The output we want to produce is the following:

```
+------+
|\..../|
| \../ |
|  \/  |
|  /\  |
| /..\ |
|/....\|
+------+
```

Problem Decomposition and Pseudocode

To generate this figure, you have to first break it down into subfigures. In doing so, you should look for lines that are similar in one way or another. The first and last lines are exactly the same. The three lines after the first line all fit one pattern, and the three lines after that fit another:

```
+------+
```
line

```
|\..../|
| \../ |
|  \/  |
```
top half

```
|  /\  |
| /..\ |
|/....\|
```
bottom half

```
+------+
```
line

Thus, you can break down the overall problem as follows:

```
draw a solid line.
draw the top half of the hourglass.
draw the bottom half of the hourglass.
draw a solid line.
```

You should solve each subproblem independently. Eventually you'll want to incorporate a class constant to make the program more flexible, but let's first solve the problem without worrying about the use of a constant.

The solid line task can be further specified as

```
write a plus on the output line.
write 6 dashes on the output line.
write a plus on the output line.
go to a new output line.
```

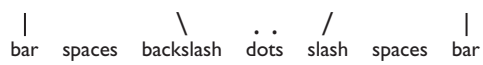
This set of instructions translates easily into a static method:

```
public static void drawLine() {
    System.out.print("+");
    for (int i = 1; i <= 6; i++) {
        System.out.print("-");
    }
    System.out.println("+");
}
```

The top half of the hourglass is more complex. Here is a typical line:

```
| \../ |
```

There are four individual characters, separated by spaces and dots.



Thus, a first approximation in pseudocode might look like this:

```
for (each of 3 lines) {
    write a bar on the output line.
    write some spaces on the output line.
    write a backslash on the output line.
    write some dots on the output line.
    write a slash on the output line.
    write some spaces on the output line.
    write a bar on the output line.
    go to a new line of output.
}
```

Again, you can make a table to figure out the required expressions. Writing the individual characters will be easy enough to translate into Java, but you need to be more specific about the spaces and dots. Each line in this group contains two sets of spaces and one set of dots. Table 2.9 shows how many to use.

The two sets of spaces fit the rule `(line – 1)`, and the number of dots is `(6 – 2 * line)`. Therefore, the pseudocode should read

```
for (line going 1 to 3) {
    write a bar on the output line.
    write (line – 1) spaces on the output line.
    write a backslash on the output line.
    write (6 – 2 * line) dots on the output line.
    write a slash on the output line.
    write (line – 1) spaces on the output line.
    write a bar on the output line.
    go to a new line of output.
}
```

Table 2.9 Analysis of Figure

Line	Spaces	Dots	Spaces
1	0	4	0
2	1	2	1
3	2	0	2

Initial Structured Version

The pseudocode for the top half of the hourglass is easily translated into a static method called `drawTop`. A similar solution exists for the bottom half of the hourglass. Put together, the program looks like this:

```
1  public class DrawFigure {
2      public static void main(String[] args) {
3          drawLine();
4          drawTop();
5          drawBottom();
6          drawLine();
7      }
8
9      // produces a solid line
10     public static void drawLine() {
11         System.out.print("+");
12         for (int i = 1; i <= 6; i++) {
13             System.out.print("—");
14         }
15         System.out.println("+");
16     }
17
18     // produces the top half of the hourglass figure
19     public static void drawTop() {
20         for (int line = 1; line <= 3; line++) {
21             System.out.print("|");
22             for (int i = 1; i <= (line — 1); i++) {
23                 System.out.print(" ");
24             }
25             System.out.print("\\");
26             for (int i = 1; i <= (6 — 2 * line); i++) {
27                 System.out.print(".");
28             }
29             System.out.print("/");
30             for (int i = 1; i <= (line — 1); i++) {
31                 System.out.print(" ");
32             }
```

```
33              System.out.println("|");
34          }
35      }
36
37      // produces the bottom half of the hourglass figure
38      public static void drawBottom() {
39          for (int line = 1; line <= 3; line++) {
40              System.out.print("|");
41              for (int i = 1; i <= (3 - line); i++) {
42                  System.out.print(" ");
43              }
44              System.out.print("/");
45              for (int i = 1; i <= 2 * (line - 1); i++) {
46                  System.out.print(".");
47              }
48              System.out.print("\\");
49              for (int i = 1; i <= (3 - line); i++) {
50                  System.out.print(" ");
51              }
52              System.out.println("|");
53          }
54      }
55  }
```

Adding a Class Constant

The `DrawFigure` program produces the desired output, but it is not very flexible. What if we wanted to produce a similar figure of a different size? The original problem involved an hourglass figure that had three lines in the top half and three lines in the bottom half. What if we wanted the following output, with four lines in the top half and four lines in the bottom half?

```
+--------+
|\....../|
| \..../ |
|  \../  |
|   \/   |
|   /\   |
|  /..\  |
| /....\ |
|/......\|
+--------+
```

Obviously the program would be more useful if we could make it flexible enough to produce either output. We do so by eliminating the magic numbers with the introduction of a class constant. You might think that we need to introduce two constants—one for the height and one for the width—but because of the regularity of this figure,

Table 2.10 Analysis of Different Height Figures

Subheight	Dashes in `drawLine`	Spaces in `drawTop`	Dots in `drawTop`	Spaces in `drawBottom`	Dots in `DrawBottom`
3	6	`line – 1`	`6 – 2 * line`	`3 – line`	`2 * (line – 1)`
4	8	`line – 1`	`8 – 2 * line`	`4 – line`	`2 * (line – 1)`

the height is determined by the width and vice versa. Consequently, we only need to introduce a single class constant. Let's use the height of the hourglass halves:

```
public static final int SUB_HEIGHT = 4;
```

We've called the constant `SUB_HEIGHT` rather than `HEIGHT` because it refers to the height of each of the two halves, rather than the figure as a whole. Notice how we use the underscore character to separate the different words in the name of the constant.

So, how do we modify the original program to incorporate this constant? We look through the program for any magic numbers and insert the constant or an expression involving the constant where appropriate. For example, both the `drawTop` and `drawBottom` methods have a `for` loop that executes 3 times to produce 3 lines of output. We change this to 4 to produce 4 lines of output, and more generally, we change it to `SUB_HEIGHT` to produce `SUB_HEIGHT` lines of output.

In other parts of the program we have to update our formulas for the number of dashes, spaces, and dots. Sometimes we can use educated guesses to figure out how to adjust such a formula to use the constant. If you can't guess a proper formula, you can use the table technique to find the appropriate formula. Using this new output with a subheight of 4, you can update the various formulas in the program. Table 2.10 shows the various formulas.

We then go through each formula and figure out how to replace it with a new formula involving the constant. The number of dashes increases by 2 when the subheight increases by 1, so we need a multiplier of 2. The expression `2 * SUB_HEIGHT` produces the correct values. The number of spaces in `drawTop` does not change with the subheight, so the expression does not need to be altered. The number of dots in `drawTop` involves the number 6 for a subheight of 3 and the number 8 for a subheight of 4. Once again we need a multiplier of 2, so we use the expression `2 * SUB_HEIGHT – 2 * line`. The number of spaces in `drawBottom` involves the value 3 for a subheight of 3 and the value 4 for a subheight of 4, so the generalized expression is `SUB_HEIGHT – line`. The number of dots in `drawBottom` does not change when subheight changes.

Here is the new version of the program with a class constant for the subheight. It uses a `SUB_HEIGHT` value of `4`, but we could change this to `3` to produce the smaller version or to some other value to produce yet another version of the figure.

```
1  public class DrawFigure2 {
2      public static final int SUB_HEIGHT = 4;
3
4      public static void main(String[] args) {
5          drawLine();
```

```
6           drawTop();
7           drawBottom();
8           drawLine();
9       }
10
11      // produces a solid line
12      public static void drawLine() {
13          System.out.print("+");
14          for (int i = 1; i <= (2 * SUB_HEIGHT); i++) {
15              System.out.print("-");
16          }
17          System.out.println("+");
18      }
19
20      // produces the top half of the hourglass figure
21      public static void drawTop() {
22          for (int line = 1; line <= SUB_HEIGHT; line++) {
23              System.out.print("|");
24              for (int i = 1; i <= (line - 1); i++) {
25                  System.out.print(" ");
26              }
27              System.out.print("\\");
28              int dots = 2 * SUB_HEIGHT - 2 * line;
29              for (int i = 1; i <= dots; i++) {
30                  System.out.print(".");
31              }
32              System.out.print("/");
33              for (int i = 1; i <= (line - 1); i++) {
34                  System.out.print(" ");
35              }
36              System.out.println("|");
37          }
38      }
39
40      // produces the bottom half of the hourglass figure
41      public static void drawBottom() {
42         for (int line = 1; line <= SUB_HEIGHT; line++) {
43             System.out.print("|");
44             for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
45                 System.out.print(" ");
46             }
47             System.out.print("/");
48             for (int i = 1; i <= 2 * (line - 1); i++) {
49                 System.out.print(".");
50             }
```

```
51              System.out.print("\\");
52              for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
53                  System.out.print(" ");
54              }
55              System.out.println("|");
56          }
57      }
58  }
```

Notice that the `SUB_HEIGHT` constant is declared with class-wide scope, rather than locally in the individual methods. While localizing variables is a good idea, the same is not true for constants. We localize variables to avoid potential interference, but that argument doesn't hold for constants, since they are guaranteed not to change. Another argument for using local variables is that it makes static methods more independent. That argument has some merit when applied to constants, but not enough. It is true that class constants introduce dependencies between methods, but often that is what you want. For example, the three methods in `DrawFigure2` should not be independent of each other when it comes to the size of the figure. Each subfigure has to use the same size constant. Imagine the potential disaster if each method had its own `SUB_HEIGHT`, each with a different value—none of the pieces would fit together.

Further Variations

The solution we have arrived at may seem cumbersome, but it adapts more easily to a new task than does our original program. For example, suppose that you want to generate the following output:

```
+----------+
|\......../|
| \....../ |
|  \..../  |
|   \../   |
|    \/    |
|    /\    |
|   /..\   |
|  /....\  |
| /......\ |
|/........\|
+----------+
|    /\    |
|   /..\   |
|  /....\  |
| /......\ |
|/........\|
|\......../|
| \....../ |
|  \..../  |
|   \../   |
|    \/    |
+----------+
```

This output uses a subheight of 5 and includes both a diamond pattern and an X pattern. You can produce this output by changing the SUB_HEIGHT constant to 5:

```
public static final int SUB_HEIGHT = 5;
```

and rewriting the main method as follows to produce both the original X pattern and the new diamond pattern, which you get simply by reversing the order of the calls on the two halves:

```
public static void main(String[] args) {
    drawLine();
    drawTop();
    drawBottom();
    drawLine();
    drawBottom();
    drawTop();
    drawLine();
}
```

Chapter Summary

Java groups data into types. There are two major categories of data types: primitive data and objects. Primitive types include `int` (integers), `double` (real numbers), `char` (individual text characters), and `boolean` (logical values).

Values and computations are called expressions. The simplest expressions are individual values, also called literals. Some example literals are: `42`, `3.14`, `'Q'`, and `false`. Expressions may contain operators, as in `(3 + 29) – 4 * 5`. The division operation is odd in that it's split into quotient (`/`) and remainder (`%`) operations.

Rules of precedence determine the order in which multiple operators are evaluated in complex expressions. Multiplication and division are performed before addition and subtraction. Parentheses can be used to force a particular order of evaluation.

Data can be converted from one type to another by an operation called a cast.

Variables are memory locations in which values can be stored. A variable is declared with a name and a type. Any data value with a compatible type can be stored in the variable's memory and used later in the program.

Primitive data can be printed on the console using the `System.out.println` method, just like text strings. A string can be connected to another value (concatenated) with the + operator to produce a larger string. This feature allows you to print complex expressions including numbers and text on the console.

A loop is used to execute a group of statements several times. The `for` loop is one kind of loop that can be used to apply the same statements over a range of numbers or to