

Solutions to the Review Problems for CS111 EXAM 1

The first exam is coming up. The exam is open notes: you may refer to any handouts, your notes, and your assignments, but you may not refer to anyone else's materials nor any materials from previous semesters of CS111. . You may not use a computer during the exam.

This handout includes some problems adapted from previous exams that you may find helpful in studying for the exam. These problems are not necessarily indicative of the kinds of problems you may be given on your exam or the length of your exam, but they do cover much of the material you are expected to know for the exam.

Here are the review problems with solutions. It is not a good idea to study by looking at the solutions first. Try to solve the problems on your own, then use the solutions to check your work and your understanding.

Problem 1: Buggle World Execution

Consider the two Java classes in Fig. 1.

```
public class DoItWorld extends BuggleWorld
{
    public void run ()
    {
        DoItBuggle dewey = new DoItBuggle();           // run statement 1
        int n = 5;                                       // run statement 2
        dewey.setPosition(new Location(n, n - 2));      // run statement 3 *
        dewey.brushUp();                                 // run statement 4
        dewey.doit(Color.green, n - 1);                // run statement 5 *
        dewey.doit(Color.blue, n + 1);                 // run statement 6 *
        dewey.forward();                                // run statement 7
        dewey.brushDown();                              // run statement 8
        dewey.forward(3);                               // run statement 9 *
    }
}

class DoItBuggle extends Buggle
{
    public void doit (Color c, int n)
    {
        Color oldColor = this.getColor();
        this.setColor(c);
        this.forward(n);
        this.brushDown();
        this.backward(n-2);
        this.brushUp();
        this.backward(2);
        this.left();
        this.setColor(oldColor);
    }
}
```

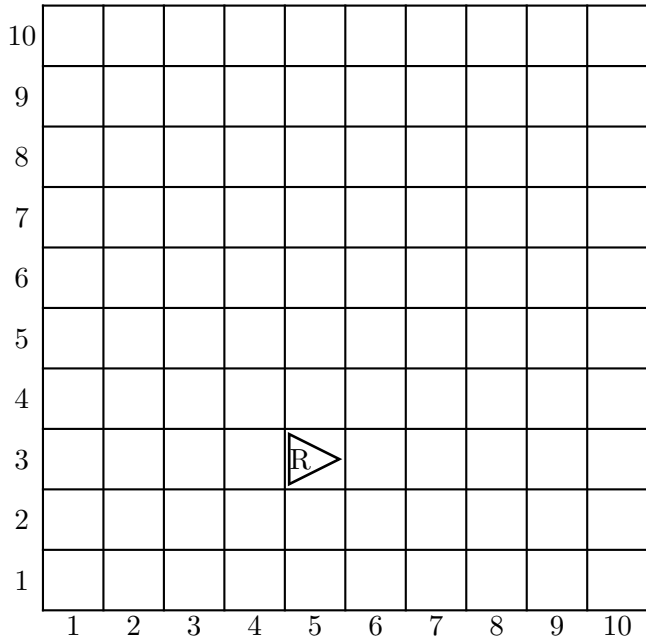
Figure 1: Two Java classes.

Suppose that the `run()` method is invoked on an instance of `DoItWorld` which has a 10×10 grid of cells. In the four grids on the following page, show the state of the grid directly *after* the execution of each of the statements in the `run()` method body marked with a `*`.

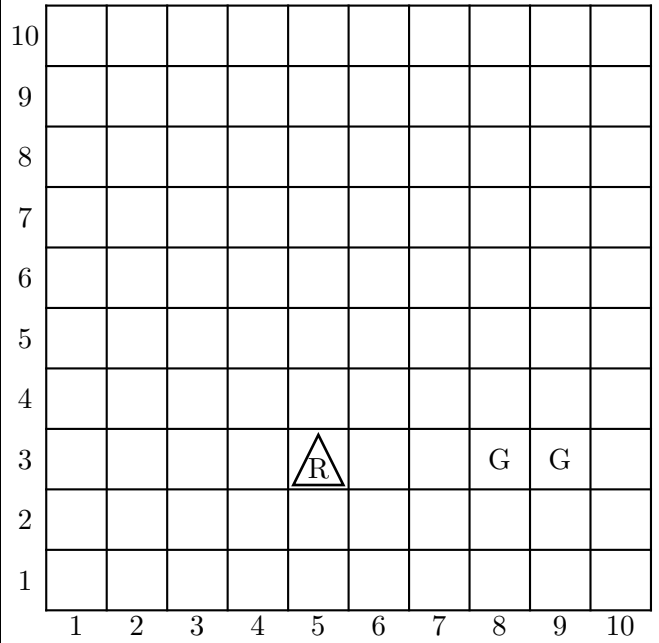
In each grid, you should show the following:

1. Draw buggle `dewey` as a triangle pointing in the direction that the buggle is facing.
2. Indicate the current color of the buggle by putting the *first letter* of the color name inside the triangle (e.g. B for blue, G for green, etc.).
3. Indicate the color of each non-white grid cell by putting the *first letter* of the color name inside the cell (e.g. B for blue, G for green, etc.).

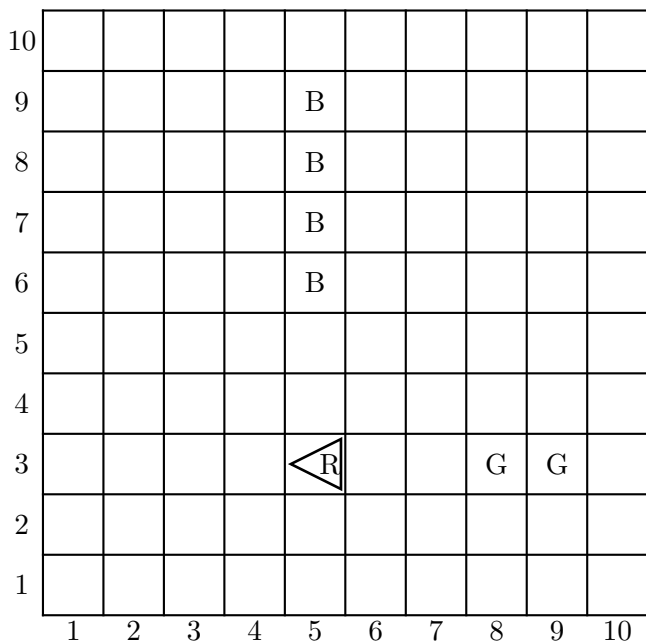
DoItWorld grid after the execution of run() statement 3



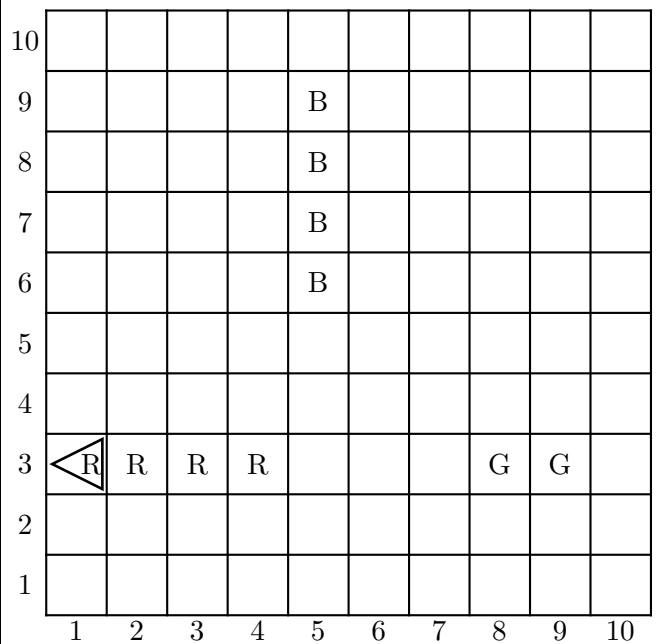
DoItWorld grid after the execution of run() statement 5



DoItWorld grid after the execution of run() statement 6



DoItWorld grid after the execution of run() statement 9



Problem 2: Debugging

The class declarations in Fig. 2 contain (at least) **10 errors** (syntax errors and type errors).

```
public class ExamBuggleWorld extends BuggleWorld // line 1
{ // line 2
    public void run () // line 3
    { // line 4
        Color c = Color.cyan(); // line 5
        int n = 4 // line 6
        ExamBuggle emma = ExamBuggle(); // line 7
        emma.mystery1(c, n); // line 8
        emma.mystery1(3, Color.red); // line 9
        boolean answer = emma.mystery2(); // line 10
        this.mystery3(); // line 11
    } // line 12
} // line 13
// line 14
class ExamBuggle extends Buggle // line 15
{ // line 16
    public void mystery1(Color c, int n1) // line 17
    { // line 18
        n2 = n1 + 1; // line 19
        this.setColor(Color.c); // line 20
        forward(n2); // line 21
        this.dropBagel(); // line 22
        // line 23
    } // line 24
    public boolean mystery2() // line 25
    { // line 26
        this.isOverBagel(); // line 27
    } // line 28
    public mystery3() // line 29
    { // line 30
        this.dropBagel(); // line 31
    } // line 32
} // line 33
```

Figure 2:

In the table on the next page, for each of 10 errors in different lines of the above program give:

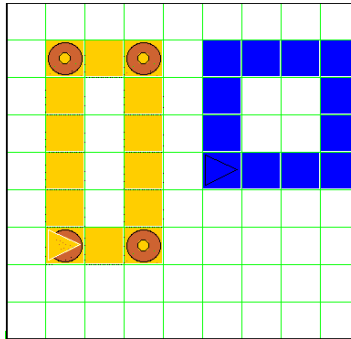
1. the line number of the error,
2. a *brief* description of the error, and
3. a corrected version of the line (i.e., with the error fixed).

You may list the errors in *any* order. You do *not* have to list them in the order in which they occur in the program.

Error #	Line #	Brief description of error	Corrected line
1	5	Color.cyan() is not a method invocation	Color c = Color.cyan;
2	6	The local variable declaration int n = 4 is missing a semi-colon at the end	int n = 4;
3	7	There is a missing new in the constructor method invocation that creates an ExamBuggle	ExamBuggle emma = new ExamBuggle();
4	9	The two arguments of the instance method invocation emma.mystery1(3, Color.red) are in the wrong order	emma.mystery1(Color.red, 3);
5	11	In this.mystery3(), this stands for an instance of ExamBuggleWorld, which does not understand the mystery3 message; the recipient should be an instance of ExamBuggle	emma.mystery3();
6	19	The local variable declaration n2 = n1 + 1; is missing a type for the contents of the variable	int n2 = n1 + 1;
7	20	Color.c attempts to reference a non-existent class constant rather than the parameter c	this.setColor(c);
8	23	The instance method declaration for mystery1() is missing a close curly brace.	}
9	26	The non-void method mystery2() is missing a return statement.	return this.isOverBagel();
10	30	The method header for mystery3() is missing the return type, void	public void mystery3() {

Problem 3: Buggle Methods

A class of Buggles enjoys doing window treatments. They call themselves Windowers. In WindowWorld, wendy and winifred each do a window treatment:



```
public class WindowWorld extends BuggleWorld {

    public void run()
    {
        Windower wendy = new Windower();           // line 1
        Windower winifred = new Windower();        // line 2

        wendy.setPosition(new Location(2, 3));     // line 3
        wendy.setColor(Color.orange);             // line 4
        wendy.forward(2);                          // line 5
        wendy.dropBagel();                         // line 6
        wendy.left();                              // line 7
        wendy.forward(5);                          // line 8
        wendy.dropBagel();                         // line 9
        wendy.left();                              // line 10
        wendy.forward(2);                          // line 11
        wendy.dropBagel();                         // line 12
        wendy.left();                              // line 13
        wendy.forward(5);                          // line 14
        wendy.dropBagel();                         // line 15
        wendy.left();                              // line 16

        winifred.setPosition(new Location(6, 5));  // line 17
        winifred.setColor(Color.blue);            // line 18
        winifred.forward(3);                      // line 19
        winifred.left();                          // line 20
        winifred.forward(3);                      // line 21
        winifred.left();                          // line 22
        winifred.forward(3);                      // line 23
        winifred.left();                          // line 24
        winifred.forward(3);                      // line 25
        winifred.left();                          // line 26
    }
}
```

a Assume there is a `Windower` class, which extends `Buggle`. Capture the repeated pattern of code in the `run()` method above by creating a single method named `decorateWindow()` that produces the same window treatments that `wendy` and `winifred` created above in lines 3–16 and

17–26. You may assume that your `decorateWindow()` method is being defined in the `Windower` class. Your method should take 5 parameters that provide the following information:

- a location specifying the position of the window’s lower left corner,
- color of the window,
- width of the window (number of cells),
- height of the window (number of cells),
- and a boolean value that says whether the window corners should be decorated with bagels.

Assume an infinite grid, i.e., you don’t have to worry about whether your windows will fit in the `BuggleWorld` grid.

answer:

```
public void decorateWindow(Location startPos, Color c, int width,
                           int height, boolean bagelInCorners)
{
    setPosition(startPos);
    setColor(c);
    forward(width - 1); // Need - 1 to convert between cells and steps forward
    if (bagelInCorners)
        dropBagel();
    left();
    forward(height - 1);
    if (bagelInCorners)
        dropBagel();
    left();
    forward(width - 1);
    if (bagelInCorners)
        dropBagel();
    left();
    forward(height - 1);
    if (bagelInCorners)
        dropBagel();
    left();
}
```

We can make this solution more compact by factoring out repeated parts of this code using the following `windowSide()` helper method:

```
private void windowSide(int length, boolean bagelInCorners)
{
    forward(length - 1); // Need - 1 to convert between cells and steps forward
    if (bagelInCorners)
        dropBagel();
    left();
}

public void decorateWindow(Location startPos, Color c, int width,
                           int height, boolean bagelInCorners)
{
    setPosition(startPos);
    setColor(c);
    windowSide(width, bagelInCorners);
    windowSide(height, bagelInCorners);
    windowSide(width, bagelInCorners);
    windowSide(height, bagelInCorners);
}
```

b Below, write the two invocations of your `decorateWindow()` method that will replace lines 3–16 and lines 17–26 in the `run()` method:

- *invocation to replace lines 3–16:*

answer:

```
wendy.decorateWindow(new Location(2, 3), Color.orange, 3, 6, true);
```

- *invocation to replace lines 17–26:*

answer:

```
winifred.decorateWindow(new Location(6, 5), Color.blue, 4, 4, false);
```


Problem 5: Booleans and Conditionals

a Bud Lojack has written the following method in a rather unclear programming style:

```
public boolean isColdAndHeadingNorth ()
{
    if (getColor().equals(Color.blue)) {
        if (getHeading().equals(Direction.NORTH)) {
            return true;
        } else {
            return false;
        }
    } else if (!getColor().equals(Color.blue)) {
        return false;
    } else if (!getHeading().equals(Direction.NORTH)) {
        return false;
    } else {
        return true;
    }
}
```

Rewrite Bud's method in a much clearer style.

answer:

The second test expression, `!getColor().equals(Color.blue)`, is only evaluated if the first test expression, `getColor().equals(Color.blue)`, evaluates to `false`. But this means that `!getColor().equals(Color.blue)` is equivalent to `!false` or `true`:

```
public boolean isColdAndHeadingNorth ()
{
    if (getColor().equals(Color.blue)) {
        if (getHeading().equals(Direction.NORTH)) {
            return true;
        } else {
            return false;
        }
    } else if (true) {
        return false;
    } else if (!getHeading().equals(Direction.NORTH)) {
        return false;
    } else {
        return true;
    }
}
```

The statement `if (true) S1 else S2;` can always be replaced by `S1`, and we don't actually have to write any of the curly braces here (do you know why?):

```
public boolean isColdAndHeadingNorth ()
{
    if (getColor().equals(Color.blue))
        if (getHeading().equals(Direction.NORTH))
            return true;
        else
            return false;
    else
        return false;
}
```

The pattern `if (E) return true; else return false;` can always be replaced by `return E;`, so we can simplify further to yield:

```
public boolean isColdAndHeadingNorth ()
{
    if (getColor().equals(Color.blue))
        return getHeading().equals(Direction.NORTH);
    else
        return false;
}
```

Finally, the pattern `if (E1) return E2; else return false;` is equivalent to `return E1 && E2;`, so the original method can be simplified to:

```
public boolean isColdAndHeadingNorth ()
{
    return getColor().equals(Color.blue) && getHeading().equals(Direction.NORTH);
}
```

b Define a `Buggle` method named `isBoxedIn()` that has no parameters and returns `true` if a buggle is in a cell surrounded by walls on all four sides, and otherwise returns `false`. The final state of the buggle when `isBoxedIn()` returns should be the same as the state of the buggle when `isBoxedIn()` is invoked. You may not use recursion or iteration in your solution, but you may define auxiliary methods if you wish.

answer: There are many different ways to define `isBoxedIn()`. Here we look at a few approaches.

One approach that is easy to read but is not particularly efficient is to use a separate predicate for each of the four positions:

```
public boolean isBoxedIn()
{
    return isFacingWall() && isWallToLeft() && isWallInBack() && isWallToRight();
}
```

```
public boolean isWallToLeft()
{
    left();
    boolean result = isFacingWall();
    right();

    return result;
}
```

```
public boolean isWallInBack()
{
    left();
    boolean result = isWallToLeft();
    right();

    return result;
}
```

```

public boolean isWallToRight()
{
    left();
    boolean result = isWallInBack();
    right();

    return result;
}

```

Note that `isWallToRight()` actually turns leftward three times, and could be made more efficient by turning right once instead:

```

public boolean isWallToRight()
{
    right();
    boolean result = isFacingWall();
    left();

    return result;
}

```

Even so, the buggle repeats a lot of turning in the helper predicates. The repeated turning can be eliminated by performing all tests within `isBoxedIn()` itself. Although the result is more efficient, it is rather difficult to read and write:

```

public boolean isBoxedIn()
{
    if (!isFacingWall()) { // No wall in front
        return false;
    } else {
        left(); // Check left wall
        if (!isFacingWall()) { // No wall to left
            right(); // Return to initial heading before return
            return false;
        } else {
            left(); // Check back wall
            if (!isFacingWall()) { // No wall in back
                right();
                right(); // Return to initial heading before return
                return false;
            } else {
                left(); // Check right wall
                if (!isFacingWall()) { // No wall to right
                    left(); // Return to initial heading before return
                    // (three rights is a left)
                    return false;
                } else { // Surrounded by four walls
                    left(); // Return to initial heading before return.
                    return true;
                }
            }
        }
    }
}

```

A more compact way to check all four sides is to maintain the result in a `boolean` variable (here named `result`) that is updated at every wall. This is simple to read and write, but is not as efficient as the above approach because it continues to visit all headings even after finding a

missing wall.

```
public boolean isBoxedIn()
{
    boolean result = isFacingWall(); // Check front wall
    left();
    result = result && isFacingWall(); // Check left wall
    left();
    result = result && isFacingWall(); // Check back wall
    left();
    result = result && isFacingWall(); // Check right wall
    left(); // Return to facing front wall
    return result;
}
```

Problem 6: Java Execution Model in BuggleWorld

Consider the following two class definitions:

```
public class RelayRaceWorld extends BuggleWorld
{
    public void run()
    {
        RelayRunner r1 = new RelayRunner();
        RelayRunner r2 = new RelayRunner();
        RelayRunner r3 = new RelayRunner();

        r2.setColor(Color.green);
        r3.setColor(Color.blue);
        r1.firstLeg(3, r2, r3);
    }
}

class RelayRunner extends Buggle
{
    public void firstLeg(int length, RelayRunner next, RelayRunner last)
    {
        this.forward(length);
        next.setPosition(this.getPosition());
        next.secondLeg(length, last);
    }

    public void secondLeg(int length, RelayRunner next)
    {
        this.forward(length);
        next.setPosition(this.getPosition());
        next.thirdLeg(length);
    }

    public void thirdLeg(int length)
    {
        this.forward(length);
    }
}
```

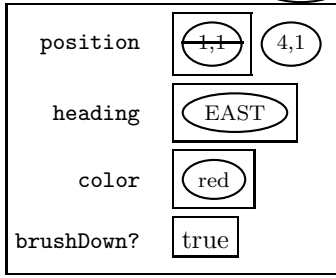
The final JEM is shown on the next page. This figure shows how each expression is evaluated to produce a value. You were asked only to show the final state of the JEM, so you were not required to show all this, however, it is very useful to do your JEMs this way so you can keep track of the computation. This problem tested your knowledge of parameters, variables, and method invocation in a very detailed way.

RelayRaceWorld (RRW)

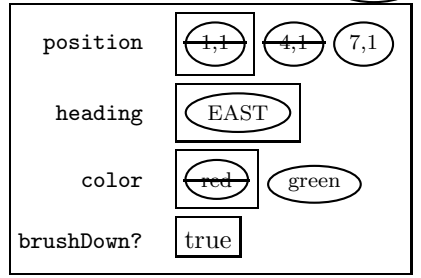


Object Land

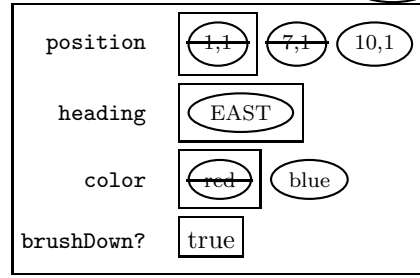
RelayRunner (RR1)



RelayRunner (RR2)



RelayRunner (RR2)



Execution Land

RRW.run()

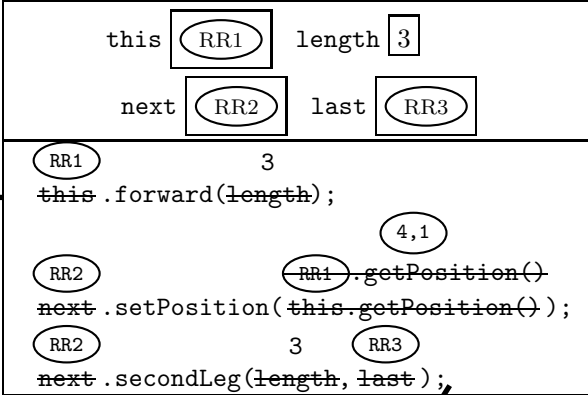
```

this (RRW) r1 (RR1)
r2 (RR2) r3 (RR3)

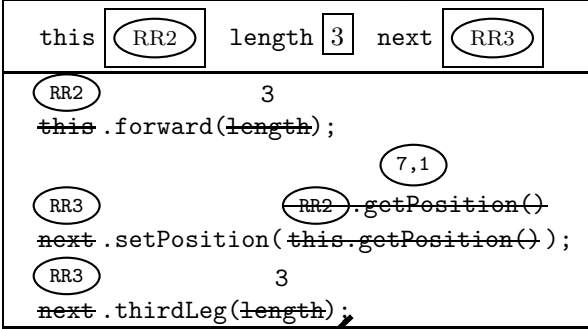
RelayRunner r1 = new RelayRunner();
RelayRunner r2 = new RelayRunner();
RelayRunner r3 = new RelayRunner();
r2.setColor(Color.green);
r3.setColor(Color.blue);
r1.firstLeg(3, r2, r3);

```

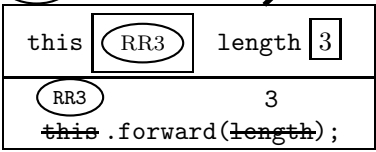
RR1.firstLeg(3, RR2, RR3)



RR2.secondLeg(3, RR3)



RR3.thirdLeg(3)



Problem 7: Java Execution Model in PictureWorld

```

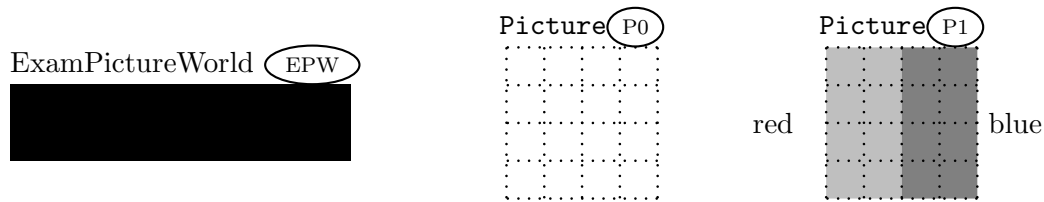
public class ExamPictureWorld extends PictureWorld
{
    public Picture meth1 (Picture a)
    {
        Picture b = beside(a, empty());
        Picture c = meth2(b);
        return overlay(c, b);
    }

    public Picture meth2 (Picture a)
    {
        Picture b = above(a, empty(), 0.75);
        return clockwise90(b);
    }
}

```

Figure 3: A subclass of PictureWorld.

Consider the subclass of PictureWorld shown in Fig. 3. Suppose that: $\textcircled{\text{EPW}}$ is an instance of ExamPictureWorld, $\textcircled{\text{P0}}$ is a Picture instance denoting the empty picture, $\textcircled{\text{P1}}$ is a Picture instance denoting the rightmost picture below:



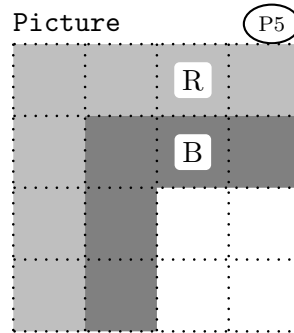
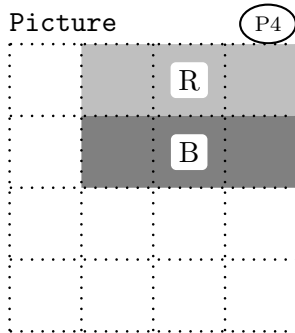
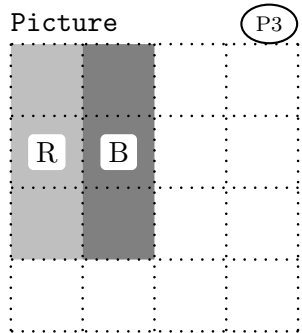
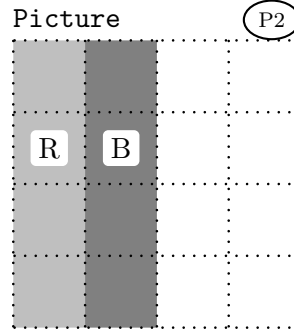
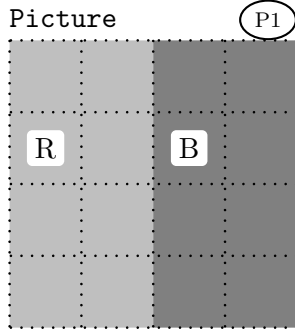
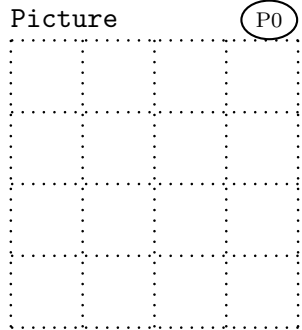
The dashed grid lines are *not* part of the pictures. They indicate coordinates within pictures. The colors names are *not* part of picture $\textcircled{\text{P1}}$. They indicate the color of the two rectangles. Each of the two rectangles is a solid color *without* any separately colored border.

On the next page, you should flesh out the Java Execution Model for the method invocation $\textcircled{\text{EPW}}.\text{meth1}(\textcircled{\text{P1}})$. In the area labeled **Execution Land**, you should flesh out the contents of the execution frame for this method invocation, as well as show the execution frame for the invocation of `meth2()`.

In the area labeled **Object Land** are the skeletons for the six Picture instances that are used during the execution. The pictures labeled $\textcircled{\text{P0}}$ and $\textcircled{\text{P1}}$ have already been drawn for you; you should draw pictures for the four new Picture instances $\textcircled{\text{P2}}$, $\textcircled{\text{P3}}$, $\textcircled{\text{P4}}$, and $\textcircled{\text{P5}}$ that will be created during the execution of $\textcircled{\text{EPW}}.\text{meth1}(\textcircled{\text{P1}})$. In each picture, you should label red areas with the letter R and blue areas with the letter B. All other areas are presumed to be “clear”.

Object Land

ExamPictureWorld (EPW)



Execution Land

EPW.meth1(P1)

```

this EPW a P1 b P2 c P4
Picture b = beside(a, empty());
Picture c = meth2(b);
return overlay(c, b);
    
```

EPW.meth2(P2)

```

this EPW a P2 b P3
Picture b = above(a, empty(), 0.75);
return clockwise90(b);
    
```

