MORE CS111 PRACTICE PROBLEMS FOR EXAM 1

Problem 1: Booleans and Conditionals

a. (Circle True or False) `true` and `false` denote objects in Java. Briefly justify your answer.

answer: False. Boolean values (as well as integers, floating point numbers, and characters) are primitive values, not objects. Primitive values cannot respond to messages.

b. Betty Bother has written the following method (in a rather unfortunate programming style) which does not compile properly:

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

(1) Explain the compiler error and (2) rewrite Betty's method in a much simpler style.

answer: (1) In the body of a fruitful method like `isColdAndHeadingNorth`, the Java compiler requires that every control path through the program must return a value. The above method has an implicit `else {}` branch that does not return a value, so the compiler complains — even though control can never actually take this path at run time (because the other two branches handle all possible cases).

(2) A much clearer and more succinct way to write the method is:

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

c. Which of the following program fragments are equivalent to:

```
if (A && B) {
    return bluePatch;
} else {
    return redPatch;
}
```

1.
```
if (A && B) {
    return bluePatch;
}
return redPatch;
```

2.
```
if (!A) {
    return redPatch;
}
if (B) {
    return bluePatch;
}
return redPatch;
```

3.
```
if (!A || !B) {
    return  redPatch;
}
return bluePatch;
```

answer: All three conditional statements are equivalent to the original one.

d. 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 want

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.

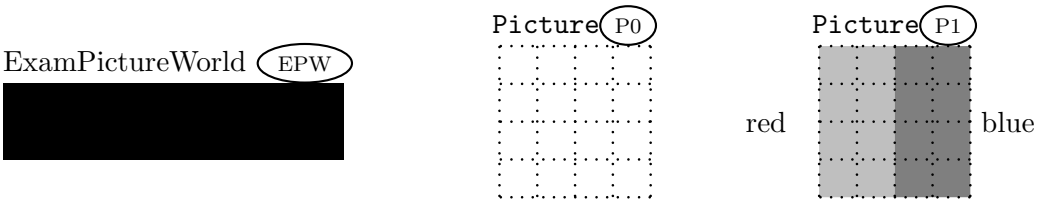
```
public boolean isBoxedIn() {
  boolean result = isFacingWall(); // Check front wall
  left();
  result = result && isFacingWall(); // Check left wall
  left();
  result = result && isFacingWall(); // Check back wall
  left();
  result = result && isFacingWall(); // Check right wall
  left();    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

Problem 2: Invocation Trees

```
public class ExamPictureWorld extends PictureWorld {

    public Picture meth1 (Picture a) {
        Picture b = beside(a, empty());
        return overlay(meth2(b), b);
    }

    public Picture meth2 (Picture c) {
        return clockwise90(above(c, empty(), 0.75));
    }
}
```

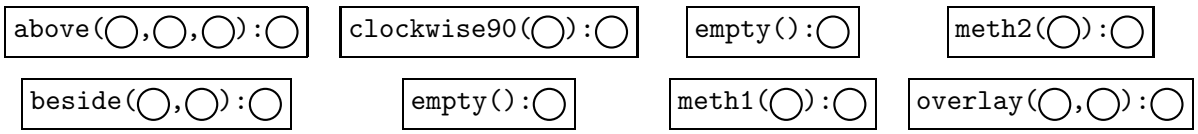
Figure 1: A sublcass of `PictureWorld`.

Consider the subclass of `PictureWorld` shown in Fig. 1. Supose that: (EPW) is an instance of `ExamPictureWorld`, (P0) is a `Picture` instance denoting the empty picture, (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 (P1). They indicate the color of the two rectangles. Each of the two rectangles is a solid color *without* any separately colored border.

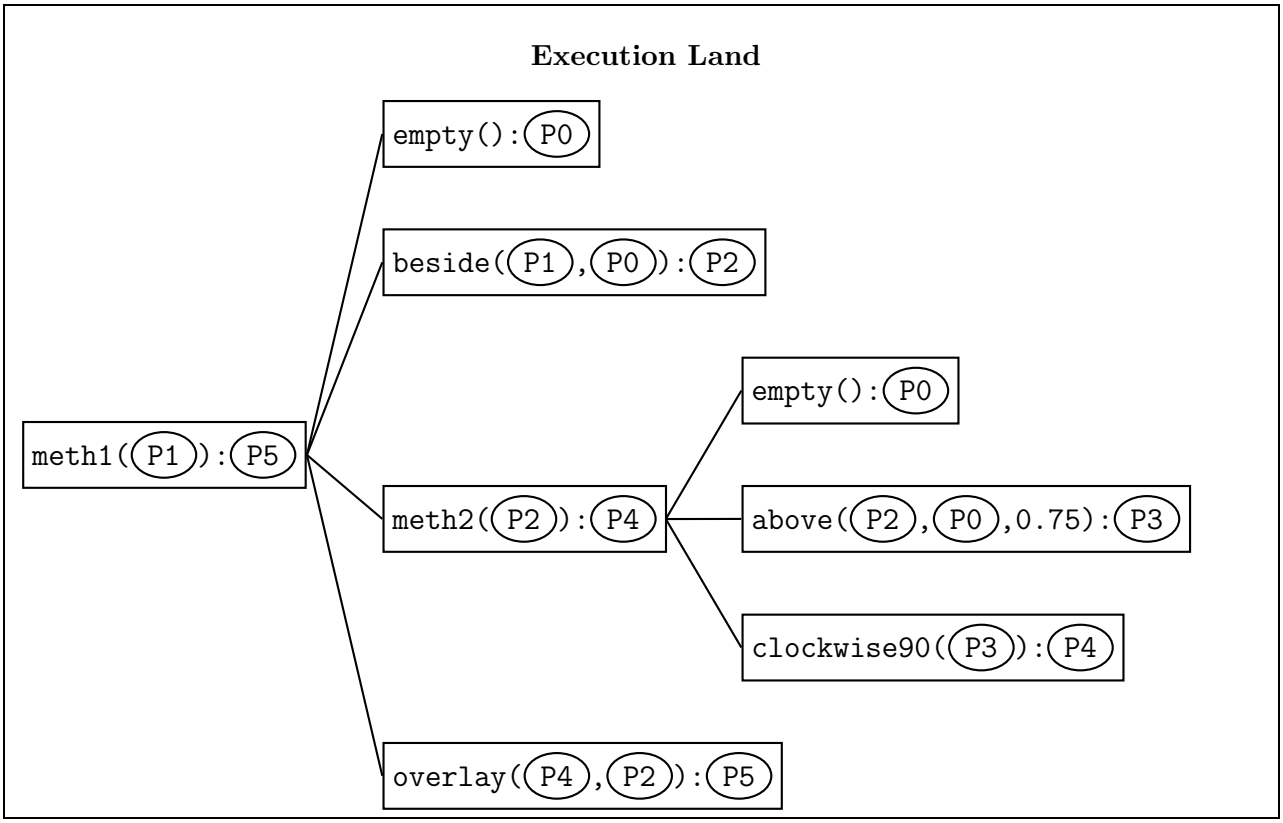
On the next page, you are to draw an invocation tree that models the instance method invocation (EPW).`meth1`((P1)). In the area labeled **Execution Land**, you should draw an invocation tree that contains the following eight nodes, arranged appropriately into a tree. You should use each node exactly once.

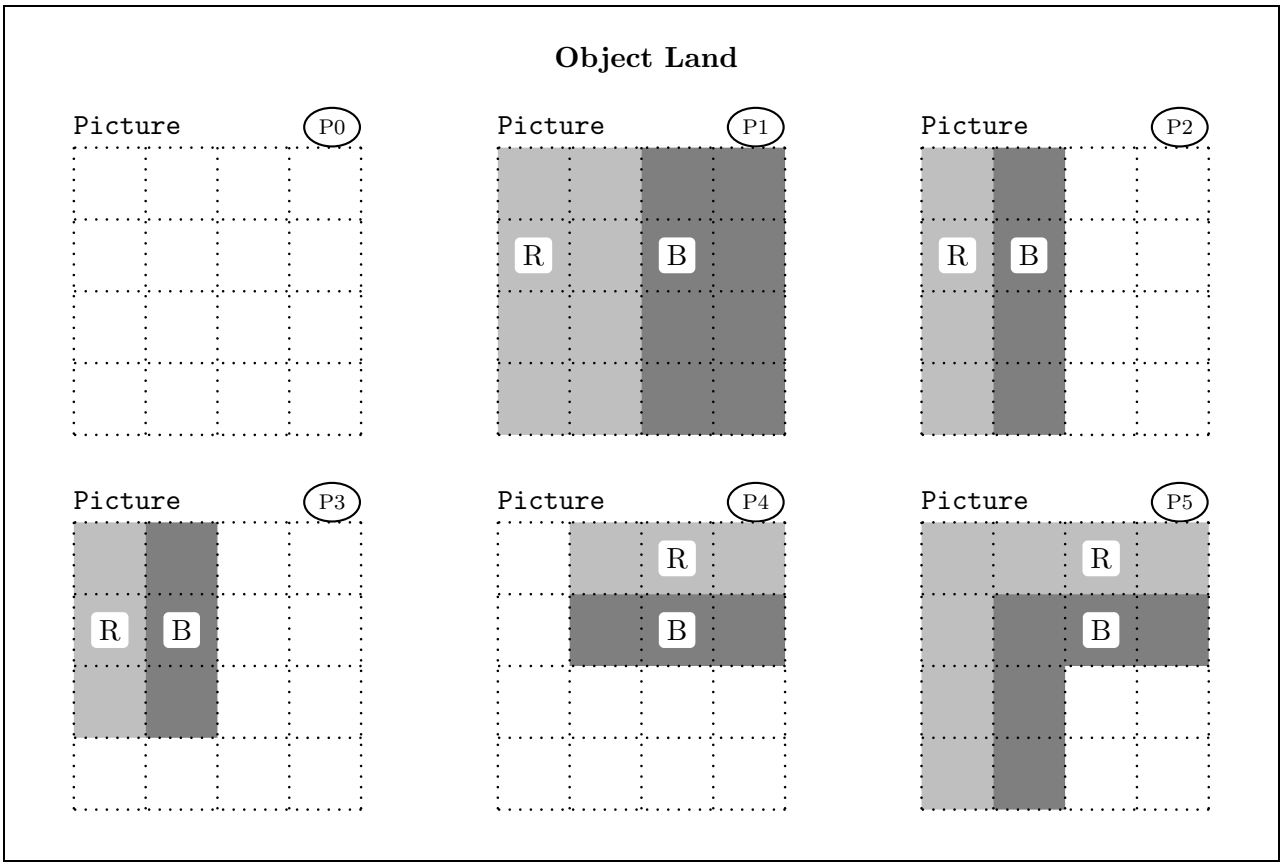


The empty circles in the nodes are skeletons for object references that you should fill in with one of the labels P0, P1, P2, P3, P4, or P5 to refer to the appropriate Picture instance in Object Land (see below). A circle enclosed by parentheses is a reference to an actual argument of the method invocation. A circle appearing after a colon is a reference to the result of the method invocation. The root of the invocation tree is the `meth1()` node, which has already been drawn for you, and whose actual argument has been filled in (you need to fill in its result).

In the area labeled **Object Land** are the skeletons for the six `Picture` instances that are used during the execution. The pictures labeled (P0) and (P1) have already been drawn for you; you should draw the pictures for (P2), (P3), (P4), and (P5). 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 white.

(Note: for simplicity, the receiver object (EPW) for each of the method invocations has been omitted. This instance has also been omitted from Object Land.)


Execution Land
empty():P0
beside(P1,P0):P2
meth1(P1):P5
empty():P0
meth2(P2):P4
above(P2,P0,0.75):P3
clockwise90(P3):P4
overlay(P4,P2):P5


Object Land
Picture P0
Picture P1
R
B
Picture P2
R
B
Picture P3
R
B
Picture P4
R
B
Picture P5
R
B

Problem 3: Java Execution Model

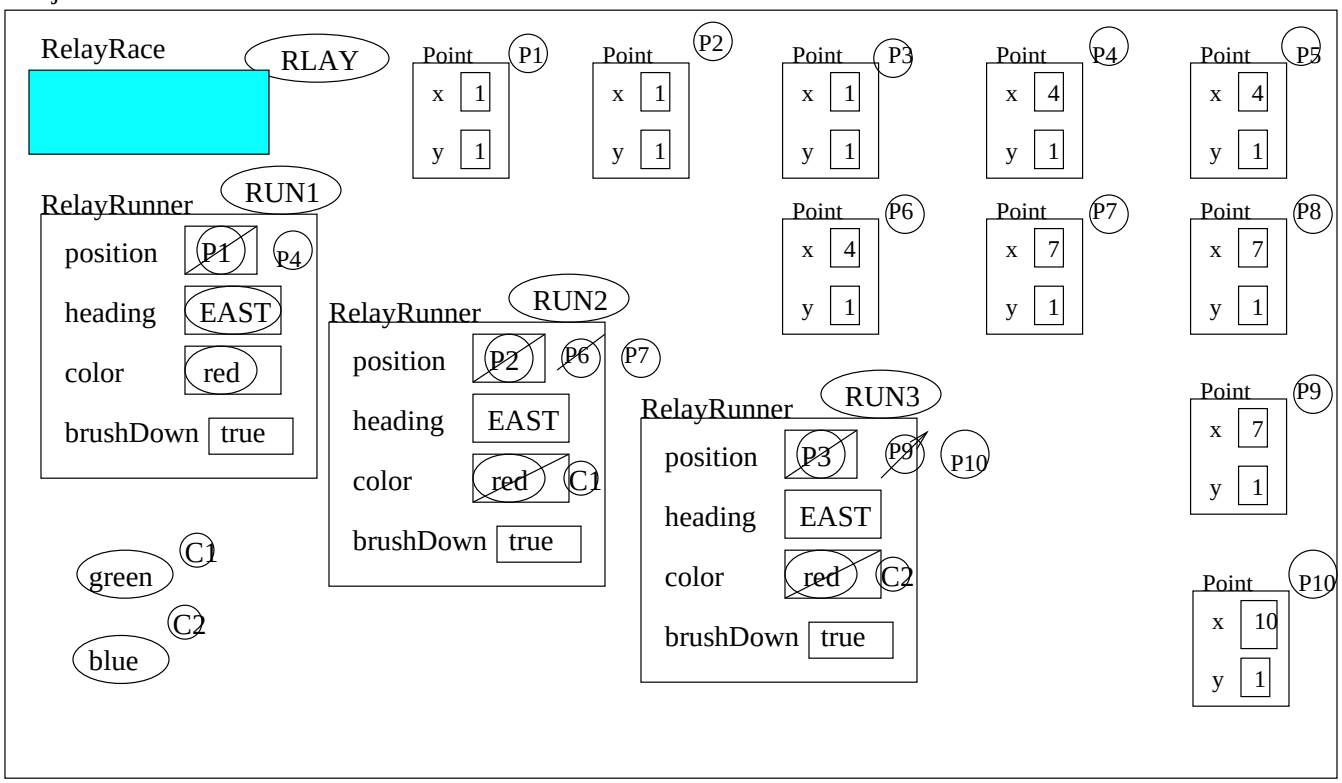
Consider the following two class definitions:

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

You were not required to remember that `getPosition()` and `setPosition()` make copies of points, though that is what actually happens in BuggleWorld.

Object Land



Execution Land