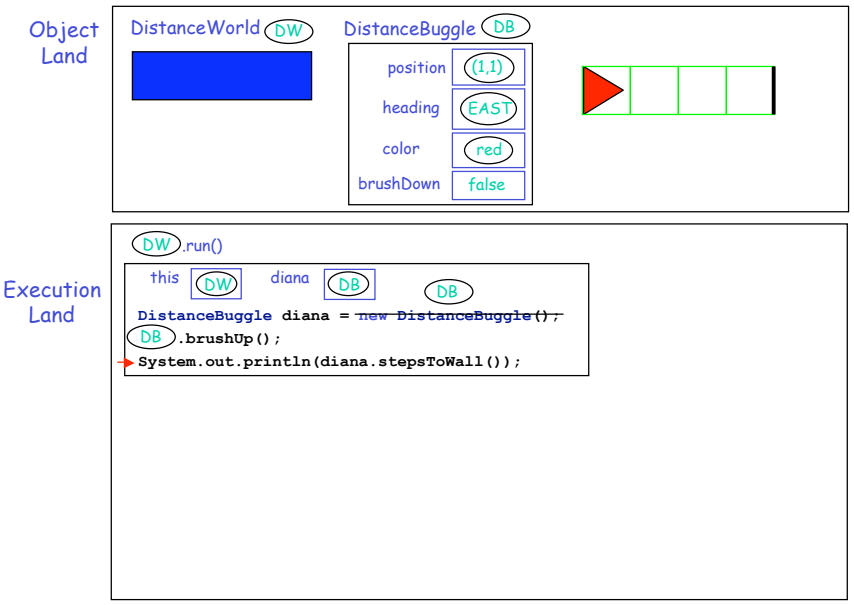
(3) **Glue:** how is the solution to the whole problem obtained from the solution(s) to the smaller problem(s)?

# Implementing stepsToWall() in Java

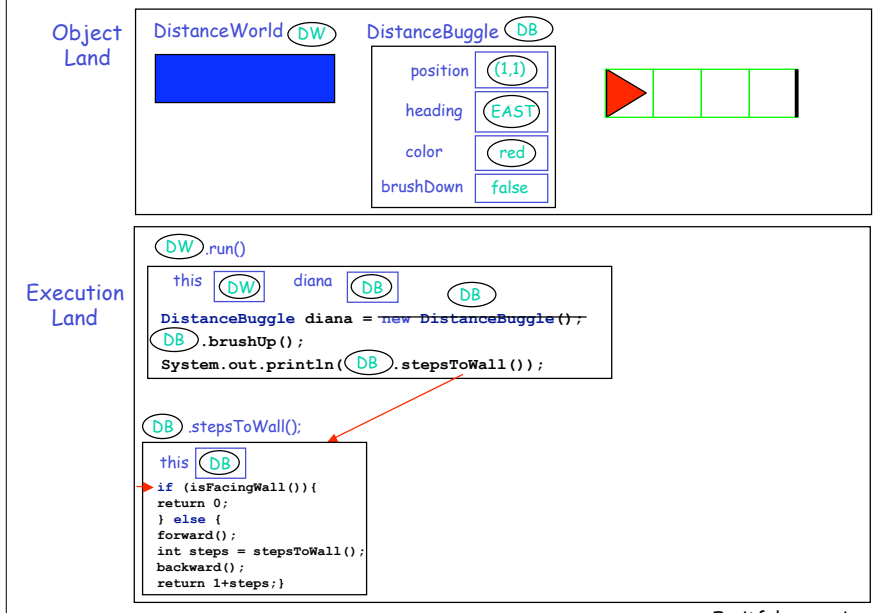
```
// Returns the number of cells between this buggle and the wall
// it is facing, *excluding* the cell it is in.
// Maintains position, heading, color, and brush state of
// buggle as invariants.
public int stepsToWall() {




}
```



# Sample JEM (Joined in Progress)

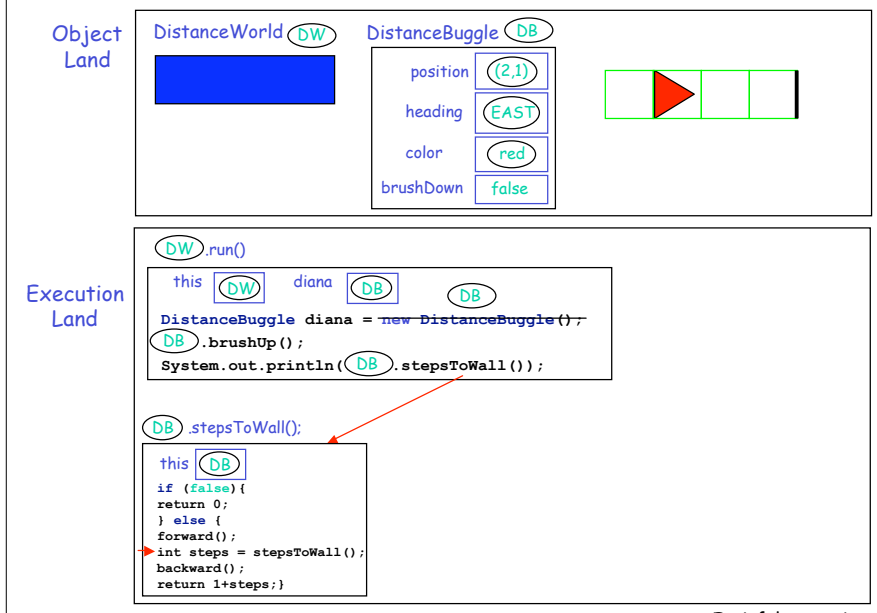


## The First Invocation

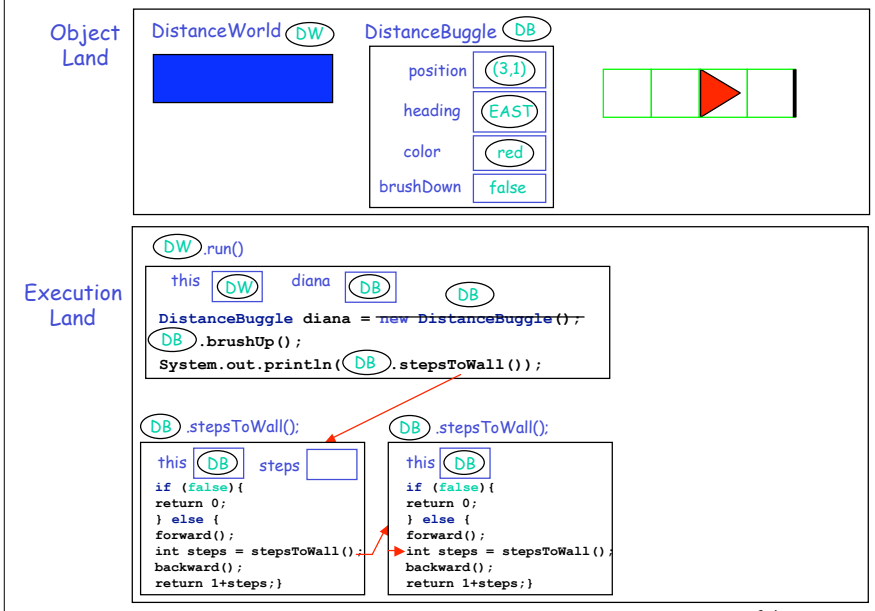





## Making the Problem Smaller

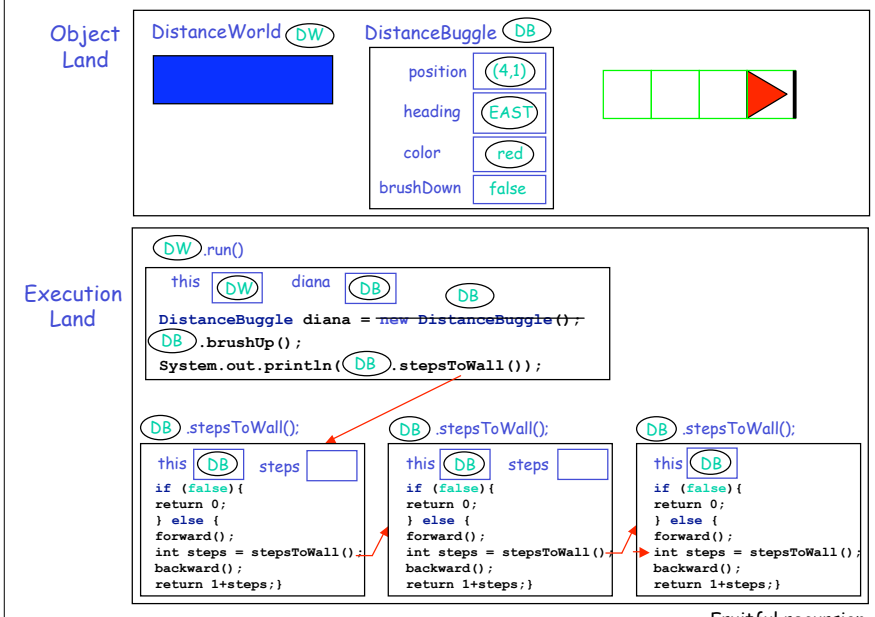


## Smaller Again

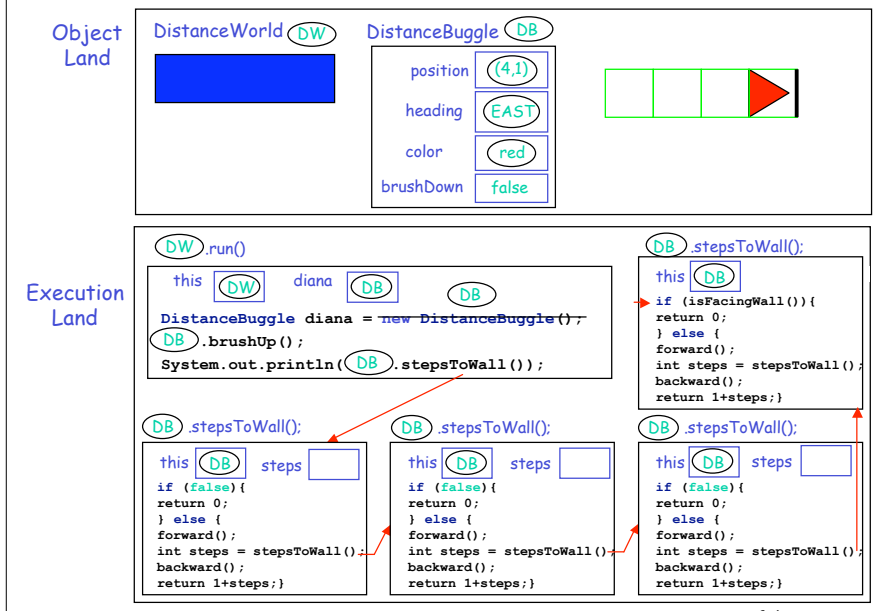





## Even Smaller

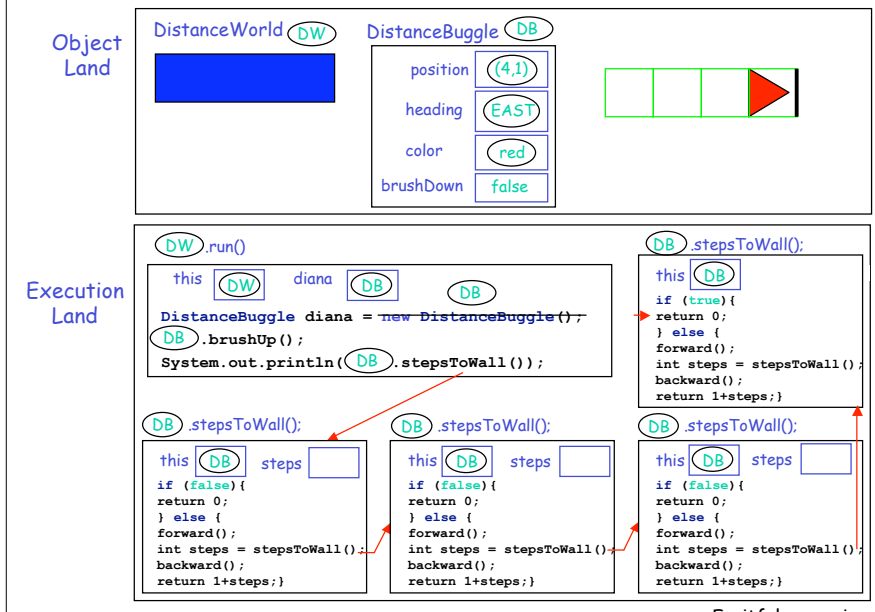


## Reaching the Base Case!

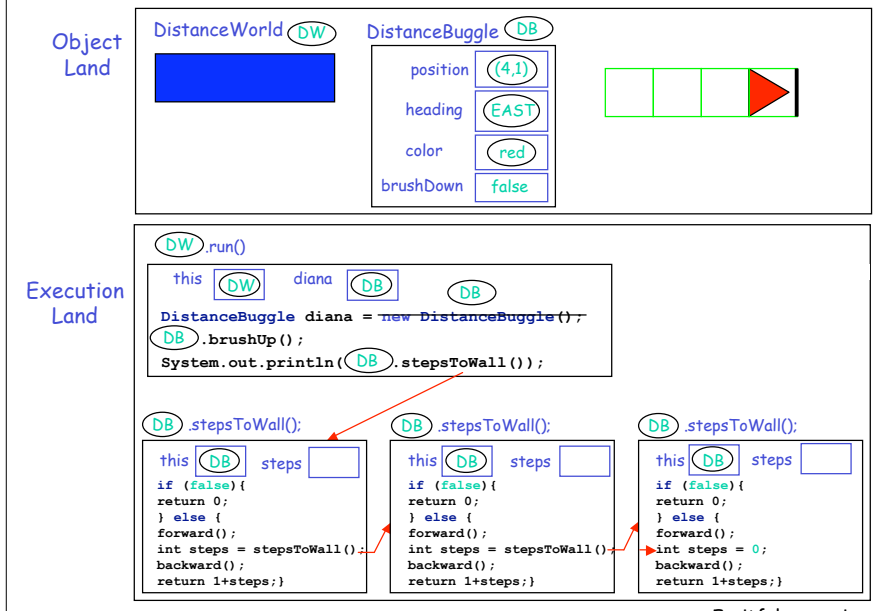





## Executing the True Clause in the Base Case

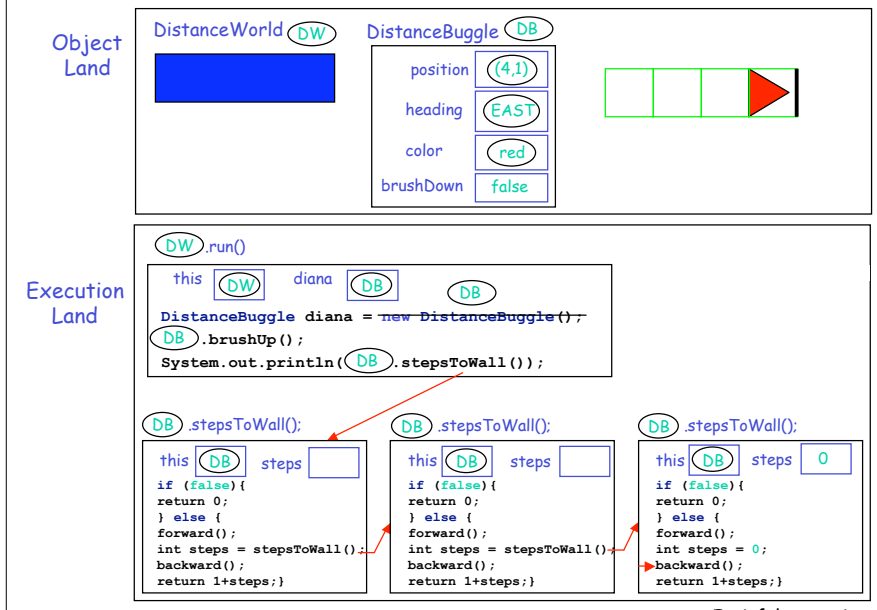


## Returning From the Base Case

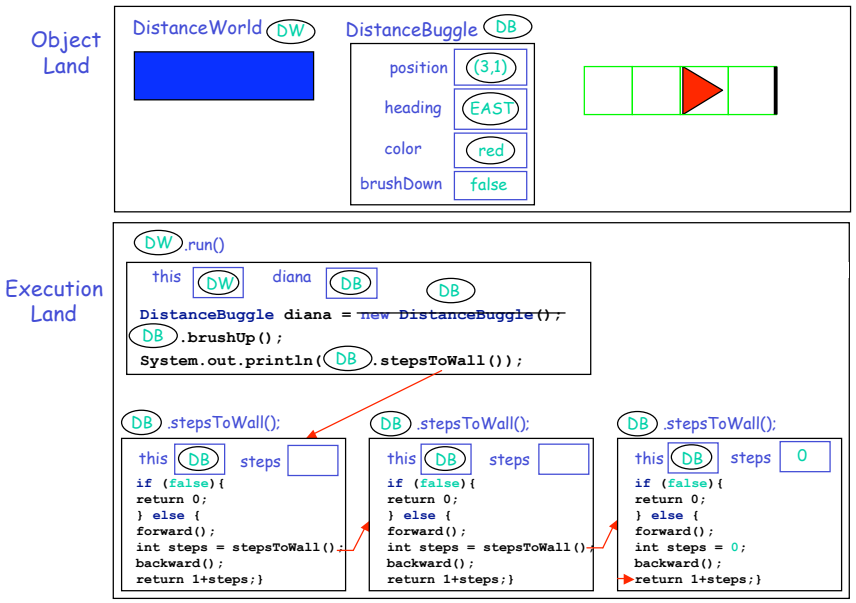





## Saving the Solution to the Subproblem

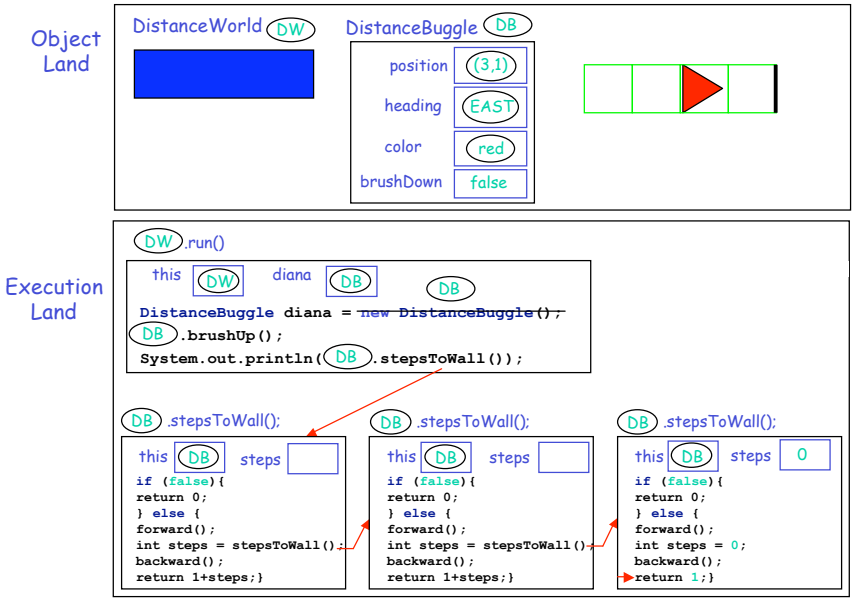


## Backing Up

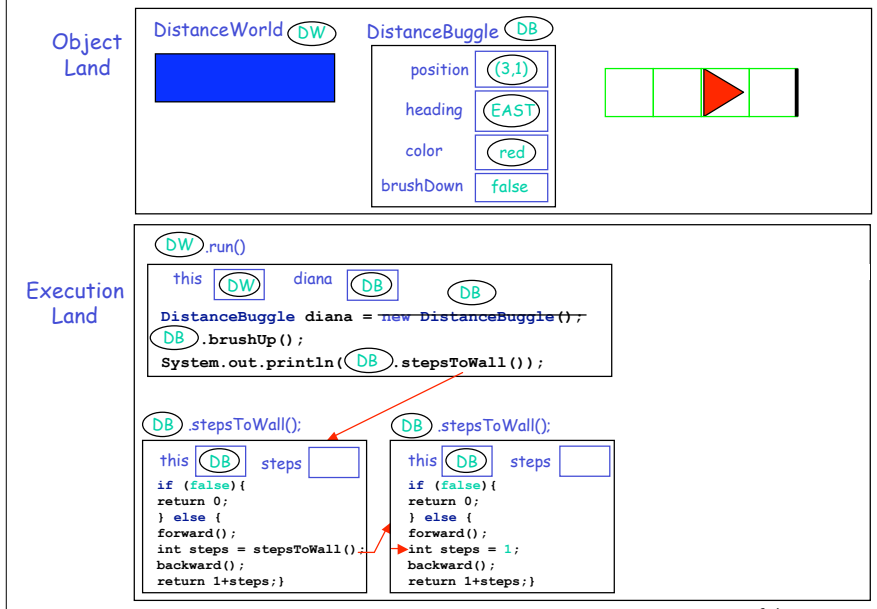





## About to Return From the Penultimate Case

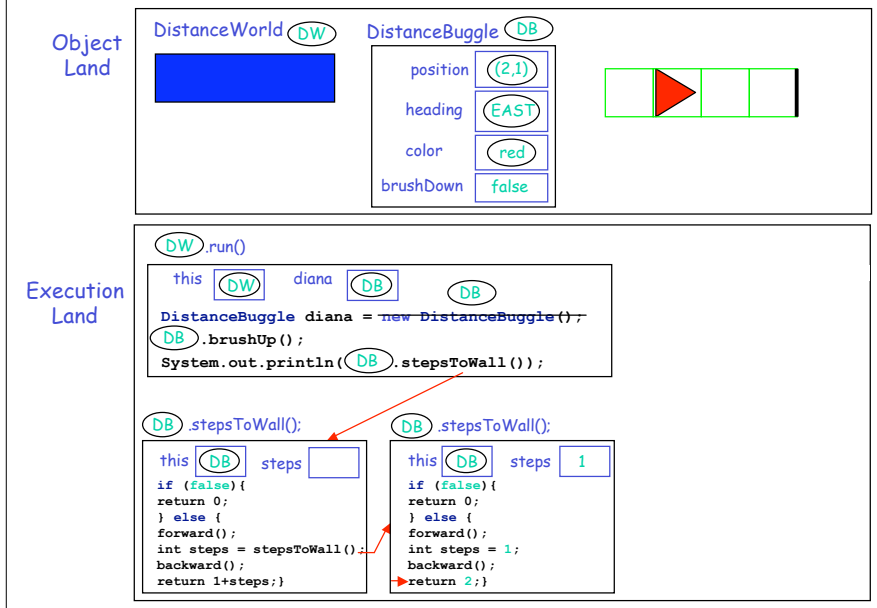


## The Story Continues

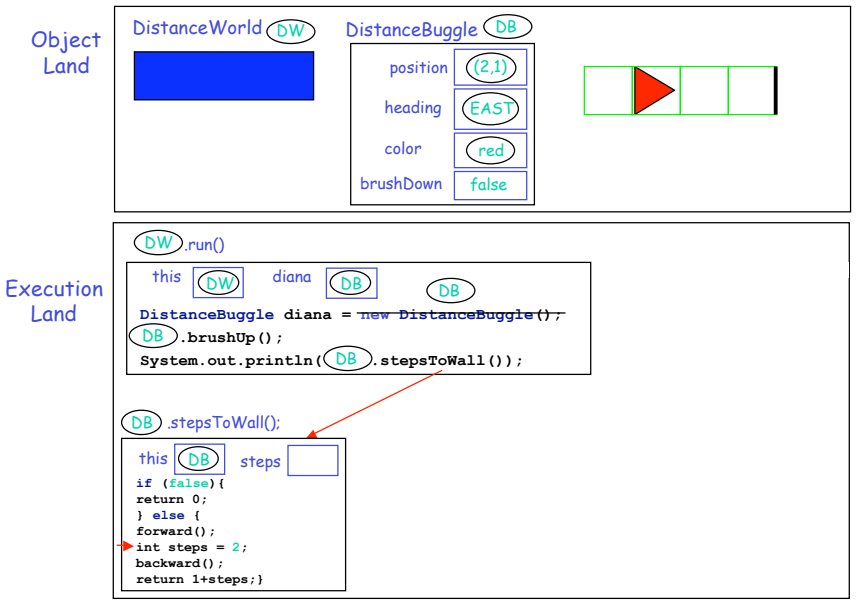





## About to Return Again

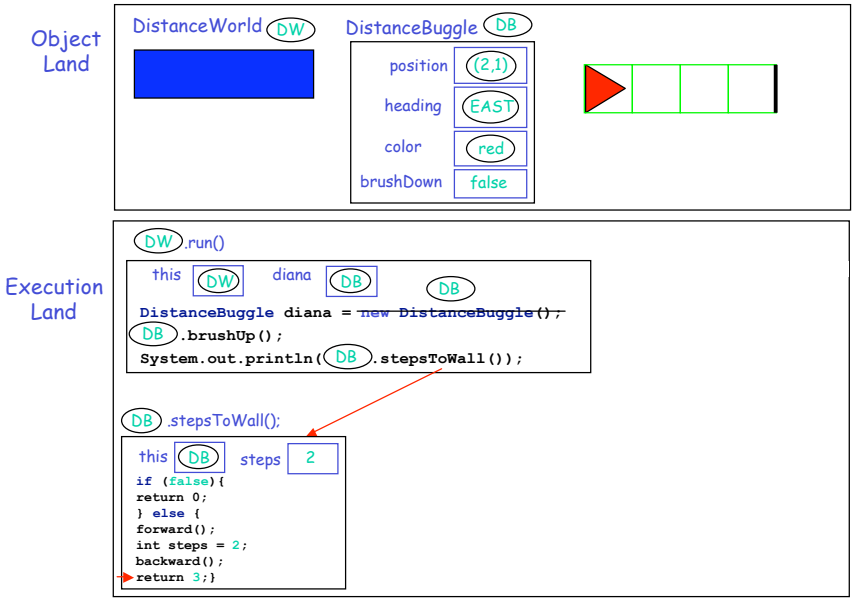


## Saving Another Subproblem Solution

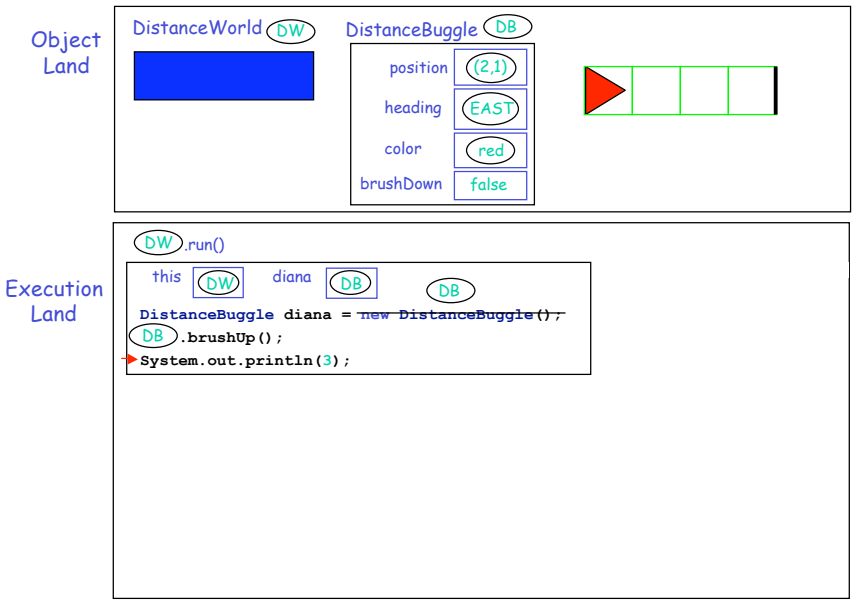





## The Final Return



## About to Display the Answer



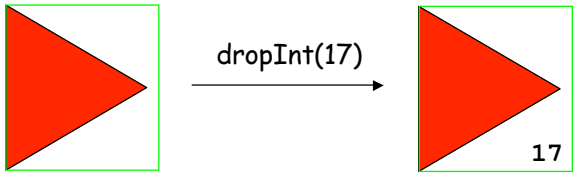



## dropInt(int n)

The dropInt() buggle method is useful for debugging buggles that count.

public int dropInt(int n);
Drops the integer n in the current cell and returns this integer.

Note that dropInt() both **performs an action** (drops an integer) and **returns a value** (the integer dropped).



# dropStepsToWall()

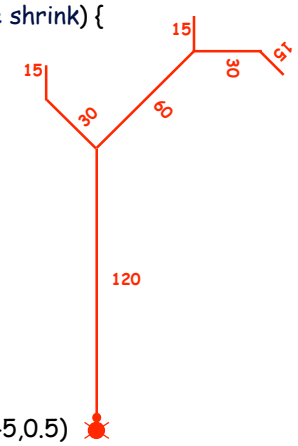

```
public int dropStepsToWall() {
   if (isFacingWall) {
      return dropInt(0);
   } else {
      forward();
      int steps = dropStepsToWall();
      backward();
      return dropInt(1+steps);
   }
}
```



# bush() : A Variation of tree()

bush() is a variation of tree() which
(1) does not have a levels parameter
(2) will not draw a branch shorter than 10 and
(3) shrinks the left branch by an extra shrink factor.

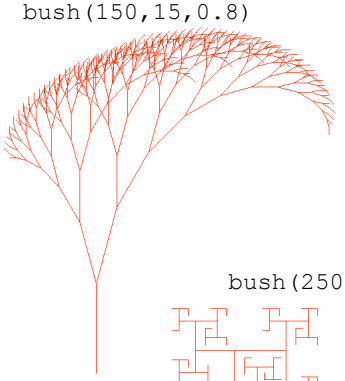
```
public void bush(double length, double angle, double shrink) {
   if (length < 10) {
      // do nothing
   } else {
      fd(length);
      rt(angle);
      bush(shrink*length, angle, shrink);
      lt(2*angle)
      bush(shrink*shrink*length, angle, shrink);
      rt(angle);
      bd(length);
   }
}
```



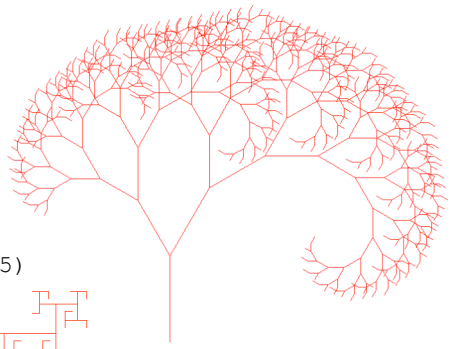

bush(120,45,0.5)

# Some Examples of bush()

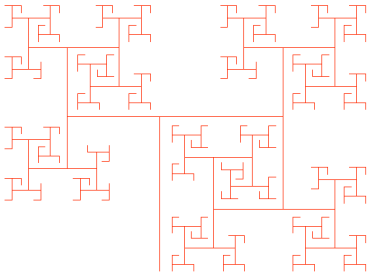
bush(150,15,0.8)



bush(150,30,0.82)



bush(250,90,0.75)

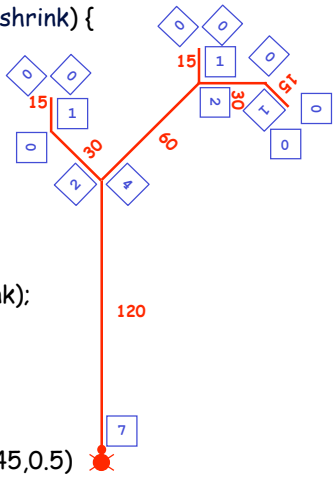




# Making bush() Fruitful

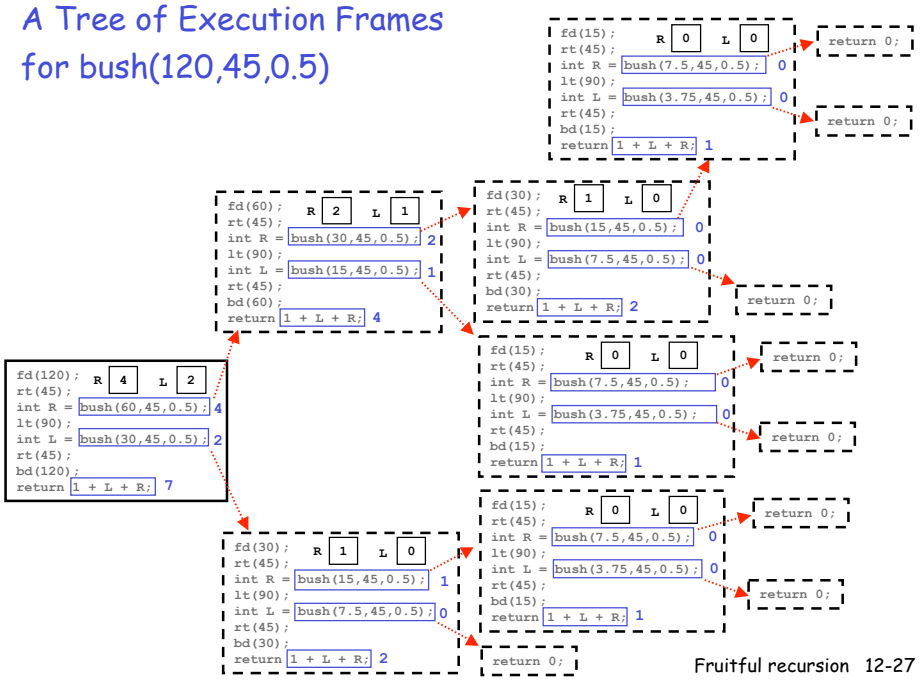
How could we modify the bush() method to return the number of branches drawn in addition to drawing the branches?

```
public int bush(double length, double angle, double shrink) {
  if (length < 10) {
    return      ;
  } else {
    fd(length);
    rt(angle);
    int R = bush(shrink*length, angle, shrink);
    lt(2*angle)
    int L = bush(shrink*shrink*length, angle, shrink);
    rt(angle);
    bd(length);
    return      ;
  }
}
```




bush(120,45,0.5)

## A Tree of Execution Frames for bush(120,45,0.5)



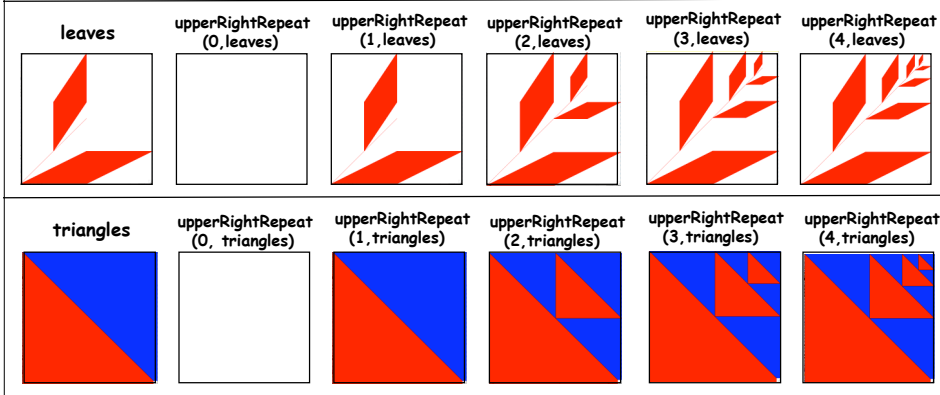



## upperRightRepeat()

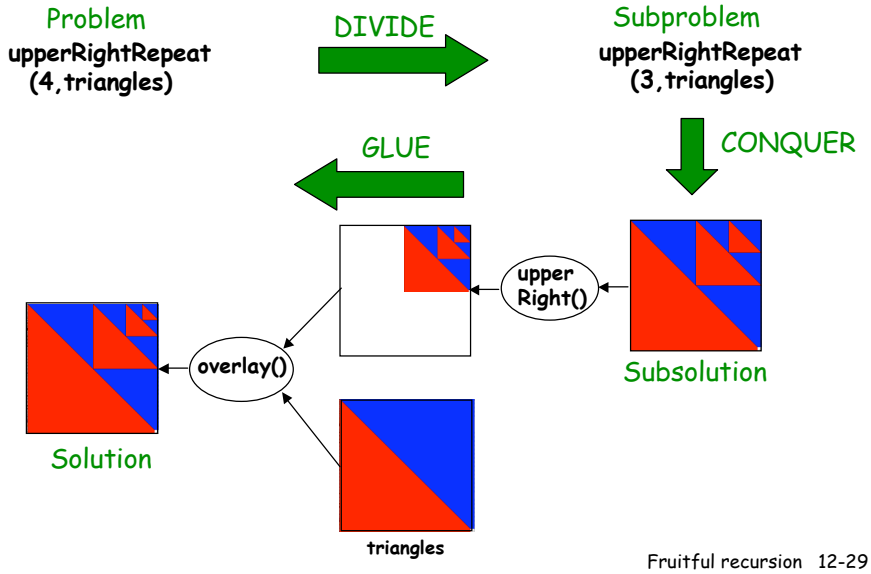
Suppose we are given:

```
public Picture upperRight(Picture p);
```


Our goal is to generate pictures composed out of recursive nestings of upperRight():



# Divide, Conquer, and Glue






# Method Declaration

Parameters and their types (if any)

```
public Picture upperRightRepeat(                    )
{
      if (                    ) {                   // base case
```

When does the recursion bottom out?

And what do we do when it does?

```
      } else {                                      // recursive case
```

What recursive call(s) are made and how are the result(s) glued together

```
      }
}
```

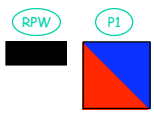
# Fleshing Out the Skeleton

```
public Picture upperRightRepeat(                                    )
{
      if (                      ) {                 // base case


      } else {                                      // recursive case









      }
}
```



# JEM for upperRightRepeat(3, triangles)

Object Land



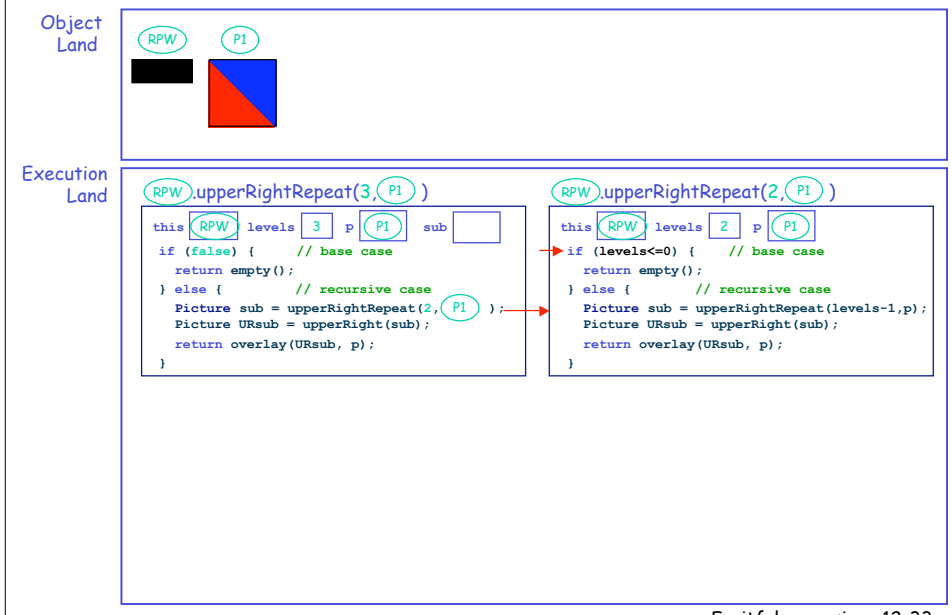

Execution Land

RPW.upperRightRepeat(3, P1 )

this RPW levels 3 p P1

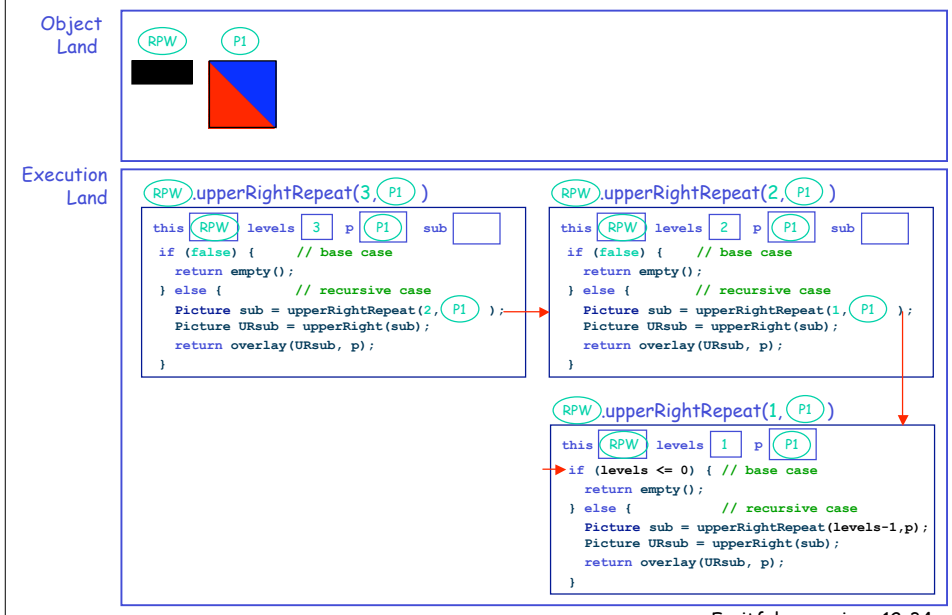
```
if (levels<=0) { // base case
   return empty();
 } else {          // recursive case
   Picture sub = upperRightRepeat(levels-1,p);
   Picture URsub = upperRight(sub);
   return overlay(URsub, p);
 }
```

# Enter upperRightRepeat at levels 2

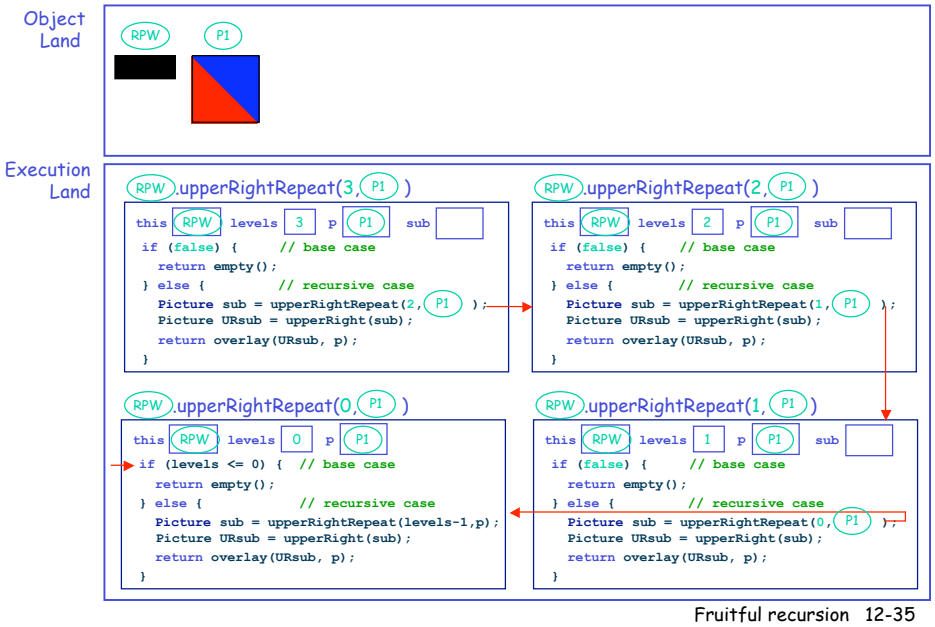





# Enter upperRightRepeat at levels 1

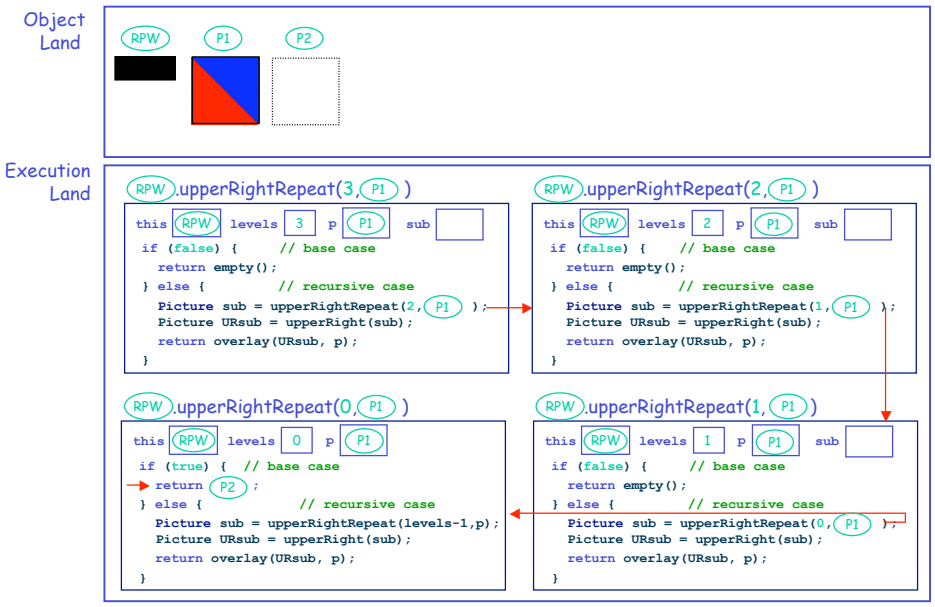


# Enter upperRightRepeat at levels 0

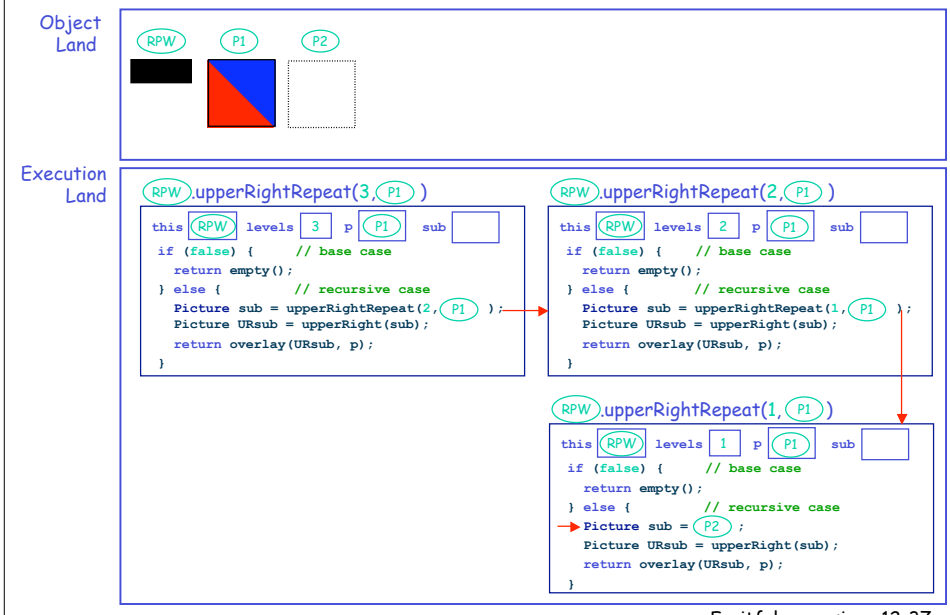





# Return from upperRightRepeat at levels 0

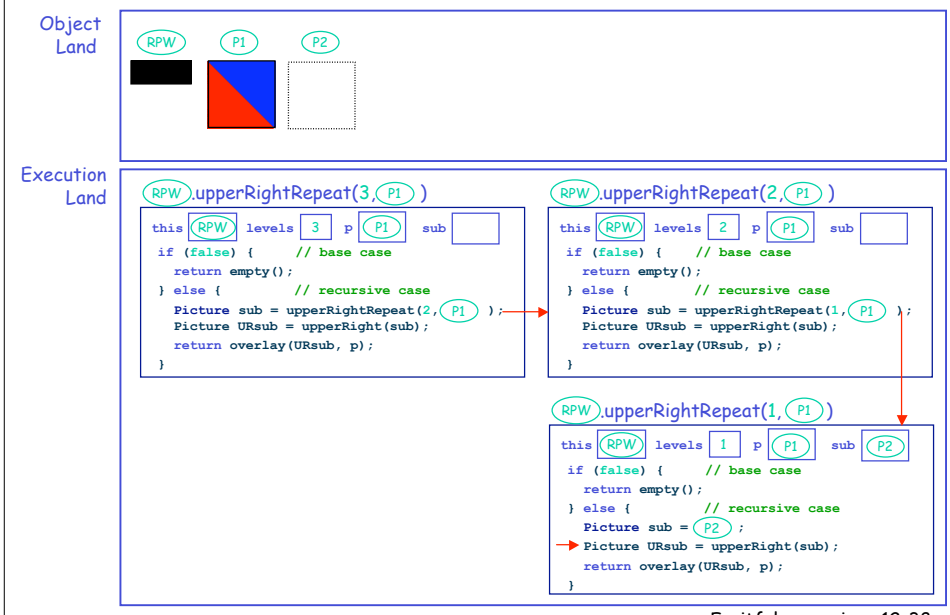


# Name the Subsolution

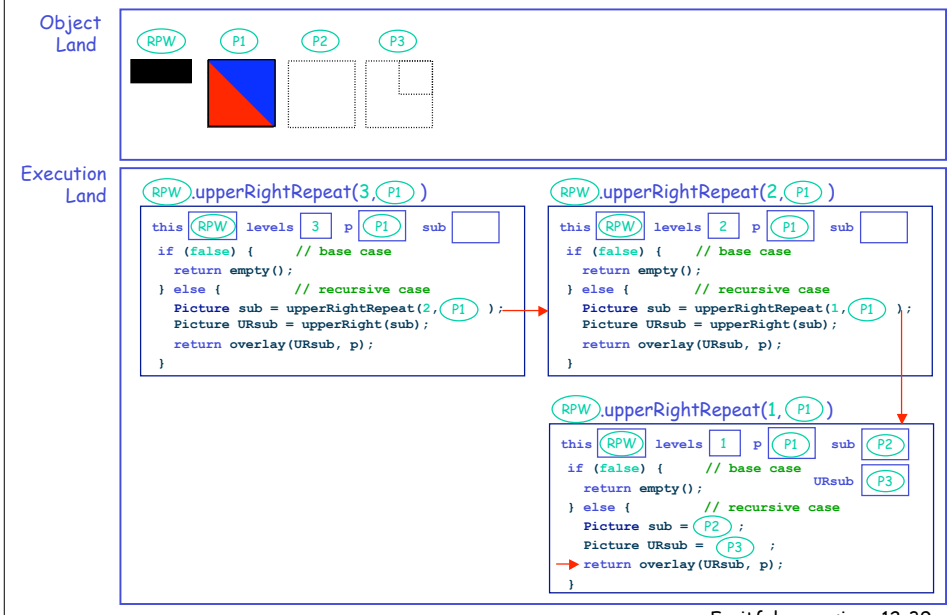





# Squeeze the Subsolution to the Upper Right

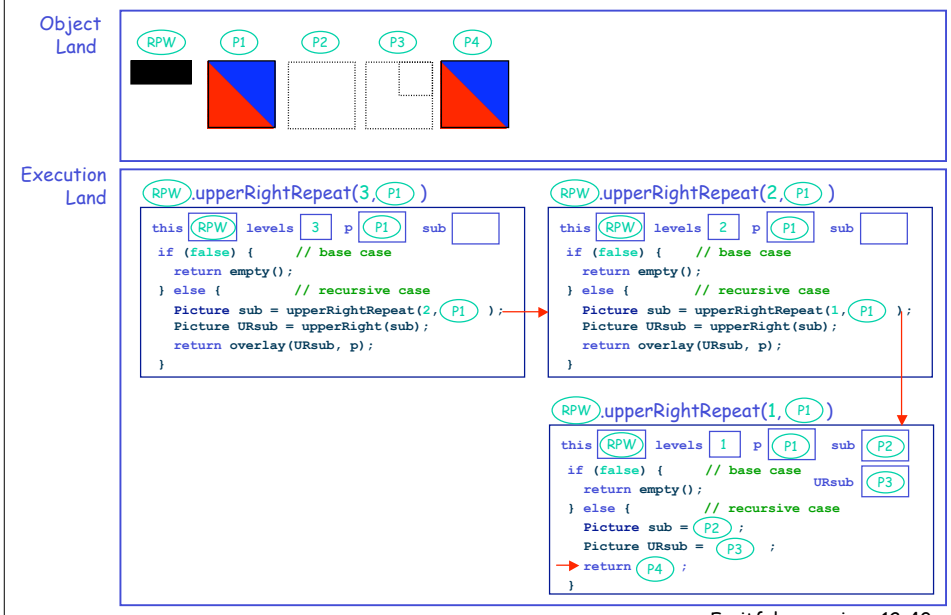


# Perform the overlay

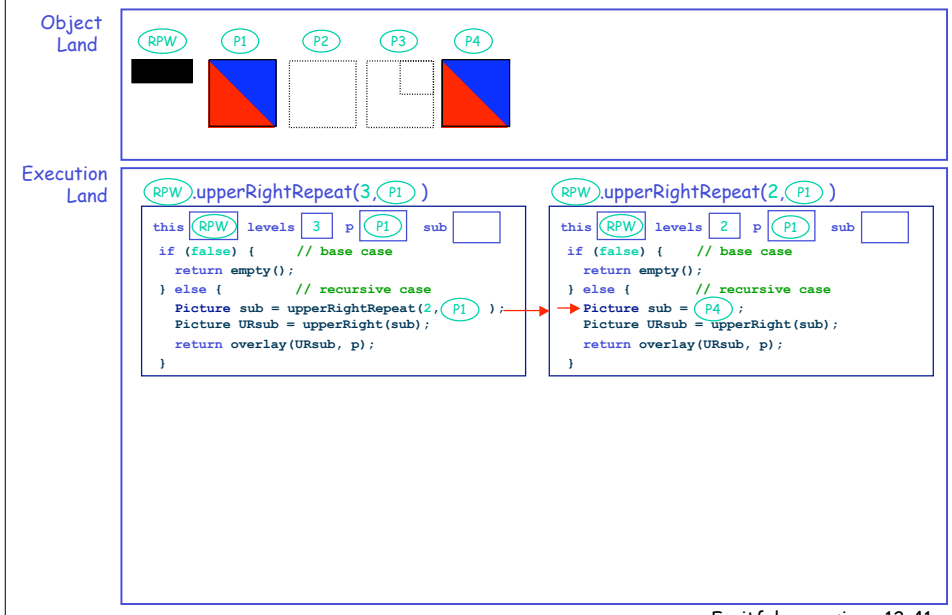





# Return from upperRightRepeat at levels 1

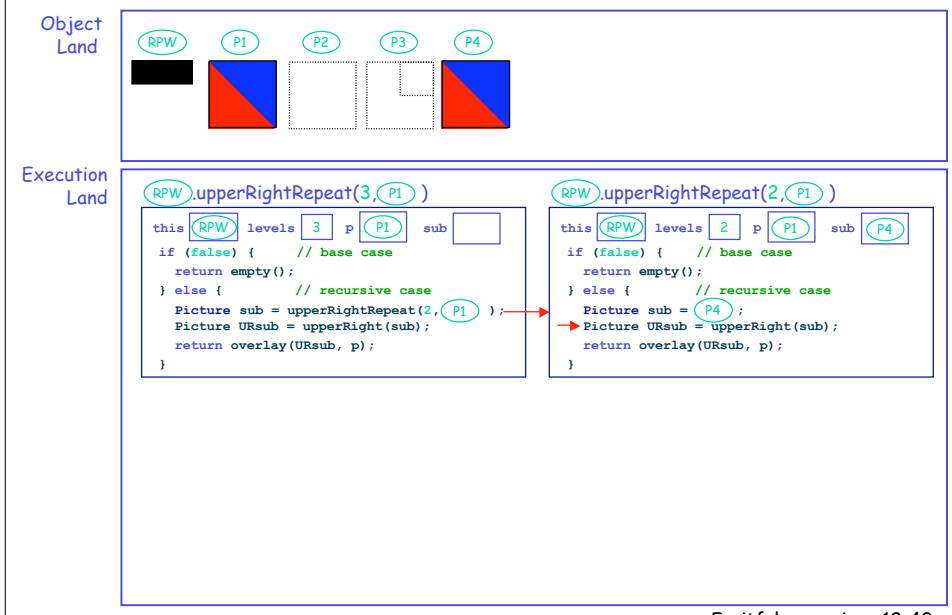


# Name the Subsolution






# Squeeze the Subsolution to the Upper Right



# Perform the Overlay

Object Land


Execution Land

RPW.upperRightRepeat(3, P1 )

```
this RPW  levels 3  p P1  sub
if (false) {      // base case
  return empty();
} else {          // recursive case
  Picture sub = upperRightRepeat(2, P1 );
  Picture URsub = upperRight(sub);
  return overlay(URsub, p);
}
```

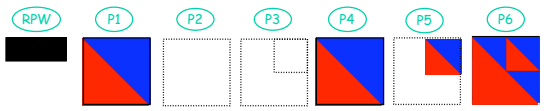
RPW.upperRightRepeat(2, P1 )

```
this RPW  levels 2  p P1  sub P4
if (false) {      // base case
  return empty();          URsub P5
} else {          // recursive case
  Picture sub = P4 ;
  Picture URsub = P5 ;
return overlay(URsub, p);
}
```



# Return from upperRightRepeat at levels 2

Object Land




Execution Land

RPW.upperRightRepeat(3, P1 )

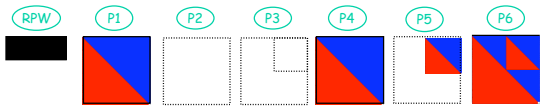
```
this RPW  levels 3  p P1  sub
if (false) {      // base case
  return empty();
} else {          // recursive case
  Picture sub = upperRightRepeat(2, P1 );
  Picture URsub = upperRight(sub);
  return overlay(URsub, p);
}
```

RPW.upperRightRepeat(2, P1 )

```
this RPW  levels 2  p P1  sub P4
if (false) {      // base case
  return empty();          URsub P5
} else {          // recursive case
  Picture sub = P4 ;
  Picture URsub = P5 ;
return P6 ;
}
```

# Name the Subsolution

Object Land




Execution Land

RPW.upperRightRepeat(3, P1 )

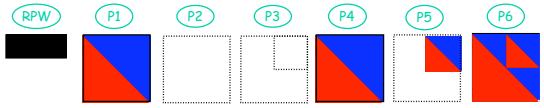
this RPW levels 3 p P1 sub

```
if (false) {      // base case
  return empty();
} else {          // recursive case
  Picture sub = P6 ;
  Picture URsub = upperRight(sub);
  return overlay(URsub, p);
}
```



# Squeeze the Subsolution to the Upper Right

Object Land




Execution Land

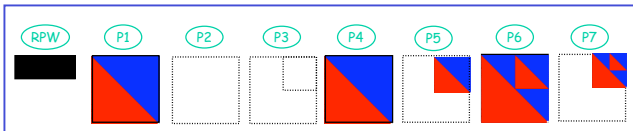
RPW.upperRightRepeat(3, P1 )

this RPW levels 3 p P1 sub P6

```
if (false) {      // base case
  return empty();
} else {          // recursive case
  Picture sub = P6 ;
  Picture URsub = upperRight(sub);
  return overlay(URsub, p);
}
```

# Perform the Overlay

Object Land




Execution Land

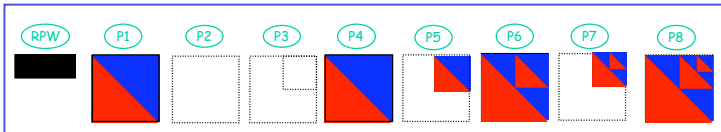
RPW.upperRightRepeat(3, P1 )

```
this RPW  levels 3  p P1  sub P6
if (false) {      // base case
  return empty();          URsub P7
} else {          // recursive case
  Picture sub = P6 ;
  Picture URsub = P7 ;
→ return overlay(URsub, p);
}
```



# Return from upperRightRepeat(3,triangles)

Object Land




Execution Land

RPW.upperRightRepeat(3, P1 )

```
this RPW  levels 3  p P1  sub P6
if (false) {      // base case
  return empty();          URsub P7
} else {          // recursive case
  Picture sub = P6 ;
  Picture URsub = P7 ;
→ return P8 ;
}
```

# Yippie