CS111 EXAM 1
Thursday, March 2, 2000

CS111 Spring 2000 Exam 1 Solutions

YOUR NAME: ____

This exam has 13 pages. There are ***three*** problems. Problem 1 has ***two*** parts, problem 2 has ***two*** parts and problem 3 has ***four*** parts. The number of points for each problem and part is shown in square brackets next to the problem or part. There are 100 total points on the exam.

Write all your answers on the exam itself. Ample space is provided. Whenever possible, show your work so that partial credit can be awarded.

The exam is open book. You may refer to the textbook, your notes, and whatever additional materials would be useful.

Please keep in mind the following tips:

- *First skim through the entire exam.* Work first on the problems on which you feel most confident. You do not need to do the problems in the order they are presented.
- *Try to do something on every problem* so that you can receive partial credit. For programming problems, you can receive partial credit for explaining your strategy with words and pictures.
- *Show your work*, so that you can receive partial credit even if the final answer is incorrect.
- *Allocate your time carefully*. If you are taking too long on a problem, wrap it up and move on.
- If you finish early, *go back and check your answers*.

GOOD SKILL!

The following table will be used in grading your exam:

Problem	**Score**
Problem 1 a **[10 points]**	
Problem 1 b **[10 points]**	
Problem 2 a **[20 points]**	
Problem 2 b **[15 points]**	
Problem 3 a **[25 points]**	
Problem 3 b **[10 points]**	
Problem 3 c **[5 points]**	
Problem 3 d **[5 points]**	
Total	

Problem 1 [20 points]

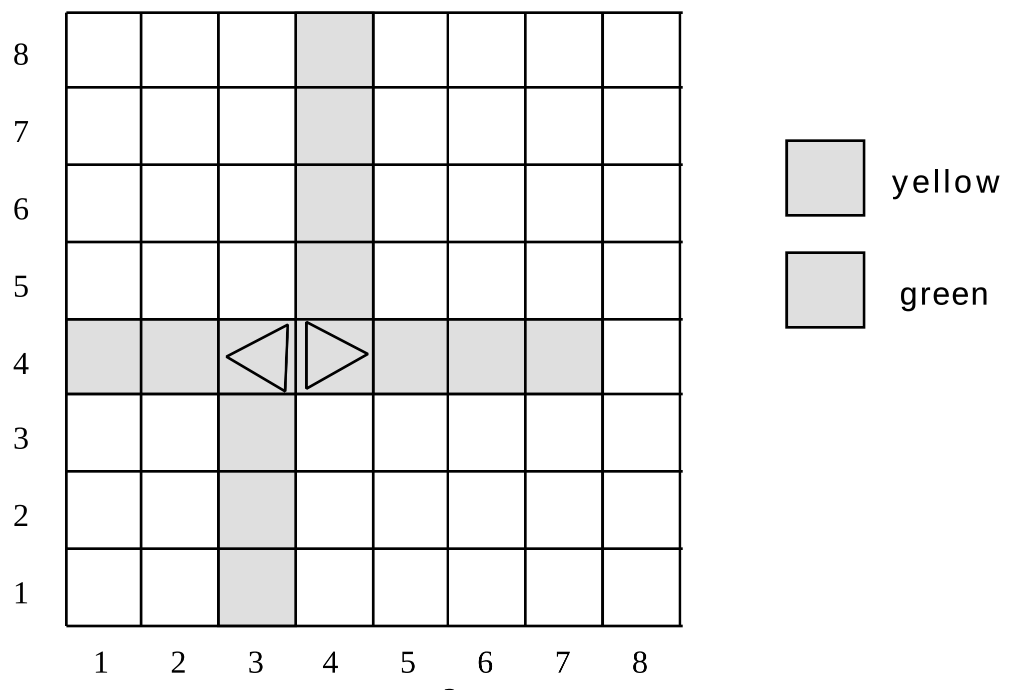
Suppose you are given the following code:

```
 public class RightAngleWorld extends BuggleWorld {
  public void run() {
    RightAngleBuggle rhonda = new RightAngleBuggle(); // line 1
    RightAngleBuggle reggie = new RightAngleBuggle(); // line 2

    rhonda.setPosition(new Point(3,4));               // line 3
    reggie.setPosition(new Point(4,4));               // line 4
    rhonda.left();                                    // line 5
    rhonda.left();                                    // line 6

    rhonda.setColor(Color.yellow);                    // line 7
    rhonda.forward(2);                                // line 8
    rhonda.backward(2);                               // line 9
    rhonda.left();                                    // line 10
    rhonda.forward(3);                                // line 11
    rhonda.backward(3);                               // line 12
    rhonda.right();                                   // line 13

    reggie.setColor(Color.green);                     // line 14
    reggie.forward(3);                                // line 15
    reggie.backward(3);                               // line 16
    reggie.left();                                    // line 17
    reggie.forward(4);                                // line 18
    reggie.backward(4);                               // line 19
    reggie.right();                                   // line 20
}
```

Also, assume you are given the following `RightAngleBuggle` class declaration. It is initially empty. Note that under these conditions run() works properly.

```
  class RightAngleBuggle extends Buggle {
   // body of class declaration will be filled in here in part a.
   }
```

The code produces the following output:



Part a) [10 points] Notice that there is a pattern to the code in the `run()` method. Specifically, lines 7-13 and 14-20 both draw a "right angle", although there are a few differences. Write an instance method in the `RightAngleBuggle` class named **`rightAngle`** to capture this pattern. The **`rightAngle`** method should take two parameters: a color and a length. Be sure to write your method in such a way that it will work for any color and length.

Notes:

- There are several correct solutions to this problem; any such solution will receive full credit.
- Here and elsewhere in this exam, you have been provided with much more space than you need for your solution.

```
class RightAngleBuggle extends Buggle {

// fill in body of class declaration below

   public void rightAngle(Color c, int length){

     this.setColor(c);
     this.forward(length);
     this.backward(length);
     this.left();
     this.forward(length+1);
     this.backward(length+1);
     this.right();
   }

}
```

Part b) [10 points] Given the instance method defined in part a), show what instance method invocation would replace the code in lines 7-13:

```
rhonda.rightAngle(Color.yellow, 2);
```

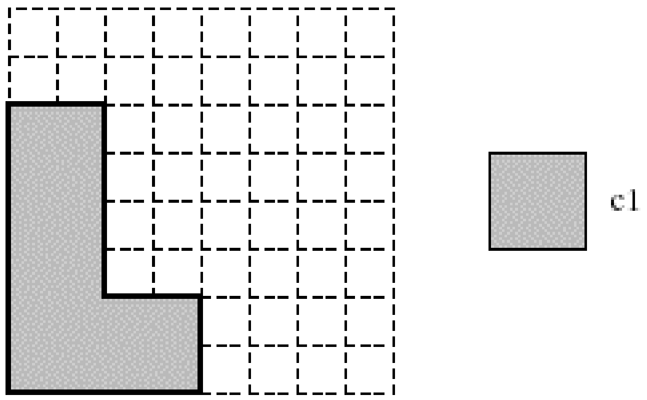
Similarly, show what instance method invocation would replace the code in lines 14 - 20:

```
reggie.rightAngle(Color.green, 3);
```

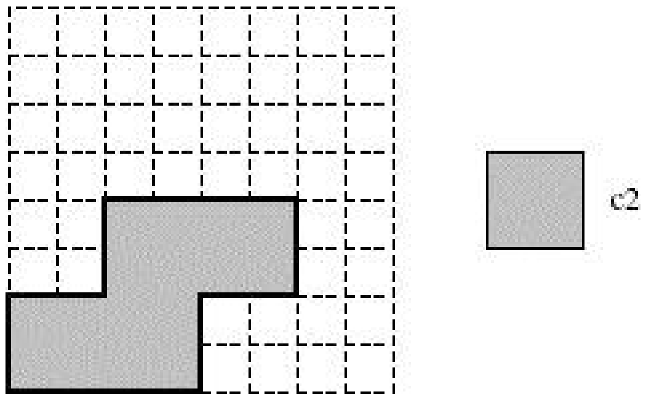
Problem 2 [35 points]

In this problem you will be working with the following three pictures from the video game Tetris:

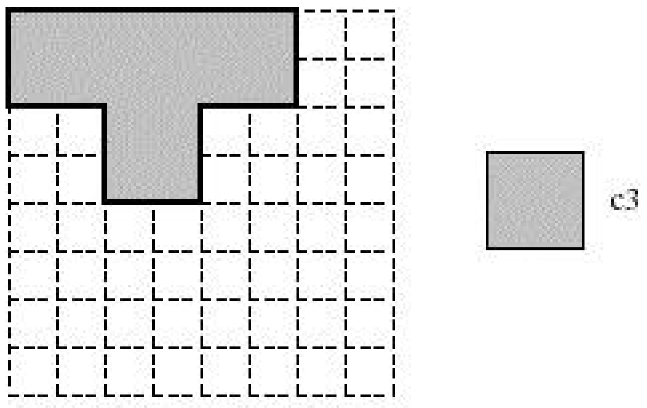
`public` `Picture L (Color c1)`
Returns the L-shaped picture depicted below. It has a black border and is filled with color `c1`. (Here and below, the dashed grid lines are not part of the picture. They are provided to show you the coordinates of points within the pictures.)



`public` `Picture S (Color c2)`
Returns the S-shaped picture depicted below. It has a black border and is filled with color `c2`.



`public` `Picture T (Color c3)`
Returns the T-shaped picture depicted below. It has a black border and is filled with color `c3`.



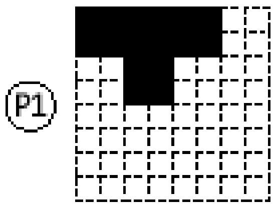
There are two parts to this problem. Please go on to the next page for part a.

Part a) [20 points]

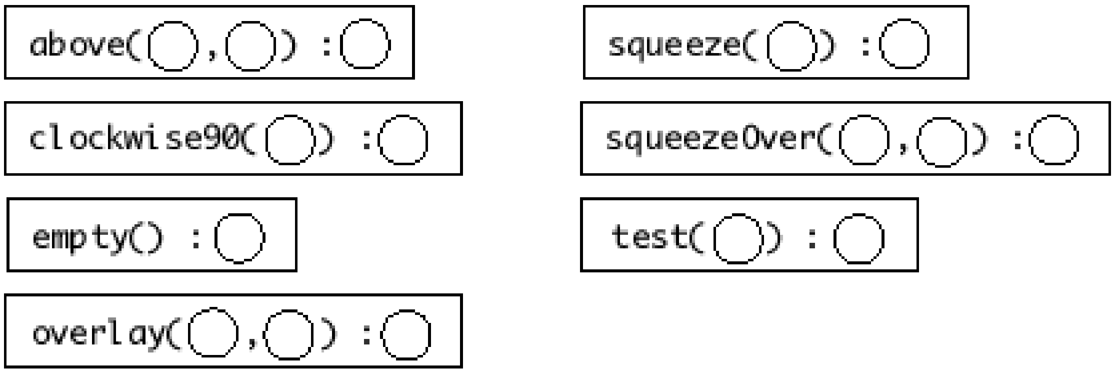
Suppose that the following methods are defined within `TetrisWorld`, a subclass of `PictureWorld`.

```
public Picture test (Picture pic) {
        return squeezeOver(clockwise90(pic), pic);
}

public Picture squeezeOver(Picture pic1, Picture pic2) {
        return overlay(squeeze(pic1), pic2);
}

public Picture squeeze (Picture pic) {
        return above(empty(), pic);
}
```

Suppose that **P1** is an object reference for the following T-shaped Picture object:



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



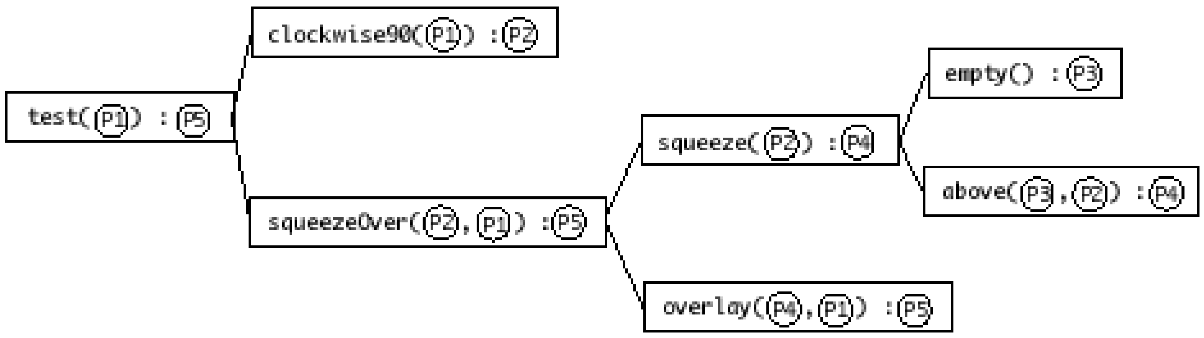
The empty circles in the nodes are skeletons for object references that you should fill in with one of the labels **P1**, **P2**, **P3**, **P4**, **P5** to refer to the appropriate `Picture` instance in Object Land (see below). A circle enclosed by parentheses is a reference to an actual argument of 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 `test()` 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 five `Picture` instances that are used during the execution. The picture labeled **P1** has already been drawn for you; you should draw the pictures **P2**, **P3**, **P4**, and **P5**.

(Note: for simplicity, the receiver object for each of the method invocations has been omitted. The omitted receiver would be an instance of the `TetrisWorld` class. This instance has also been omitted from Object Land.)

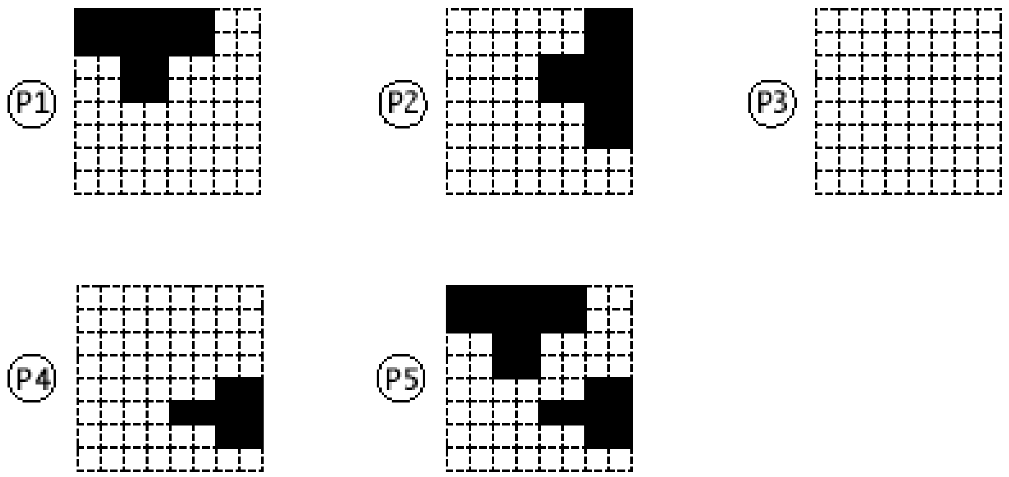
Execution Land

(draw your invocation tree here)

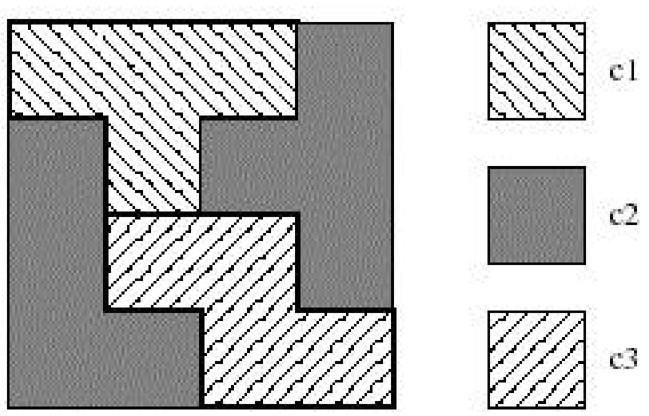


Object Land

(draw your objects here)



Part b) [15 points] In this part, you are to write a `square()` method that, when called on three color arguments, returns the following `Picture`:



Below, you have been provided with a skeleton for the `square()` method, whose body you should flesh out. In your method body, you may use the `L()`, `S()`, and `T()` methods described at the beginning of this problem, as well as any of the methods in the `PictureWorld` contract (i.e., `above()` , `beside()`, `clockwise90()`, `clockwise180()`, `clockwise270()`, `empty()`, `flipDiagonally()`, `flipHorizontally()`, `flipVertically()`, and `overlay()`.)

```
public Picture square(Color c1, Color c2, Color c3) {

  // This is one of many solutions. Other solutions can be obtained
  // permuting the order of the arguments to the overlays.

  return overlay(T(c1),
                 overlay(clockwise90(T(c2)),
                         overlay(L(c2),
                                 flipVertically(S(c3)))));
}
```

Problem 3 [45 points]

Below are the declarations of two classes: a `DashBuggleWorld` class that is a subclass of `BuggleWorld` and a `DashBuggle` class that is a subclass of `Buggle`.

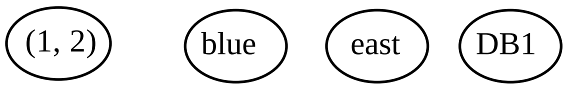
```
class DashBuggleWorld extends BuggleWorld {
  public void run() {
    DashBuggle jed = new DashBuggle();
    DashBuggle ned = new DashBuggle();
    Point jp = new Point(4,7);
    Point np = new Point(5,3);
    jed.drawDash(jp, Color.blue, Direction.NORTH, 3);
    ned.drawDash(np, Color.green, Direction.WEST, 2);

  }
}

class DashBuggle extends Buggle {
  public void drawDash(Point p, Color c, Direction d,int n) {
    this.setPosition(p);
    this.setColor(c);
    this.setHeading(d);
    this.forward(n);
    this.brushUp();
    this.backward(3 * n);
    this.brushDown();
    this.forward(n);
    }
}
```

Part a) [25 points] Suppose that Object Land contains an instance of the DashBuggleWorld class that has the reference label **DW1.** On the next page, use the skeleton of the Java Execution Model to draw an execution diagram for the execution of the statement

```
DW1.run();
```

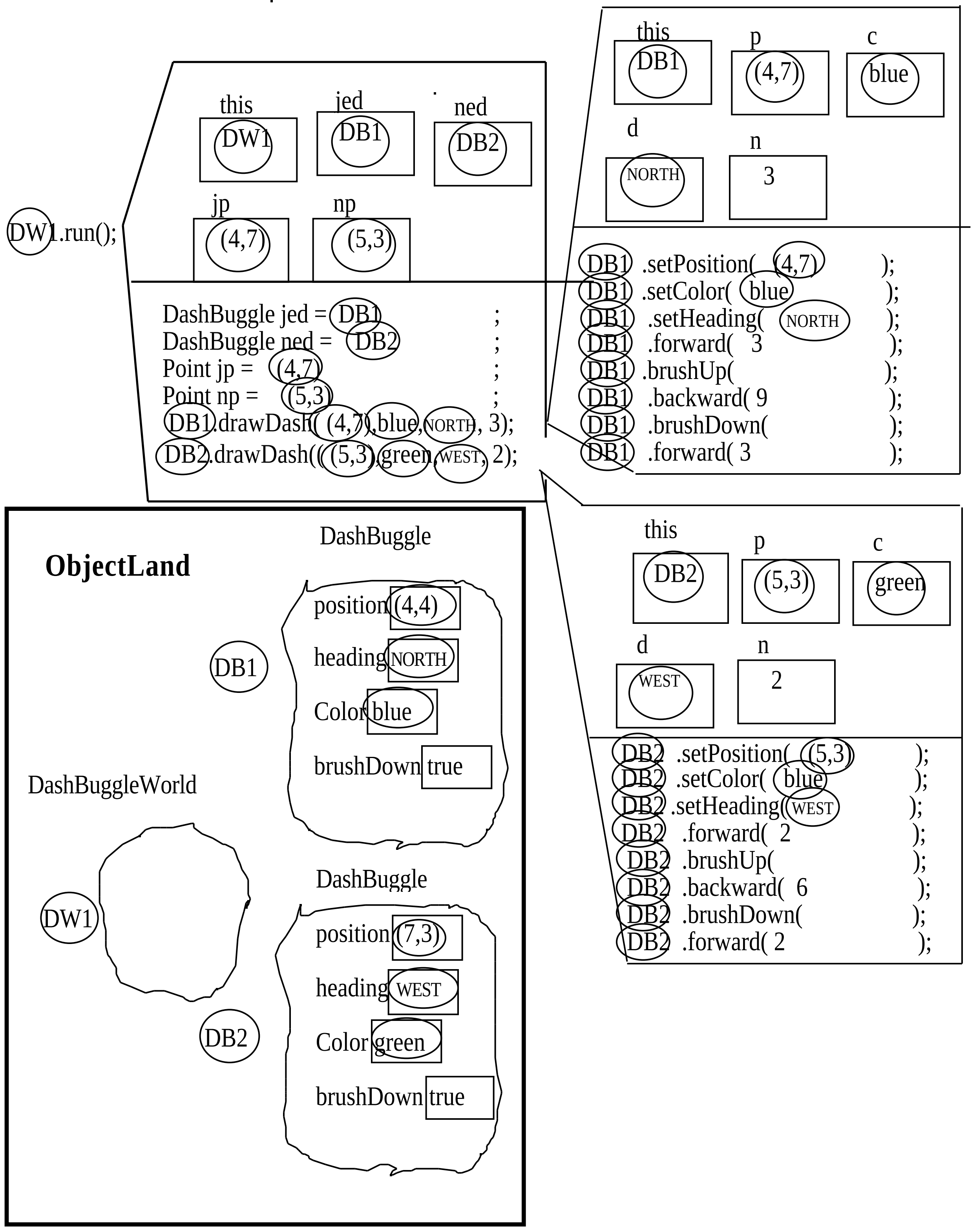
Pay attention to the following notes:

- You should follow the established conventions for drawing execution diagrams.
- Your Object Land should show three objects: the given instance of the `DashBuggleWorld` class (**DW1**) and two instances of the `DashBuggle` class. Label your `DashBuggles` **DB1** and **DB2**. You need not show any `Point`, `Color` or `Direction` instances in Object Land. You should draw references to instances of the `Point`, `Color`, `Direction`, and `LBuggle` classes as oval surrounding information identifying the instance. For `Point`, this information should be parenthesized coordinates; for `Color`, this should be the name of the color; for `Direction`, this should be the name of the direction; for `DashBuggle`, this should be the reference label. For example:



- Your diagram should show the state of each object after the completion of the execution of the `run()` method.
- The next page provides a skeleton to save you time in drawing your diagram. You must fill in the skeleton where appropriate.
- Please try to make your diagram as neat as possible.

Execution Land

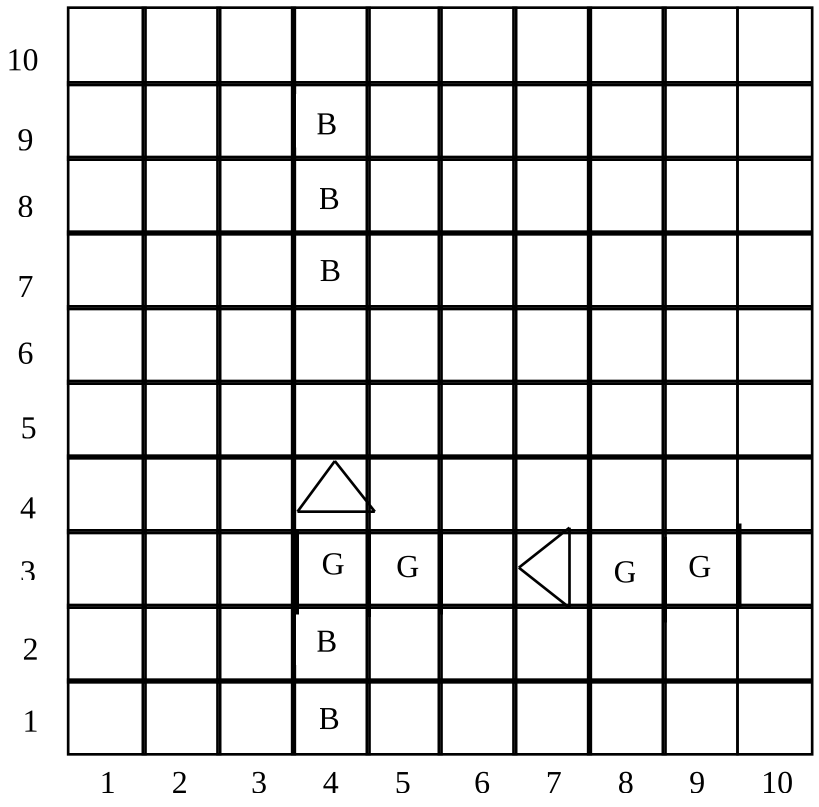


```
DW1.run();

DashBuggle jed = DB1                  ;
DashBuggle ned = DB2                  ;
Point jp = (4,7)                      ;
Point np = (5,3)                      ;
DB1.drawDash( (4,7),blue,NORTH, 3);
DB2.drawDash( (5,3),green,WEST, 2);
```

```
DB1 .setPosition( (4,7)       );
DB1 .setColor( blue           );
DB1 .setHeading( NORTH        );
DB1 .forward(  3              );
DB1 .brushUp(                 );
DB1 .backward( 9              );
DB1 .brushDown(               );
DB1 .forward( 3               );
```

```
DB2 .setPosition( (5,3)       );
DB2 .setColor( blue           );
DB2 .setHeading(WEST          );
DB2 .forward(  2              );
DB2 .brushUp(                 );
DB2 .backward(  6             );
DB2 .brushDown(               );
DB2 .forward( 2               );
```

Part b) [10 points] A grid is provided below, in which you should sketch the pattern that is drawn by the `DashBuggles`. In your sketch, you should indicate that a square has been colored the color blue by the letter **B**, and been colored the color green by the letter **G.** (Remember, a buggle doesn't color a cell in the grid until it moves out of the cell.) You should also indicate the final position and direction of each `DashBuggle` in your diagram. To do so, represent each `DashBuggle` by a triangle and place the triangle in the grid cell of the DashBuggle's final position. Also, the triangle should point in the direction (North, South, East or West) in which the DashBuggle ends up heading. Do not worry about indicating the color of the buggle when it is in its final position.



Part c) [5 points] Suppose the `drawDash()` method is rewritten such that a `DashBuggle` parameter is added, as is shown below. The invocations of `drawDash()` in the `run()` method are modified to reflect this change. Please note that these are the *only* changes to the original code.

```
class DashBuggleWorld extends BuggleWorld {
    public void run() {
      DashBuggle jed = new DashBuggle();
      DashBuggle ned = new DashBuggle();
      Point jp = new Point(4,7);
      Point np = new Point(5,3);
      // The following two lines have been modified:
      this.drawDash(jed, jp, Color.blue, Direction.NORTH, 3); // Modified
      this.drawDash(ned, np, Color.green, Direction.WEST, 2); // Modified
    }
  }

  class DashBuggle extends Buggle {
    // drawDash() now takes a DashBuggle as an additional first parameter.
    public void drawDash(DashBuggle db, Point p, Color c, Direction d, int n) {
      db.setPosition(p);
      db.setColor(c);
      db.setHeading(d);
      db.forward(n);
      db.brushUp();
      db.backward(3 * n);
      db.brushDown();
      db.forward(n);
      }
  }
```

Will this new piece of code work properly? *Briefly* explain why or why not.

Answer:
No, it will not.

In `run()`, the receiver of the `drawDash()` method is **this**, which points to an instance of `DashBuggleWorld`. However, `drawDash()` is a `DashBuggle` instance method, and will not be recognized by the `DashBuggleWorld` receiver when the method is invoked.

Part d) [5 points] Consider again the `drawDash()` method from part (a) (not the version from part (c)). When the `drawDash()` method is invoked on a `DashBuggle`, it does not end up in its initial state. Suppose that we want to modify the `drawDash()` method so that it satisfies the following invariant:

`drawDash()` invariant:
after executing drawDash(), a buggle should have the same position and heading that it did when drawDash() was initially invoked.

The code below is a version of `drawDash()` that has been rewritten in an attempt to meet the above invariant. It declares a local variable `DashBuggle db` to "remember" the original buggle, and adds two new lines of code to the end of the method, as shown below. Please note that these are the *only* changes to the original code.

```
class DashBuggleWorld extends BuggleWorld {
    public void run() {
      DashBuggle jed = new DashBuggle();
      DashBuggle ned = new DashBuggle();
      Point jp = new Point(4,7);
      Point np = new Point(5,3);
      jed.drawDash(jp, Color.blue, Direction.NORTH,3);
      ned.drawDash(np, Color.green, Direction.WEST,2);

    }
  }

  class DashBuggle extends Buggle {
    public drawDash(Point p, Color c, Direction d,int n) {
      DashBuggle db = this;  // Remember the original buggle
      this.setPosition(p);
      this.setColor(c);
      this.setHeading(d);
      this.forward(n);
      this.brushUp();
      this.backward(3 * n);
      this.brushDown();
      this.forward(n);
      this.setPosition(db.getPosition());  // Reset position
      this.setHeading(db.getHeading());    // Reset heading
    }
```

Does the modified *drawDash()* satisfy the desired invariant?*Briefly* explain why or why not.

Answer:
No, it does not.

When the local variable `db` is set to **this**, a new `DashBuggle` object is not created; rather, `db` merely *points* to the `DashBuggle` which is the receiver of the method. So, `db` does not retain the original position and heading of the receiver as the receiver is modified in the method, and therefore can not be used to restore those values at the end of the method.