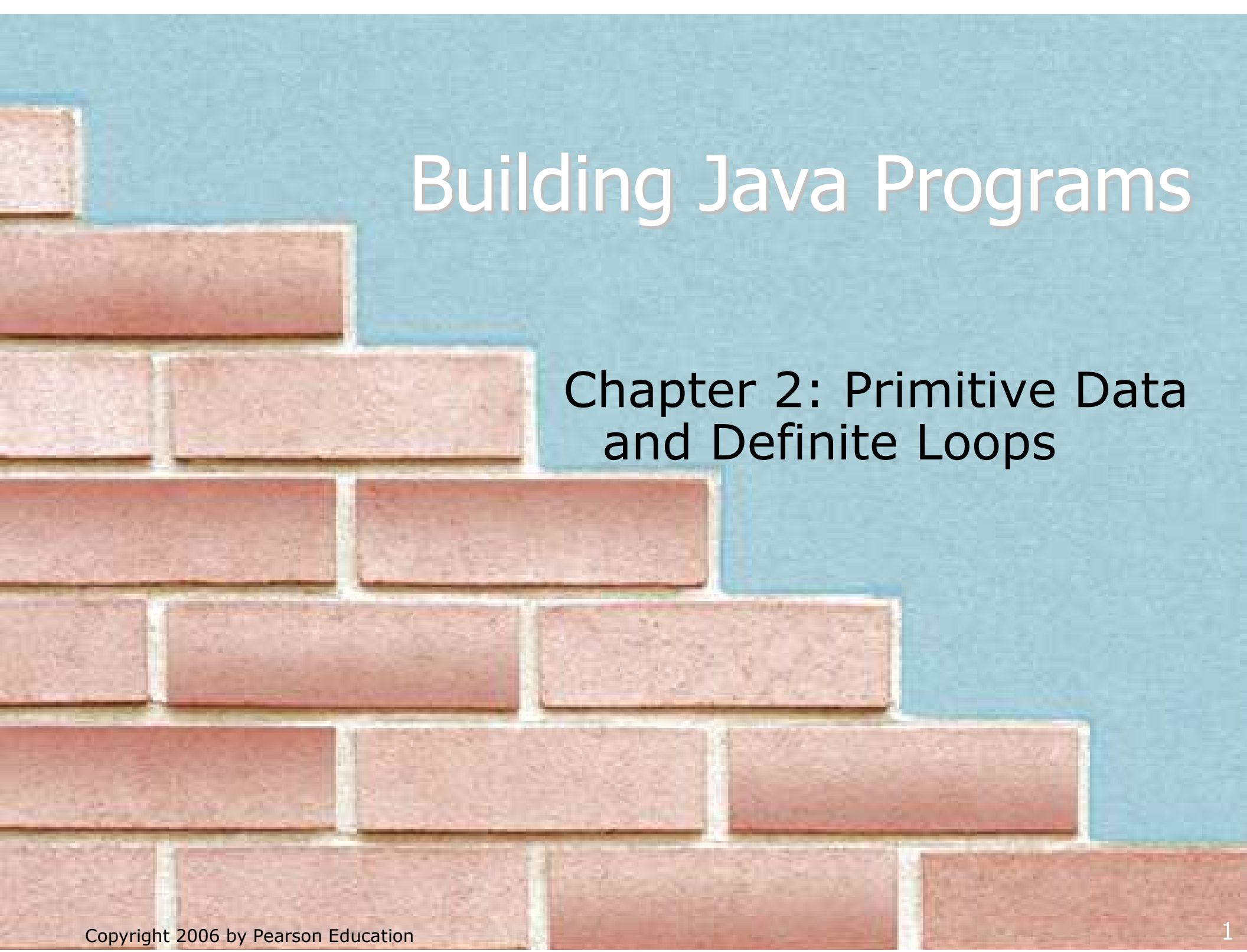
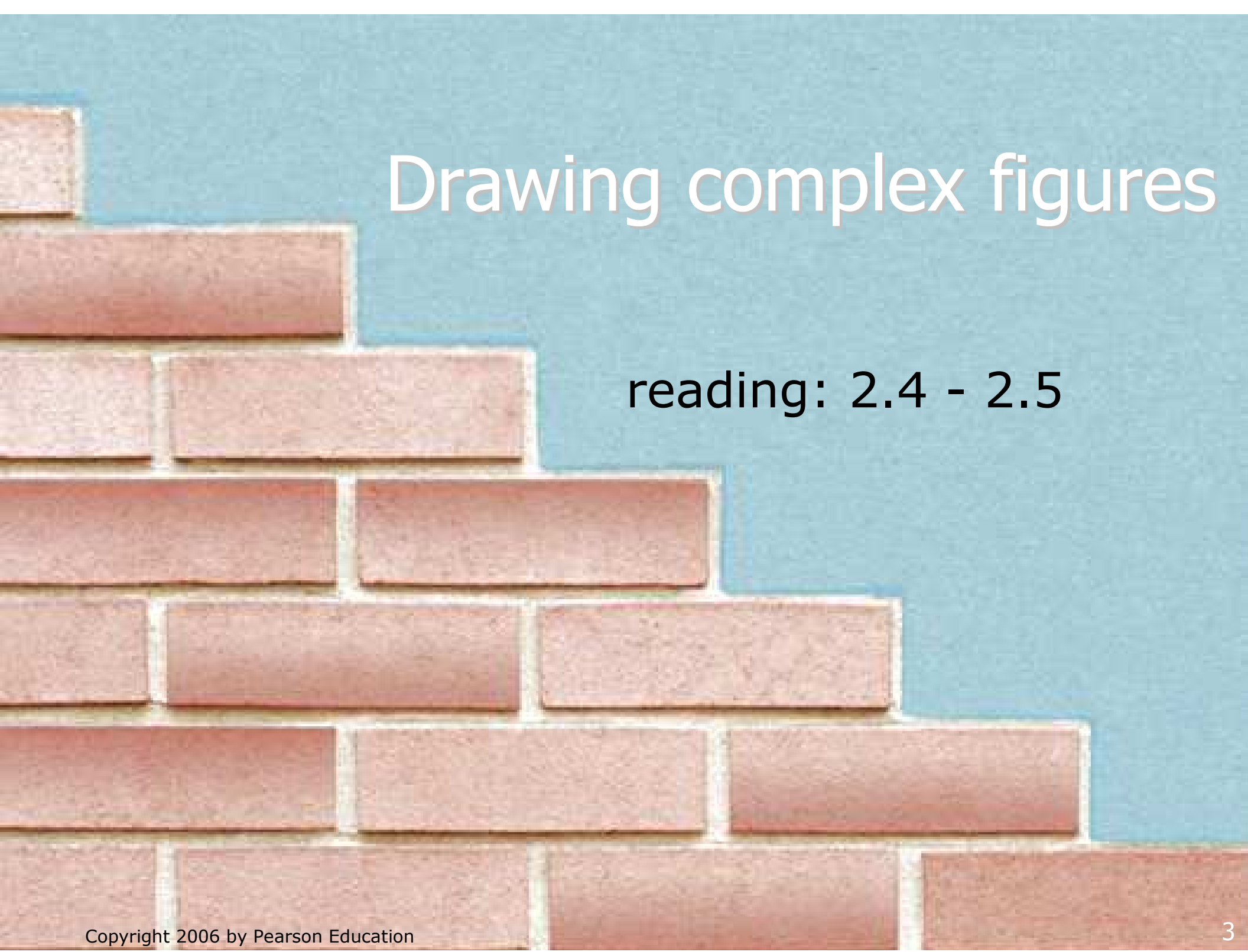


Lecture outline

- managing complexity
  - variable scope
  - class constants
- drawing complex figures with `for` loops



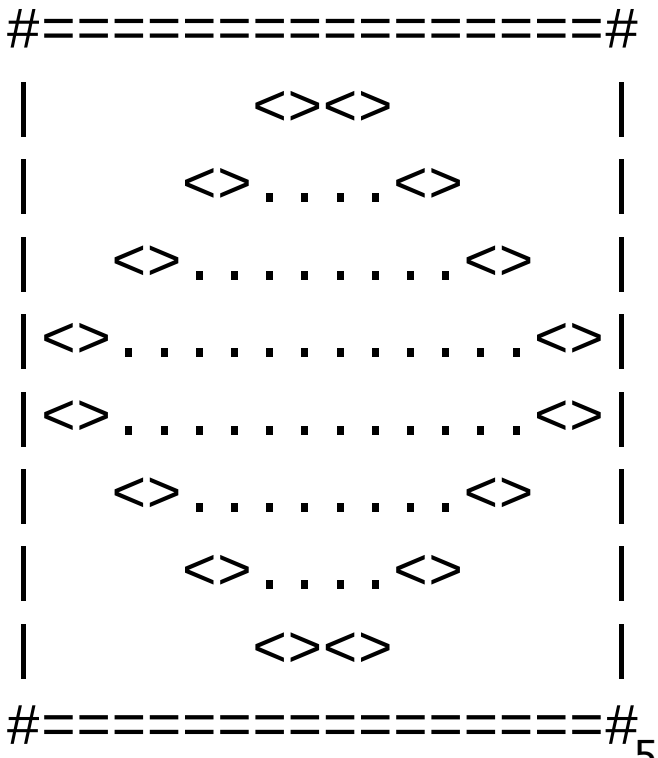
Drawing complex figures

- Write a program that produces the following output.
  - Use nested `for` loops to capture the repetition.

```
#==================#
|        <><>       |
|     <>....<>      |
|   <>........<>    |
|<>..............<>|
|<>..............<>|
|   <>........<>    |
|     <>....<>      |
|        <><>       |
#==================#
```

Drawing complex figures

- When the task is as complicated as this one, it may help to write down steps on paper before we write our code:
  - 1. A *pseudo-code* description of the algorithm (written in English)
  - 2. A table of each line's contents, to help see the pattern in the input



Pseudo-code

- **pseudo-code**: A written English description of an algorithm to solve a programming problem.

- Example: Suppose we are trying to draw a box of stars on the screen which is 12 characters wide and 7 tall.
  - A possible pseudo-code for this algorithm:

```
print 12 stars.
for (each of 5 lines) {
    print a star.
    print 10 spaces.
    print a star.
}
print 12 stars.
```

```
************
*          *
*          *
*          *
*          *
*          *
************
```

A pseudo-code algorithm

- A possible pseudo-code for our complex figure task:
  1. *Draw top line with # , 16 =, then #*
  2. *Draw the top half with the following on each line:*

     |

     spaces (decreasing in number as we go downward)

     <>

     dots (decreasing in number as we go downward)

     <>

     spaces (same number as above)

     |
  3. *Draw the bottom half, which is the same as the top half but upside-down*
  4. *Draw bottom line with # , 16 =, then #*

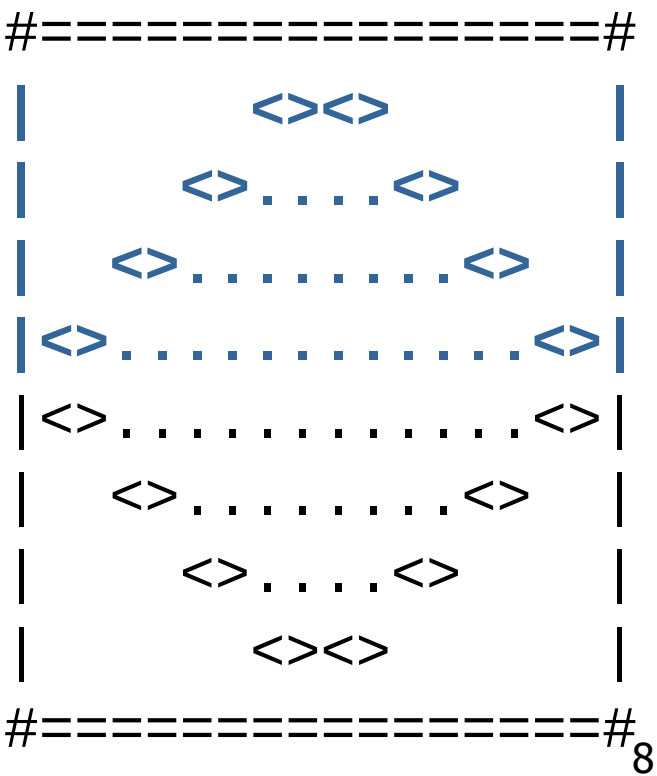
- Our pseudo-code suggests we should use a table to learn the pattern in the top and bottom halves of the figure.

```
#================#
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>..........<>|
|<>..........<>|
|  <>........<>  |
|    <>....<>    |
|      <><>      |
#================#
```

Tables to examine output

- A table of the contents of the lines in the "top half" of the figure:
  - What expressions connect each line with its number of spaces and dots?

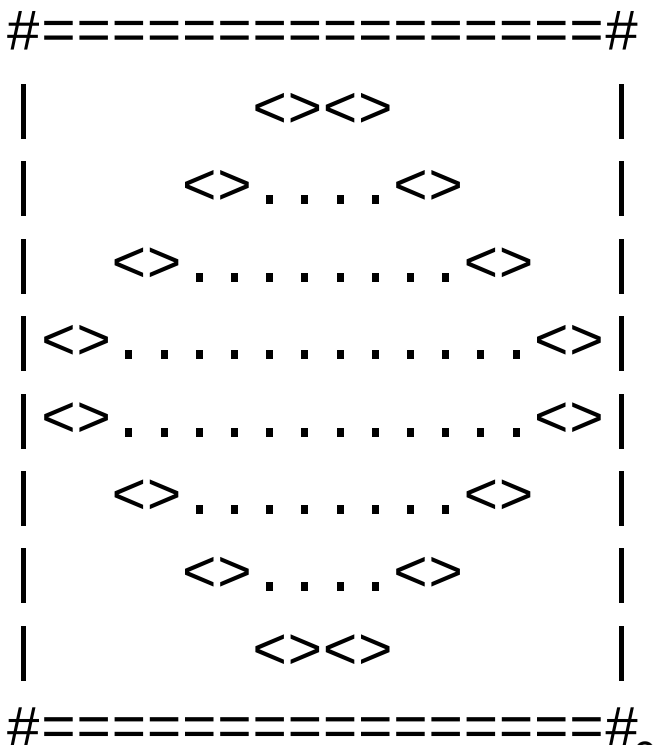
line	spaces	line * -2 + 8	dots	4 * line - 4
1	6	6	0	0
2	4	4	4	4
3	2	2	8	8
4	0	0	12	12



Implementing the figure

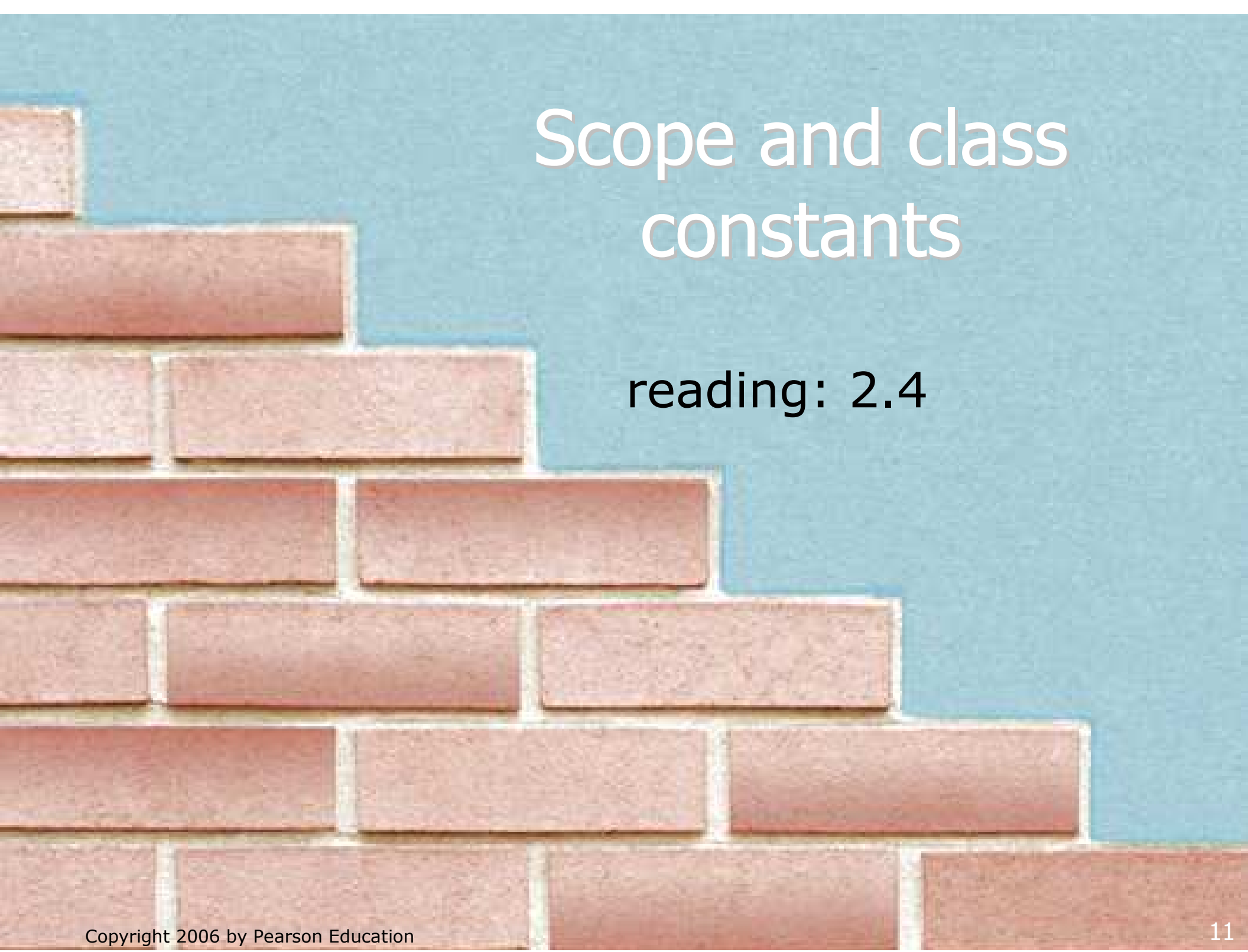
- Let's implement the code for this figure together.

- Some questions we should ask ourselves:
  - How many loops do we need on each line of the top half of the output?
  - Which loops are nested inside which other loops?
  - How should we use static methods to represent the structure and redundancy of the output?



Partial solution

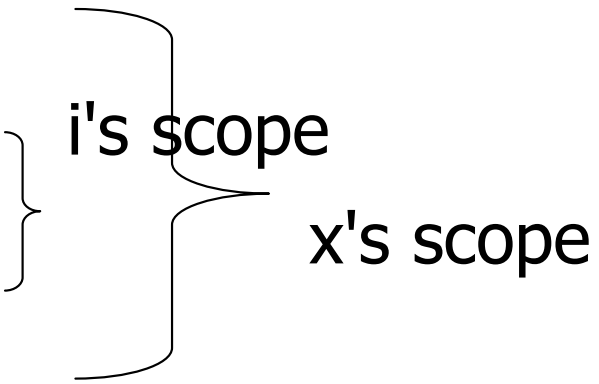
```
// Prints the expanding pattern of <> for the top half of the figure.
public static void drawTopHalf() {
    for (int line = 1; line <= 4; line++) {
        System.out.print("|");

        for (int space = 1; space <= (line * -2 + 8); space++) {
            System.out.print(" ");
        }

        System.out.print("<>");

        for (int dot = 1; dot <= (line * 4 - 4); dot++) {
            System.out.print(".");
        }

        System.out.print("<>");

        for (int space = 1; space <= (line * -2 + 8); space++) {
            System.out.print(" ");
        }

        System.out.println("|");
    }
}
```



Variable scope

- **scope**: The part of a program where a variable exists.
  - A variable's scope is from its declaration to the end of the `{ }` braces in which it was declared.
  - If a variable is declared in a `for` loop, it exists only in that loop.
  - If a variable is declared in a method, it exists in that method.

```
public static void example() {
    int x = 3;
    for (int i = 1; i <= 10; i++) {
        System.out.println(x);
    }
    // i no longer exists here
} // x ceases to exist here
```



Scope and using variables

- It is illegal to use a variable outside of its scope.

```
public static void main(String[] args) {
    example();
    System.out.println(x);  // illegal

    for (int i = 1; i <= 10; i++) {
        int y = 5;
        System.out.println(y);
    }
    System.out.println(y);  // illegal
}

public static void example() {
    int x = 3;
    System.out.println(x);
}
```

Overlapping scope

- It is legal to declare variables with the same name, as long as their scopes do not overlap:

```
public static void main(String[] args) {
    int x = 2;

    for (int i = 1; i <= 5; i++) {
        int y = 5;
        System.out.println(y);
    }
    for (int i = 3; i <= 5; i++) {
        int y = 2;
        int x = 4;  // illegal
        System.out.println(y);
    }
}

public static void anotherMethod() {
    int i = 6;
    int y = 3;
    System.out.println(i + ", " + y);
}
```

Problem: redundant values

- **magic number**: A value used throughout the program.
  - Magic numbers are bad; what if we have to change them?
  - A normal variable cannot be used to fix the magic number problem, because its scope is not large enough.

```
public static void main(String[] args) {
    int max = 3;
    printTop();
    printBottom();
}

public static void printTop() {
    for (int i = 1; i <= max; i++) {        // ERROR: max not found
        for (int j = 1; j <= i; j++) {
            System.out.print(j);
        }
        System.out.println();
    }
}

public static void printBottom() {
    for (int i = max; i >= 1; i--) {        // ERROR: max not found
        for (int j = i; j >= 1; j--) {
            System.out.print(max);          // ERROR: max not found
        }
        System.out.println();
    }
}
```

Class constants

- **class constant**: A named value that can be seen throughout the program.
  - The value of a constant can only be set when it is declared.
  - It can not be changed while the program is running.

- Class constant syntax:

```
public static final <type> <name> = <value> ;
```

  - Constants' names are usually written in ALL_UPPER_CASE.
  - Examples:

```
public static final int DAYS_IN_WEEK = 7;
public static final double INTEREST_RATE = 3.5;
public static final int SSN = 658234569;
```

Class constant example

- Making the 3 a class constant removes the redundancy:

```
public static final int MAX_VALUE = 3;

public static void main(String[] args) {
    printTop();
    printBottom();
}

public static void printTop() {
    for (int i = 1; i <= MAX_VALUE; i++) {
        for (int j = 1; j <= i; j++) {
            System.out.print(j);
        }
        System.out.println();
    }
}

public static void printBottom() {
    for (int i = MAX_VALUE; i >= 1; i--) {
        for (int j = i; j >= 1; j--) {
            System.out.print(MAX_VALUE);
        }
        System.out.println();
    }
}
```

Constants and figures

- Consider the task of drawing the following figures:

```
+/\/\/\/\/\+
|           |
+/\/\/\/\/\+

+/\/\/\/\/\+
|           |
|           |
|           |
|           |
|           |
|           |
+/\/\/\/\/\+
```

  - Each figure is strongly tied to the number 5 (or a multiple of 5, such as 10 ...)
  - Use a class constant so that these figures will be resizable.

Repetitive figure code

- Note the repetition of numbers based on 5 in the code:

```
public static void drawFigure1() {
    drawPlusLine();
    drawBarLine();
    drawPlusLine();
}

public static void drawPlusLine() {
    System.out.print("+");
    for (int i = 1; i <= 5; i++) {
        System.out.print("/\\");
    }
    System.out.println("+");
}

public static void drawBarLine() {
    System.out.print("|");
    for (int i = 1; i <= 10; i++) {
        System.out.print(" ");
    }
    System.out.println("|");
}
```

Output:

```
+/\/\/\/\/\+
|           |
+/\/\/\/\/\+
```

  - It would be cumbersome to resize the figure.

Fixing our code with constant

- A class constant will fix the "magic number" problem:

```
public static final int FIGURE_WIDTH = 5;

public static void drawFigure1() {
    drawPlusLine();
    drawBarLine();
    drawPlusLine();
}

public static void drawPlusLine() {
    System.out.print("+");
    for (int i = 1; i <= FIGURE_WIDTH; i++) {
        System.out.print("/\\");
    }
    System.out.println("+");
}

public static void drawBarLine() {
    System.out.print("|");
    for (int i = 1; i <= 2 * FIGURE_WIDTH; i++) {
        System.out.print(" ");
    }
    System.out.println("|");
}
```

Output:

```
+/\/\/\/\/\+
|           |
+/\/\/\/\/\+
```

Complex figure w/ constant

- Modify the code from the previous slides to use a constant so that it can show figures of different sizes.
  - The figure originally shown has a size of 4.

```
#==================#
|        <><>       |
|     <>....<>      |
|   <>........<>    |
|<>............<>  |
|<>............<>  |
|   <>........<>    |
|     <>....<>      |
|        <><>       |
#==================#
```

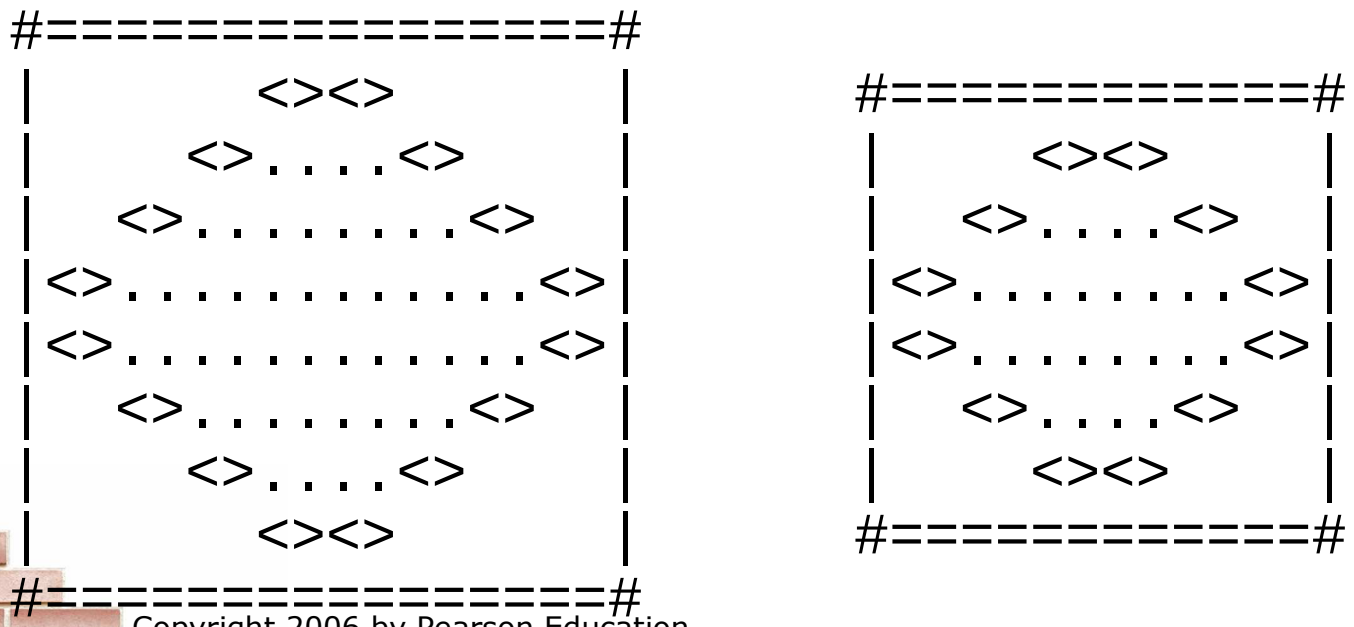
A figure of size 3:

```
#=============#
|      <><>     |
|    <>....<>   |
|  <>........<> |
|  <>........<> |
|    <>....<>   |
|      <><>     |
#=============#
```

Loop tables and constant

- Let's modify our loop table to take into account SIZE
  - Adding the constant sometimes changes the *b* in $y = mx + b$

SIZE	line	spaces	`-2*line + (2*SIZE)`	dots	`4*line - 4`
4	1,2,3,4	6,4,2,0	-2*line + **8**	0,4,8,12	4*line - 4
3	1,2,3	4,2,0	-2*line + **6**	0,4,8	4*line - 4



Partial solution

```
public static final int SIZE = 4;

// Prints the expanding pattern of <> for the top half of the figure.
public static void drawTopHalf() {
    for (int line = 1; line <= SIZE; line++) {
        System.out.print("|");

        for (int space = 1; space <= (line * -2 + (2 * SIZE)); space++) {
            System.out.print(" ");
        }

        System.out.print("<>");

        for (int dot = 1; dot <= (line * 4 - 4); dot++) {
            System.out.print(".");
        }

        System.out.print("<>");

        for (int space = 1; space <= (line * -2 + (2 * SIZE)); space++) {
            System.out.print(" ");
        }

        System.out.println("|");
    }
}
```

Observations about constant

- Adding a constant often changes the amount added (the intercept) in a loop expression.
  - Usually the multiplier (slope) is unchanged.

```
public static final int SIZE = 4;

for (int space = 1; space <= (line * -2 + (2 * SIZE)); space++) {
    System.out.print(" ");
}
```

- The constant doesn't replace *every* occurrence of the original value.

```
for (int dot = 1; dot <= (line * 4 - 4); dot++) {
    System.out.print(".");
}
```

Another complex figure

- Write a program that produces the following output.
  - Write nested `for` loops to capture the repetition.
  - Use static methods to capture structure and redundancy.

```
====+====
#   |   #
#   |   #
#   |   #
====+====
#   |   #
#   |   #
#   |   #
====+====
```

- After implementing the program, add a constant so that the figure can be resized.