Dot Patterns Laboratory Tasks

Overview

In this laboratory, you will draw a series of 16 dot patterns using nested for loops. You may think of the dot patterns as geometric puzzles to be solved using loop structures. The 16 dot patterns are shown below:

As you can see, when the All Patterns button is clicked, the drawing window will show all patterns that are available. As you develop your program and complete the definition of the various patterns, more and more patterns will be displayed when this button is clicked. You will also be able to display each pattern individually in an enlarged and annotated version.
Project Files

The Dot Patterns Laboratory will be programmed in Java. You will be given three Java files to work with:

- **DotPattern.java**  
  The class in which the patterns are defined
- **DotWidget.java**  
  The class that controls the drawing of dots
- **DotsApplication.java**  
  The class that controls the graphical user interface

In addition, you will find two auxiliary files:

- **jpt.jar**  
  The compiled Java Power Tools user interface library
- **Dots.mcp**  
  The Metrowerks project that organizes the Dots files

Since the focus of this lab is loops, you will work entirely in the file **DotPattern.java**. You do not need to modify **DotWidget.java** or **DotsApplication.java** in any way but of course you will wish to learn how these files work to some degree.

Project Context

If you run the sample solution program, **Dots.exe**, you will see Pattern 1, Square By Rows, drawn in the graphics panel. The dot pattern should look like the picture below (except that we have reduced the size and have added row and column annotations).
Let us examine the crucial section of Java code in the file `DotPattern.java` that provides the definition of this pattern:

```java
protected final static DotPattern squareByRows =
    new DotPattern() {
        protected void draw() {
            for (int row = 0; row <= 7; row++)
                for (int col = 0; col <= 7; col++)
                    dot.draw(row, col);
    }
};
```

If you look inside this code, you will see that the heart of the definition is found in the `draw()` method in which the sequence of dots to draw is determined by the structure of the `for` loops:

```java
for (int row = 0; row <= 7; row++)
    for (int col = 0; col <= 7; col++)
        dot.draw(row, col);
```

The outer loop is controlled by the `row` parameter and the inner loop is controlled by the `col` (or column) parameter. The fundamental fact about how nested loops work is that each time the `row` parameter in the outer loop takes on a specific value from 0 to 7 the inner loop is repeated for each value of `col` from 0 to 7. Thus, when `row` is 0, the following 8 pairs of `row, col` indices are processed:

- 0,0 0,1 0,2 0,3 0,4 0,5 0,6 0,7

Similarly, when `row` is 1, the following 8 pairs of `row, col` indices are processed:

- 1,0 1,1 1,2 1,3 1,4 1,5 1,6 1,7

Similarly, when `row` is 2, the following 8 pairs of `row, col` indices are processed:

- 2,0 2,1 2,2 2,3 2,4 2,5 2,6 2,7

This continues until `row` is 7 at which point the following 8 pairs of `row, col` indices are processed:

- 7,0 7,1 7,2 7,3 7,4 7,5 7,6 7,7

If you now go back to the picture of the `Square By Rows` pattern on the previous page, you will see that the annotations in the diagram show that the dot positions have been visited in precisely the order we have just described.

The intellectual essence of the `Dots Pattern Laboratory` is for you to learn how to control loops so perfectly that you can visit a sequence of `row, col` positions in any desired order.
Let us now look more closely at the loop code for the pattern **Square By Rows**:

```java
for (int row = 0; row <= 7; row++)
    for (int col = 0; col <= 7; col++)
        dot.draw(row, col);
```

Notice that the action within the nested loops is the line of code:

```java
dot.draw(row, col);
```

The variable `dot` is an object of class `DotWidget`. This class knows how to draw dots as *diamond shaped objects*, how to annotate the dots with `row, col` pairs if that option is selected in the user interface, and how to connect a new dot to the one that was drawn previously. The `DotWidget` class also arranges that the first dot drawn is colored *red* and all subsequent dots are colored *black*. Thus, the `DotWidget` class organizes all of the technical details of how to graphically represent a sequence of dots as a *pattern*.

Where does the variable `dot` come from? The quick answer is that the `DotPattern` class creates a new `dot` object whenever a new dot pattern is initiated. We will see precisely how this is done in a moment.

Let us first go back to the full definition of the pattern **Square By Rows**:

```java
protected final static DotPattern squareByRows =
    new DotPattern() {
        protected void draw() {
            for (int row = 0; row <= 7; row++)
                for (int col = 0; col <= 7; col++)
                    dot.draw(row, col);
        }
    };
```

The first line of this code says that `squareByRows` is to be defined as a variable of type `DotPattern`. At first this seems quite curious. We are defining a variable of a class inside the definition of the class. Isn’t this a circular definition?

The answer is NO due to the keyword `static`. When an entity within a class is declared `static`, this means that *the entity is not a member of each instance of the class* but is rather an object that is defined once and is shared by any instance of the class. In fact, we could have placed this `static` definition in an entirely separate file and things would have worked pretty much the same.

We decided to place the individual `static` definitions of specific dot patterns within the file `DotPattern.java` for three reasons:
1. Using **static** definitions is a common idiom in Java. Indeed, almost all Java programs are launched in the **main** method by effectively creating a **static** instance of the enclosing class and inserting this instance into a window frame.

2. By placing the pattern definitions within the file **DotPattern.java**, we avoid the proliferation of separate files that can often become a nuisance.

3. As you have seen, the **DotPattern** class provides a mechanism to draw all patterns at once in the graphics panel. This is based on the **drawAll()** method that accesses the list of all available dot patterns that have been defined within the file **DotPattern.java**.

Let us continue examining the full definition of **Square By Rows**:

```java
protected final static DotPattern squareByRows =
    new DotPattern() {
        protected void draw() {
            for (int row = 0; row <= 7; row++)
                for (int col = 0; col <= 7; col++)
                    dot.draw(row, col);
        }
    };
```

The keyword **protected** says that this object will not be accessed directly by other classes. The keyword **final** says that once this object is defined its definition cannot be modified. The keyword **static** says that this object is to be shared. The type **DotPattern** says that this object will be an instance of the **DotPattern** class.

Java demands that *each instance of an object be created* by the **new** operator. This is the purpose of the code following the **:=** sign:

```java
new DotPattern() { ... }
```

What is somewhat unusual in this definition is the code in the braces `{ }`. We want to create a new **DotPattern** but *we need to change the behavior of the generic pattern by providing the explicit code for the draw() method that defines the particular pattern we wish to draw*. Java permits us to do this by allowing us to define or redefine any methods we wish inside braces `{ }` immediately following the constructor **DotPattern()**. In cases where no changes to the methods of a class are required, the braces `{ }` would of course be omitted.

You may have noticed that the **draw()** method is declared **protected** and **abstract** in the class **DotPattern**. The keyword **abstract** means that an implementing object must define the **draw()** method. The keyword **protected** means that the **draw()** method is *not* the one directly called by the **DotsApplication** class. The reason for this is that the **DotsApplication** must pass parameters to specify the *graphics context* of the graphics panel, the size of the area in which to draw, and whether or not to annotate the dots.
Thus, there is a public version of the `draw` method in `DotPattern` that accepts these three parameters:

```java
public final void draw
    (Graphics2D graphics, int paneSize, boolean annotate)
{
    this.graphics = graphics;
    this.paneSize = paneSize;
    this.annotate = annotate;
    dot = new DotWidget(graphics, paneSize, annotate);
    // call the hidden draw method to draw the pattern
draw();
}
```

Notice, first of all, that `DotPattern` in fact has two `draw` methods, one with zero parameters that we have been discussing and the new one with three parameters that we have just introduced. *Java permits a class to have two methods of the same name as long as the usage of the methods can be distinguished by the number or type of the parameters.*

In the three-parameter `draw` method, the `graphics` parameter encapsulates the information that Java needs to draw the pattern into the graphics panel (including any translation needed to shift the position of the pattern); the `paneSize` parameter indicates the size of the area available for drawing the pattern; and the `boolean` parameter `annotate` indicates whether or not to annotate each dot with its `row, col` position.

The first three lines of the `draw` method simply save these parameters into the member data variables of the `DotPattern` object. Since the member data variables have exactly the same names as the method parameters, we must distinguish the member data by adding the specification `this`. For example,

```java
this.graphics = graphics;
```

means assign a reference to the incoming parameter `graphics` of the `draw` method to the corresponding member data variable also called `graphics` of “this dot pattern object”.

The fourth line of the method creates a new `DotWidget` object by passing to it all of the parameters of the `draw` method. This `DotWidget` object is stored in the member data variable `dot` of the `DotPattern` object. In this case, we do not need to say `this.dot` because there is no potential for ambiguity. In general, a method can access a member data variable of its own object without the need for `this` as long as no ambiguity results.

Finally, with the member data of the `DotPattern` object properly initialized, the three-parameter `draw` method calls the zero-parameter `draw` method to do the actual work of drawing the particular dot pattern.
Notice that the purpose of the three-parameter `draw` method is to perform certain steps that must be done before the actual drawing can take place in the zero-parameter `draw` method. These steps are always the same and for this reason the three-parameter `draw` method is declared `final` so that its code cannot be modified. This is an example of a design pattern known as the template method in which the steps of an algorithm are specified in a master method (here the three-parameter `draw` method) and the details that vary are encapsulated separately in methods that are called from the master (here the zero-parameter `draw` method that encapsulates the particular dot pattern).

**Project Programming Tasks**

Your basic programming task is to complete the definition of the 15 patterns that have not yet been defined in the file `DotPattern.java`. Do not change the other two Java files since that is unnecessary and since the grader will only use your `DotPattern.java` file in grading and will ignore any other Java files that you submit.

In `DotPattern.java`, we have begun the definition of Pattern 2, *Square By Cols* but the code is missing a crucial component.

```java
protected final static DotPattern squareByCols = new DotPattern() {
    protected void draw() {
    }
};
```

The `draw()` method has no code and so nothing will be drawn. You must program the code to obtain the following pattern:
This pattern looks very similar to Pattern 1 except that the drawing proceeds down the columns rather than across the rows. If you understood the discussion of Pattern 1, you will recognize that Pattern 1 proceeds across the rows because the row loop is the outer loop and for each row the inner loop moves across the columns along that row. Hence, in order to do Pattern 2, it will be essential to interchange the role of the rows and columns. This is an example of a general principle:

To generate a pattern that proceeds across the rows, the row loop must be the outer loop. To generate a pattern that proceeds across the columns, the column loop must be the outer loop.

One of the most important goals of the Dot Patterns Laboratory is that you will deduce similar general principles that will guide your future algorithmic work involving loops.

Let us now move on to Pattern 3, **Triangle Lower Left**:
Overall the pattern proceeds across the rows so, by the general principle above, we expect the row loop to be the outer loop. However, there is a complication, since the number of dots in each row is not constant. This means that the loop limits used in the inner column loop cannot be constant, that is, they must vary in some way from one row to the next. In other words:

*In a loop across the rows in which the number of dots in each row varies, the loop limits for the inner column loop must depend on or be a function of the row index.*

If we examine the rows for Pattern 3 carefully, we see that:
In row 0, the start column is 0 and the end column is 0
In row 1, the start column is 0 and the end column is 1
In row 2, the start column is 0 and the end column is 2
In row 3, the start column is 0 and the end column is 3

Do you see the numerical pattern? Can you turn this numerical pattern into the code for Pattern 3?

Once you have the code for Pattern 3 written, you need to know how to let the user interface know that you have provided a new pattern. To simplify things, we have collected references to the static dot pattern variables into a static array variable in \textit{DotPattern.java}:

\begin{verbatim}
protected final static DotPattern[] patternArray = {
    allPatterns,
    squareByRows,
    squareByCols /*, 
    triangleLowerLeft; 
    triangleUpperRight, 
    triangleUpperLeft, 
    triangleLowerRight, 
    mainDiagonal, 
    diagonalPair, 
    halfButterfly,
    fullButterfly, 
    squareBorder, 
    squareSpiral, 
    triDiagonal, 
    slantWarmup, 
    slantTriangle, 
    squareZigZag */
};

protected final static int MAXIMUM_PATTERN = patternArray.length - 1;
\end{verbatim}

Notice that we have introduced a \textit{block comment} with the symbols /* after Pattern 2 and ended this \textit{block comment} with the symbols */ on the last line of the declaration. To make Pattern 3 visible, you must name the pattern \texttt{triangleLowerLeft} and you must move the /* from line 3 to line 4 being careful to place the comma after /*.

As you finish each pattern, you will continue to move the /* downward until when you are done there is no need for a block comment.
The 16 dot patterns are designed to mimic the looping patterns of many important algorithms in computer science and mathematics. However, some patterns are present simply to provide challenge and amusement. One such is the **Full Butterfly**:

As a warmup for this pattern, you will be asked to first draw only the left half of the pattern. As you attempt to generalize, you will realize that it is not easy to build the full pattern using a single nested loop structure. It is possible but the code is tricky. In this case, a simpler solution is to use two nested loop structures, one after another.

The patterns are set up in such a way that you will be lulled into a *pattern of thinking* as you draw the *patterns of dots*. Then, abruptly, a new pattern of thinking will be required. Be prepared to break out of one thought pattern and move into another. In particular, you will need to figure out how to do patterns that proceed along a *diagonal* rather than along a row or column.
The final pattern, \textit{Square Zig Zag}, is an especially wonderful challenge since the traversals go back and forth:

You may think that this pattern is a mere amusement but if your task was to program a robot to visit all locations in a warehouse then this method of traversal is far more efficient than the earlier patterns \textit{Square By Rows} or \textit{Square by Cols}.

\textbf{Questions To Answer}

In addition to the programming tasks for the \textit{Dot Patterns Laboratory}, we want you to examine the Java code supplied and answer the following questions. Your answers should be typed into a file and submitted along with your \texttt{DotPattern.java} file. As you answer the questions, you may cut-and-paste code fragments from the Java code into your document.

\textit{It is not expected that you will understand every detail of the code.} But, just as in learning a foreign language, you can grasp the purpose and meaning of something written in the language long before you understand the nuances and can write fluently in that language yourself.

1. Geometry Questions

What is the size in pixels of each side of the graphics panel?
What is the amount of space between two adjacent dots in pixels?
How much space in pixels separates each edge of the graphics panel from the dots?
2. How does `DotWidget.java` arrange that the first dot is red but all later dots are black?

3. How does `DotsApplication.java` learn about what dots patterns are available in `DotPattern.java` and what `String`'s should be used as names for the radio buttons in the user interface? In other words, what methods are called?

4. Radio Button Questions

   What lines of code create the radio button panel?
   What lines of code insert the radio button panel into the user interface?
   In what area of the `BorderLayout` are the radio buttons installed?
   What lines of code determine which radio button is currently selected?

5. The user interface has a “listener” that checks periodically to see if the user has changed either the radio button selection or the annotate check box. Cut and paste the code that handles this. Then explain as best you can what is happening in the `run()` method of the associated `Thread`.

**Grading Scheme and Extra Credit**

15  1 point per pattern that is programmed correctly and with adequate style.
5   You may invent up to 5 additional “interesting” patterns for 1 point each.
5   Up to 5 points for programming in a high quality, readable style.
10  Up to 10 points for the answers to the 5 questions.

35  Maximum total grade

**Comments:**

1. You may not simply draw a pattern by a list of specific calls for individual dots. In other words, code such as

   ```java
dot.draw(0,0); dot.draw(0,0); dot.draw(0,0); dot.draw(0,0); dot.draw(0,1); dot.draw(0,1); dot.draw(0,1); dot.draw(0,1); ...
```

   is totally unacceptable.

2. If the code for a pattern is especially horrible, you may lose credit for the pattern even if it is drawn correctly.

3. Look at the Java files you have been given for this assignment to see a model for high quality style with appropriate naming, indentation, and comments.