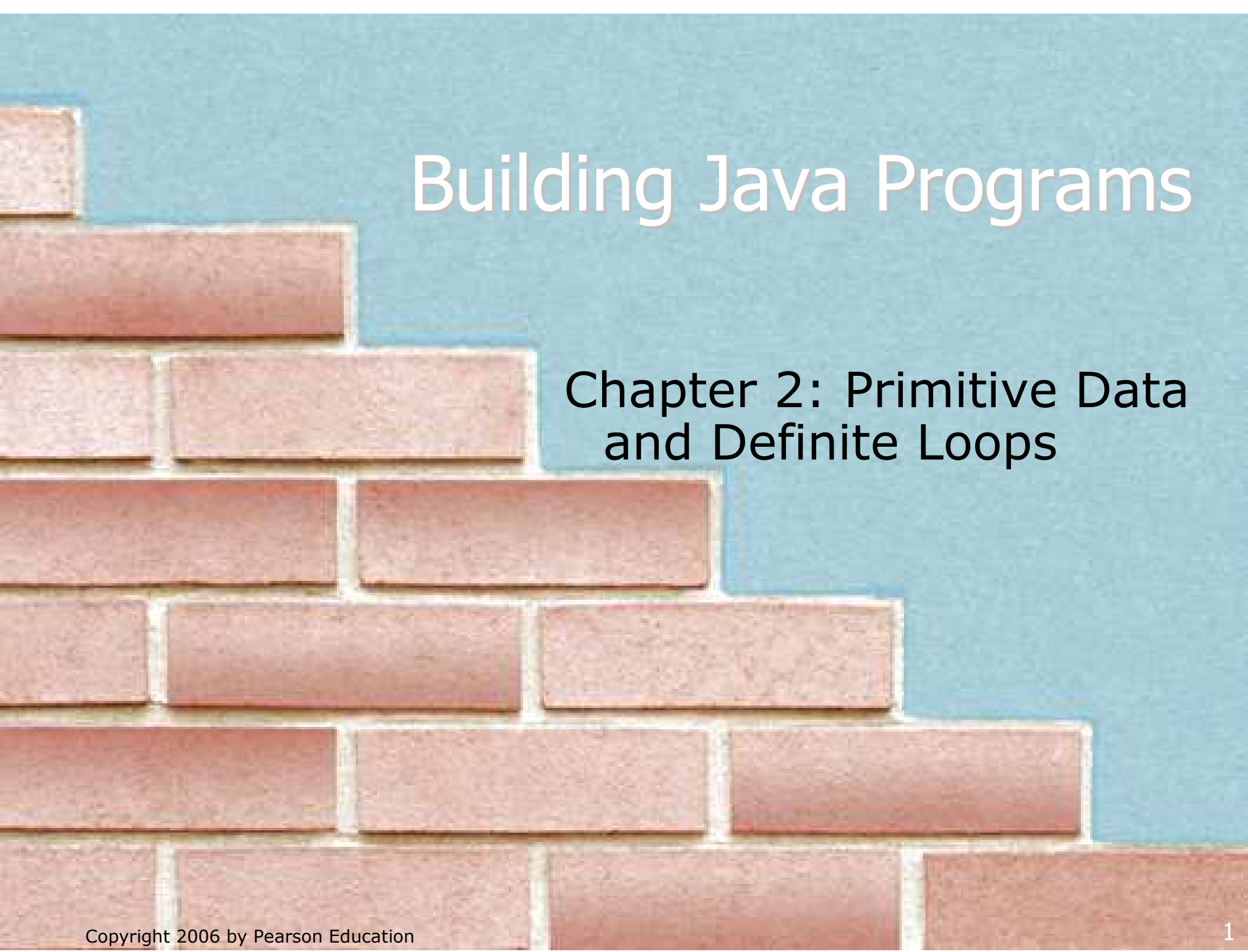
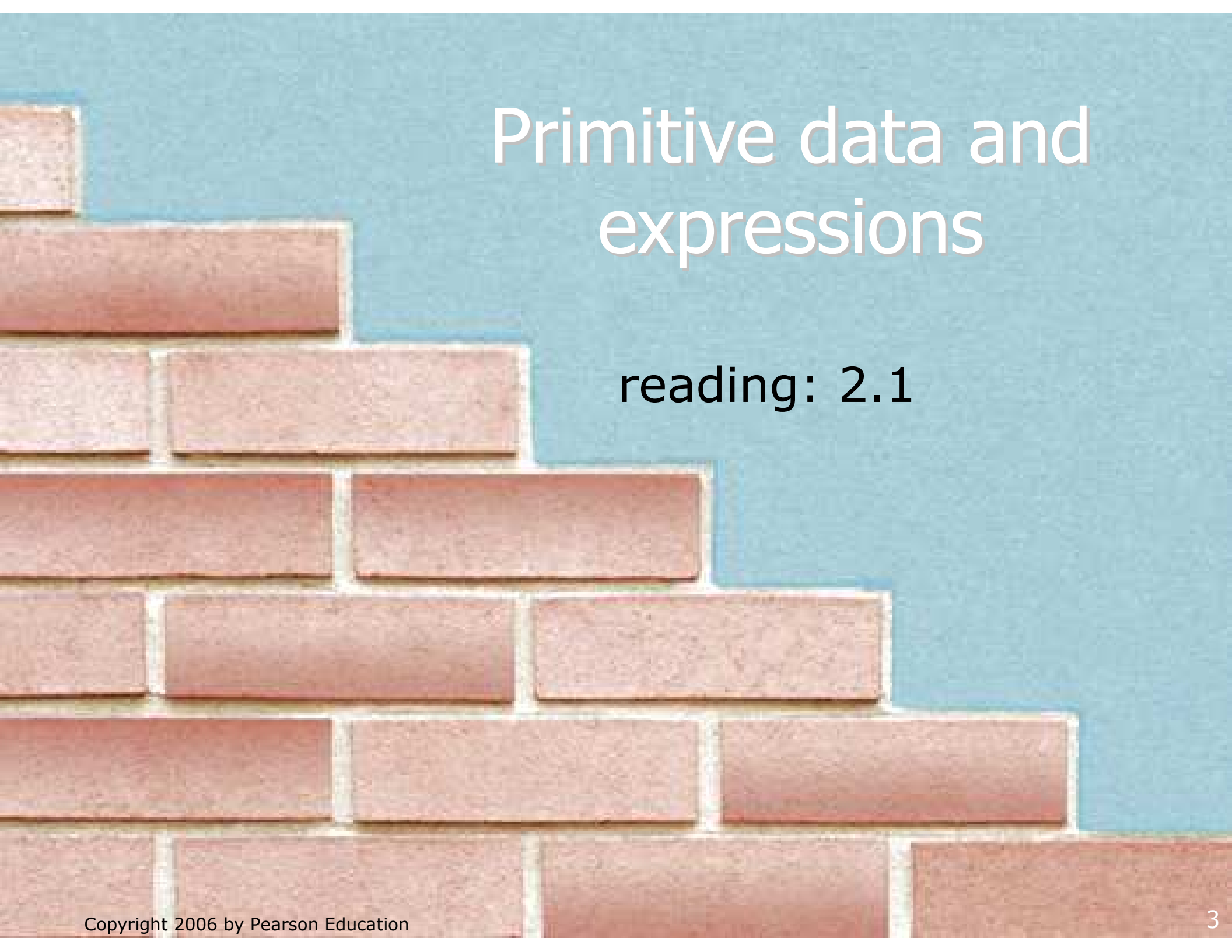


Chapter outline

- data concepts
  - primitive types, expressions, and precedence
  - variables: declaration, initialization, assignment
  - mixing types: casting, string concatenation
  - modify-and-reassign operators
  - `System.out.print`
- repetition
  - the `for` loop
  - nested loops
- managing complexity
  - variable scope
  - class constants
- drawing complex figures



Primitive data and expressions

reading: 2.1

Programs that examine data

- We have printed text with `println` and strings:

```
System.out.println("Hello, world!");
```

- Now we will learn how to print and manipulate other kinds of data, such as numbers:

```
                                        // OUTPUT:
System.out.println(42);                 // 42
System.out.println(3 + 5 * 7);          // 38
System.out.println(12.5 / 8.0);         // 1.5625
```

Data types

- **type**: A category or set of data values.
  - Many languages have a notion of data *types* and ask the programmer to specify what type of data is being manipulated.
  - Examples: integer, real number, string.

- Internally, the computer stores all data as 0s and 1s.
  - examples: `42` ⟶ 101010
    `"hi"` ⟶ 0110100001101001

Java's primitive types

- **primitive types**: Java's built-in simple data types for numbers, text characters, and logic.
  - Java has eight primitive types.
  - Types that are not primitive are called *object* types. (seen later)

- Four primitive types we will use:

Name	Description	Examples
`int`	integers (whole numbers)	`42, -3, 0, 926394`
`double`	real numbers	`3.14, -0.25, 9.4e3`
`char`	single text characters	`'a', 'X', '?', '\n'`
`boolean`	logical values	`true, false`

Expressions

- **expression**: A data value, or a set of operations that compute a data value.

  Example: `1 + 4 * 3`

  - The simplest expression is a *literal value*.
  - A complex expression can use *operators* and parentheses.
    - The values to which an operator applies are called *operands*.

- Five arithmetic operators we will use:

`+`	addition
`-`	subtraction or negation
`*`	multiplication
`/`	division
`%`	modulus, a.k.a. remainder

Evaluating expressions

- As your Java program executes:
  - When a line with an expression is reached, the expression is *evaluated* (its value is computed).
    - `1 + 1` is evaluated to `2`
  - `System.out.println(3 * 4);` prints `12`
    (How would we print the text `3 * 4` ?)
- When an expression contains more than one operator of the same kind, it is evaluated left-to-right.
  - `1 + 2 + 3` is `(1 + 2) + 3` which is `6`
  - `1 - 2 - 3` is `(1 - 2) - 3` which is `-4`

Integer division with /

- When we divide integers, the quotient is also an integer.
  - `14 / 4` is `3`, not `3.5`

```
     3           4           52
4 ) 14     10 ) 45     27 ) 1425
    12          40          135
    --          --          ---
     2           5           75
                             54
                             --
                             21
```

  - More examples:
    - `1425 / 27` is `52`
    - `35 / 5` is `7`
    - `84 / 10` is `8`
    - `156 / 100` is `1`

  - Dividing by 0 causes an error when your program runs.

Integer remainder with %

- The % operator computes the remainder from a division of two integers.
  - `14 % 4` is `2`
  - `218 % 5` is `3`

```
    3
4 ) 14
    12
    2
```

```
    43
5 ) 218
    20
    18
    15
    3
```

- What are the results of the following expressions?

```
45 % 6
2 % 2
8 % 20
11 % 0
```

Applications of % operator

- Obtains the last digit (units place) of a number:
  - Example: From 230857, obtain the 7.
- Obtain the last 4 digits of a Social Security Number:
  - Example: From 658236489, obtain 6489.
- Obtains a number's second-to-last digit (tens place):
  - Example: From 7342, obtain the 4.
- Use the % operator to see whether a number is odd:
  - Can it help us determine whether a number is divisible by 3?

Operator precedence

- **precedence**: Order in which operations are computed.
  - `*  /  %` have a higher level of precedence than `+  -`

    `1 + 3 * 4` is 13

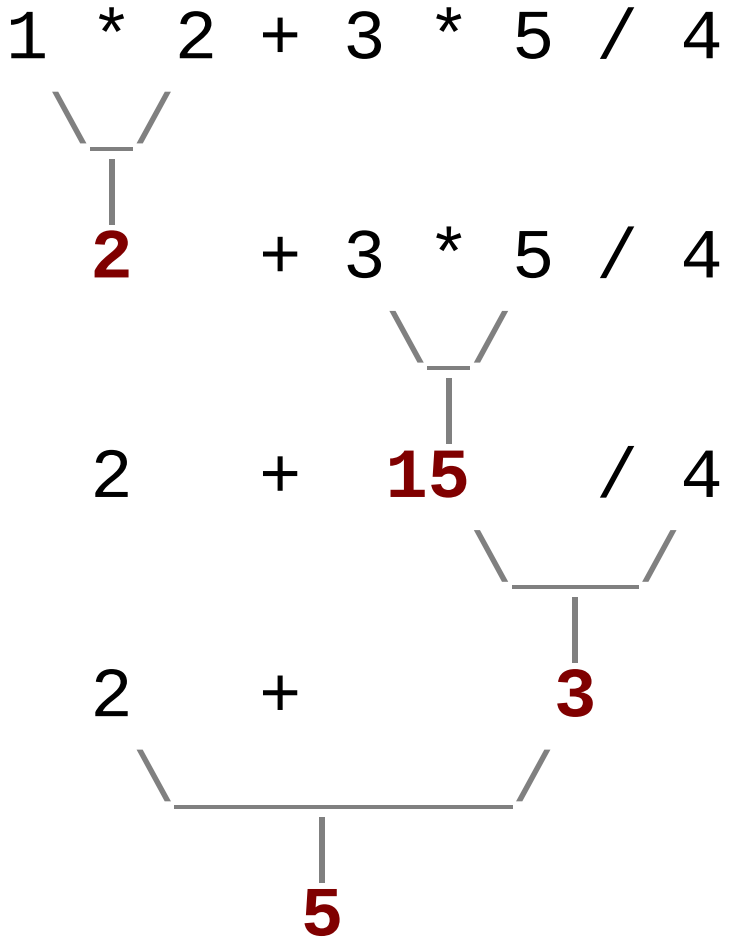
  - Parentheses can be used to force a certain order of evaluation.

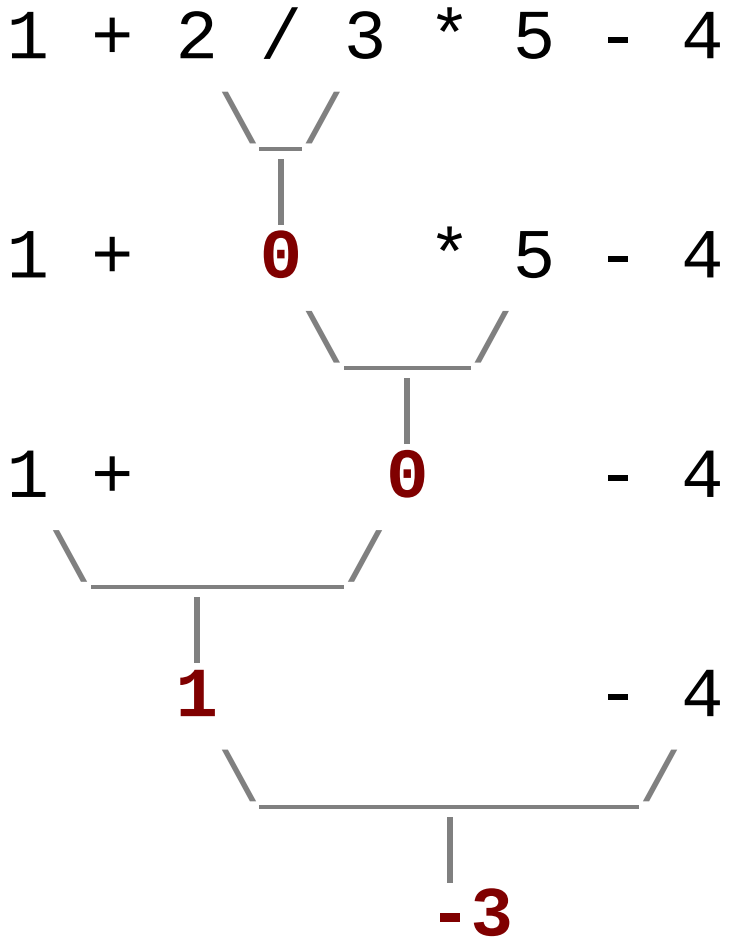
    `(1 + 3) * 4` is 16

  - Spacing does not affect order of evaluation.

    `1+3 * 4-2` is 11

Precedence examples





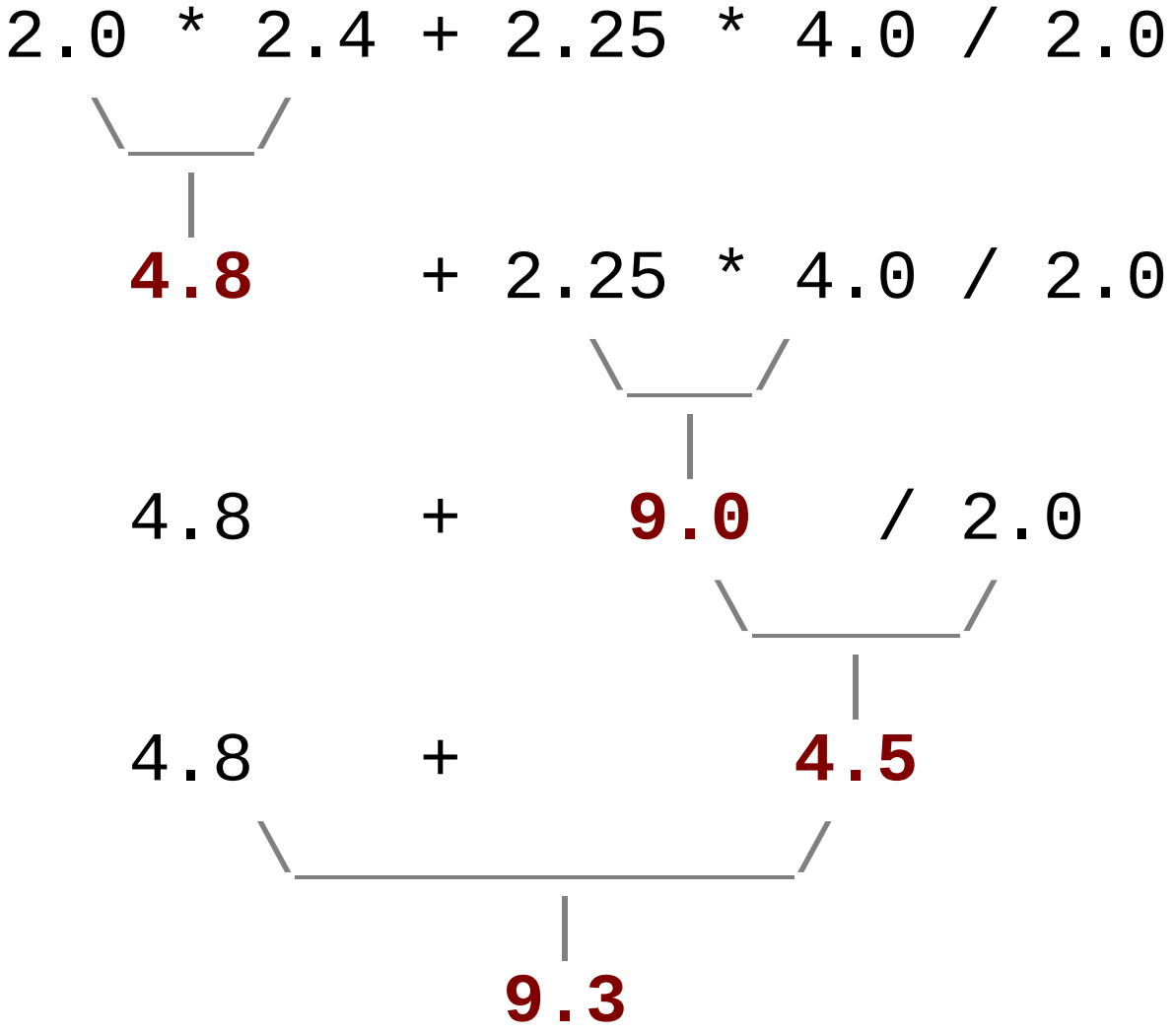
Precedence questions

- What values result from the following expressions?
  - `9 / 5`
  - `695 % 20`
  - `7 + 6 * 5`
  - `7 * 6 + 5`
  - `248 % 100 / 5`
  - `6 * 3 - 9 / 4`
  - `(5 - 7) * 4`
  - `6 + (18 % (17 - 12))`

Real numbers (`double`)

- Java can also manipulate real numbers (type `double`).
  - Examples: `6.022` `-15.9997` `42.0` `2.143e17`
- The operators `+ - * / % ( )` all work for real numbers.
  - The `/` produces an exact answer when used on real numbers. `15.0 / 2.0` is `7.5`
- The same rules of precedence that apply to integers also apply to real numbers.
  - Evaluate `( )` before `* / %` before `+ -`

Real number example



Real number precision

- The computer internally represents real numbers in an imprecise way.

- Example:

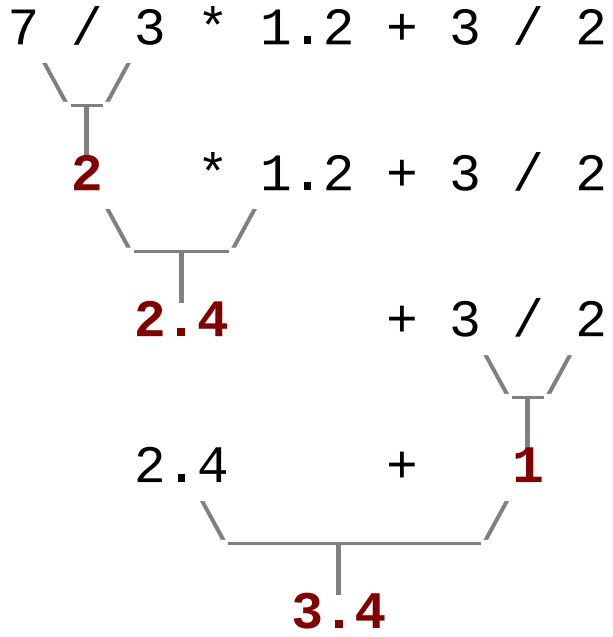
```
System.out.println(0.1 + 0.2);
```

  - The mathematically correct answer should be 0.3
  - Instead, the output is 0.30000000000000004

- Later we will learn some ways to produce a better output for examples like the above.

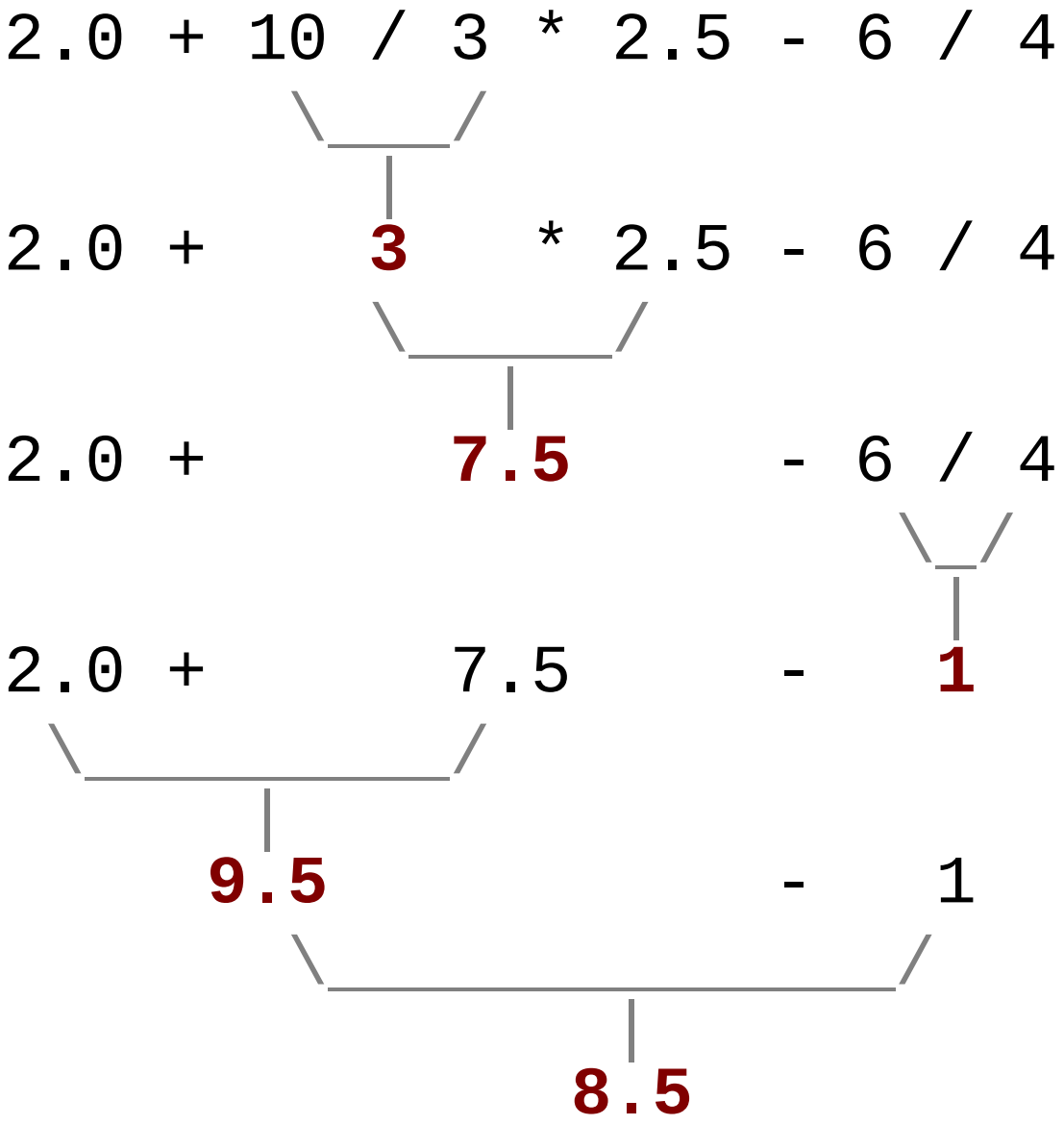
Mixing integers and reals

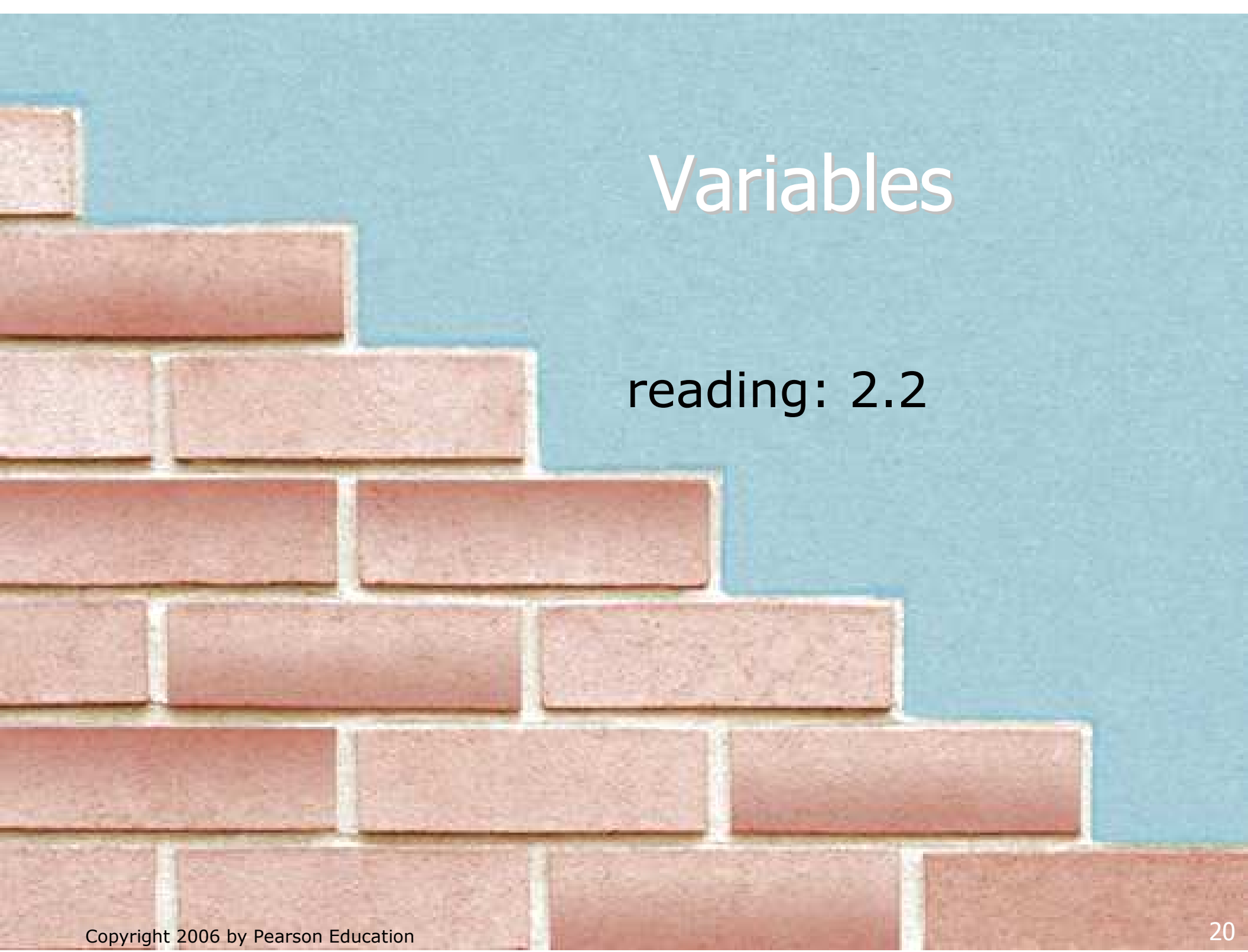
- When a Java operator is used on an integer and a real number, the result is a real number.
  - `4.2 * 3` is `12.6`
  - `1 / 2.0` is `0.5`
- The conversion occurs on a per-operator basis. It affects only its two operands.



  - Notice how `3 / 2` is still `1` above, not `1.5`.

Mixed types example





The computer's memory

- Expressions are like using the computer as a calculator.

- Calculators have memory keys to store/retrieve values.
  - When is this useful?

  - We'd like the ability to save and restore values in our Java programs, like the memory keys on the calculator.

Variables

- **variable**: A piece of your computer's memory that is given a name and type and can store a value.
  - Usage:
    - compute an expression's result,
    - store that result into a variable,
    - and use that variable later in the program.
  - Unlike with a calculator, we can declare as many variables as we want.
- Variables are a bit like preset stations on a car stereo.

Declaring variables

- **variable declaration statement**: A Java statement that creates a new variable of a given type.
  - A variable is *declared* in a statement with its type and name.
  - Variables must be declared before they can be used.

- Declaration syntax:

  <type> <name> ;

  - `int x;`
  - `double myGPA;`

  - The name can be any identifier.

More on declaring variables

- Declaring a variable sets aside a piece of memory in which you can store a value.
  - `int x;`
  - `int y;`

  - Part of the computer's memory:

(The memory has no values in it yet.)

Assignment statements

- **assignment statement**: A statement that stores a value into a variable's memory.
  - Variables must be declared before they can be assigned a value.

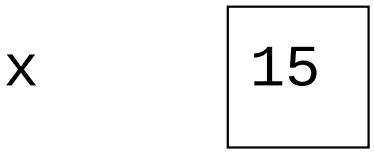
- Assignment statement syntax:

```
<name> = <value> ;
```

  - `x = 3;`
  - `myGPA = 3.25;`

More about assignment

- The ***<value>*** assigned can be a complex expression.
  - The expression is evaluated; the variable stores the result.
  - `x = (2 + 8) / 3 * 5;`



- A variable can be assigned a value more than once.
  - Example:

```
int x;
x = 3;
System.out.println(x);   // 3

x = 4 + 7;
System.out.println(x);   // 11
```

Using variables' values

- Once a variable has been assigned a value, it can be used in an expression, just like a literal value.

```
int x;
x = 3;
System.out.println(x * 5 - 1);
```

  - The above has output equivalent to:

```
System.out.println(3 * 5 - 1);
```

Assignment and algebra

- Though the assignment statement uses the = character, it is not an algebraic equation.
  - = means, "store the value on the right in the variable on the left"
  - Some people read `x = 3;` as, "x becomes 3" or, "x gets 3"
  - We would not say `3 = 1 + 2;` because 3 is not a variable.

- What happens when a variable is used on both sides of an assignment statement?

```
int x;
x = 3;
x = x + 2;   // what happens?
```

  - The above wouldn't make any sense in algebra...

Some errors

- A compiler error will result if you declare a variable twice, or declare two variables with the same name.
  - ```
    int x;
    int x;                          // ERROR: x already exists
    ```

- A variable that has not been assigned a value cannot be used in an expression or `println` statement.
  - ```
    int x;
    System.out.println(x);   // ERROR: x has no value
    ```

Assignment and types

- A variable can only store a value of its own type.
  - ```
    int x;
    x = 2.5;   // ERROR: x can only store int
    ```
- An `int` value can be stored in a `double` variable.
  - The value is converted into the equivalent real number.
  - ```
    double myGPA;
    myGPA = 2;
    ```

Assignment examples

- What is the output of the following Java code?

```
int number;
number = 2 + 3 * 4;
System.out.println(number - 1);

number = 16 % 6;
System.out.println(2 * number);
```

- What is the output of the following Java code?

```
double average;
average = (11 + 8) / 2;
System.out.println(average);

average = (5 + average * 2) / 2;
System.out.println(average);
```

Declaration/initialization

- A variable can be declared and assigned an initial value in the same statement.

- Declaration/initialization statement syntax:

  <type> ***<name>*** = ***<value>*** ;

  - `double myGPA = 3.95;`
  - `int x = (11 % 3) + 12;`

same effect as:

```
double myGPA;
myGPA = 3.95;

int x;
x = (11 % 3) + 12;
```

Multiple declaration error

- The compiler will fail if you try to declare-and-initialize a variable twice.
  - ```
    int x = 3;
    System.out.println(x);

    int x = 5;      // ERROR: variable x already exists
    System.out.println(x);
    ```
  - This is the same as trying to declare x twice.

- How can the code be fixed?
    ```

Multiple declarations per line

- It is legal to declare multiple variables on one line:

  <type> ***<name>***, ***<name>***, ..., ***<name>*** ;

  - `int a, b, c;`
  - `double x, y;`

- It is legal to declare/initialize several at once:

  <type> ***<name>*** = ***<value>*** , ..., ***<name>*** = ***<value>*** ;

  - `int a = 2, b = 3, c = -4;`
  - `double grade = 3.5, delta = 0.1;`

- The variables must be of the same type.

Integer or real number?

- Categorize each of the following quantities by whether an `int` or `double` variable would best to store it:

integer (`int`)	real number (`double`)

1. Temperature in degrees Celsius
2. The population of lemmings
3. Your grade point average
4. A person's age in years
5. A person's weight in pounds
6. A person's height in meters
7. Number of miles traveled
8. Number of dry days in the past month
9. Your locker number
10. Number of seconds left in a game
11. The sum of a group of integers
12. The average of a group of integers

Type casting

- **type cast**: A conversion from one type to another.

  Common uses:
  - To promote an `int` into a `double` to achieve exact division.
  - To truncate a `double` from a real number to an integer.

- type cast syntax:

  (***<type>***) ***<expression>***

  Examples:
  - `double result = (double) 19 / 5;  // 3.8`
  - `int result2 = (int) result;  // 3`

More about type casting

- Type casting has high precedence and only casts the item immediately next to it.
  - `double x = (double) 1 + 1 / 2;  // 1`
  - `double y = 1 + (double) 1 / 2;  // 1.5`
- You can use parentheses to force evaluation order.
  - `double average = (double) (a + b + c) / 3;`
- A conversion to `double` can be achieved in other ways.
  - `double average = 1.0 * (a + b + c) / 3;`

String concatenation

- **string concatenation**: Using the + operator between a String and another value to make a longer String.
  - Examples:
    - Recall: Precedence of + operator is below * / %

```
"hello" + 42     is "hello42"
1 + "abc" + 2    is "1abc2"
"abc" + 1 + 2    is "abc12"
1 + 2 + "abc"    is "3abc"
"abc" + 9 * 3    is "abc27"
"1" + 1          is "11"
4 - 1 + "abc"    is "3abc"

"abc" + 4 - 1    causes a compiler error... why?
```

Printing String expressions

- String expressions with + are useful so that we can print complicated messages that involve computed values.
  - ```
    double grade = (95.1 + 71.9 + 82.6) / 3.0;
    System.out.println("Your grade was " + grade);

    int students = 11 + 17 + 4 + 19 + 14;
    System.out.println("There are " + students +
                       " students in the course.");
    ```

  Output:

  ```
  Your grade was 83.2
  There are 65 students in the course.
  ```
  ```

Example variable exercise

- Write a Java program that stores the following data:
  - Section AA has 17 students.
  - Section AB has 8 students.
  - Section AC has 11 students.
  - Section AD has 23 students.
  - Section AE has 24 students.
  - Section AF has 7 students.
  - The average number of students per section.

and prints the following:

```
There are 24 students in Section AE.
There are an average of 15 students per section.
```

Increment and decrement

- The *increment* and *decrement* operators increase or decrease a variable's value by 1.

Shorthand	Equivalent longer version
<variable> `++ ;`	***<variable>*** `=` ***<variable>*** `+ 1;`
<variable> `-- ;`	***<variable>*** `=` ***<variable>*** `- 1;`

- Examples:

```
int x = 2;
x++;                          // x = x + 1;
                              // x now stores 3

double gpa = 2.5;
gpa--;                        // gpa = gpa - 1;
                              // gpa now stores 1.5
```

Modify-and-assign operators

- Java has several shortcut operators that allow you to quickly modify a variable's value:

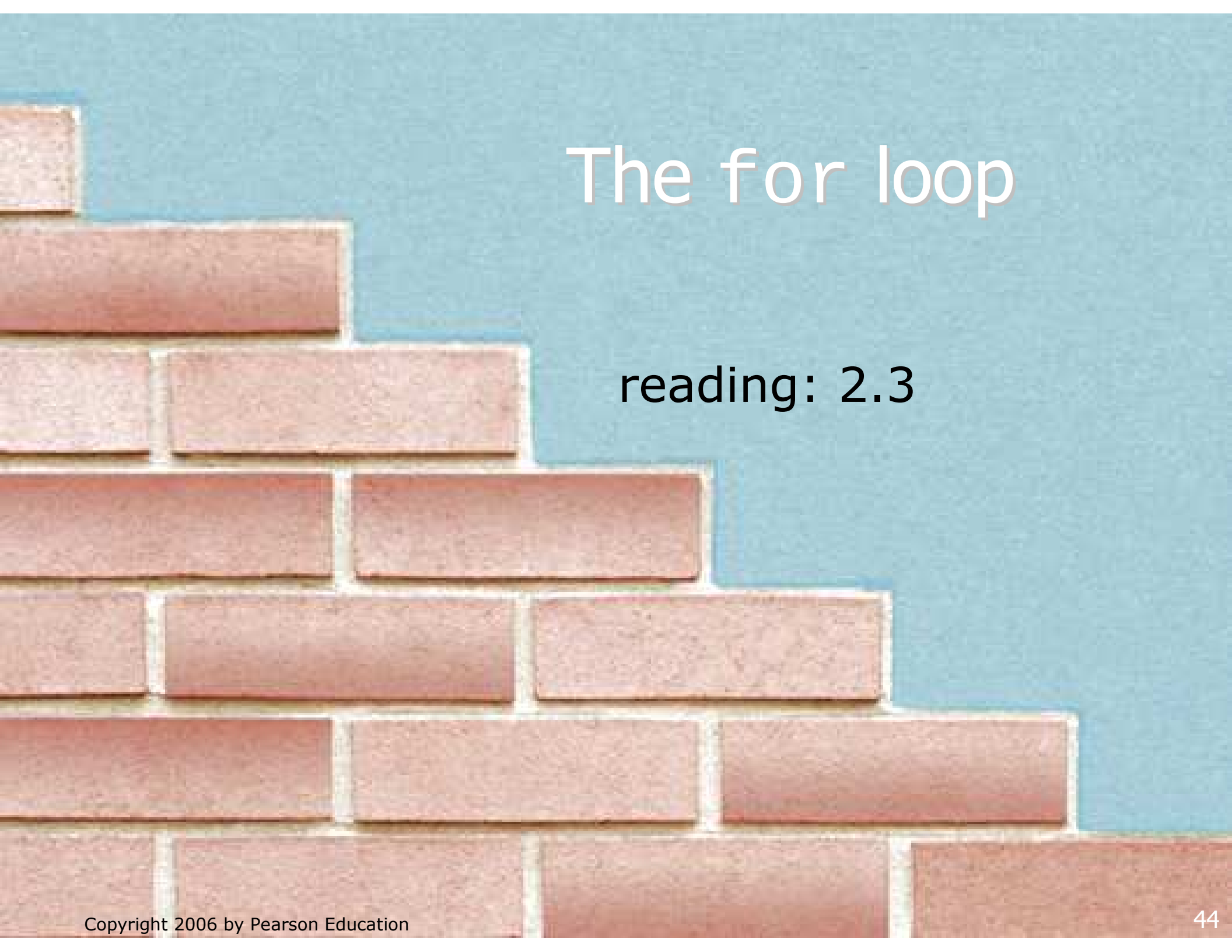
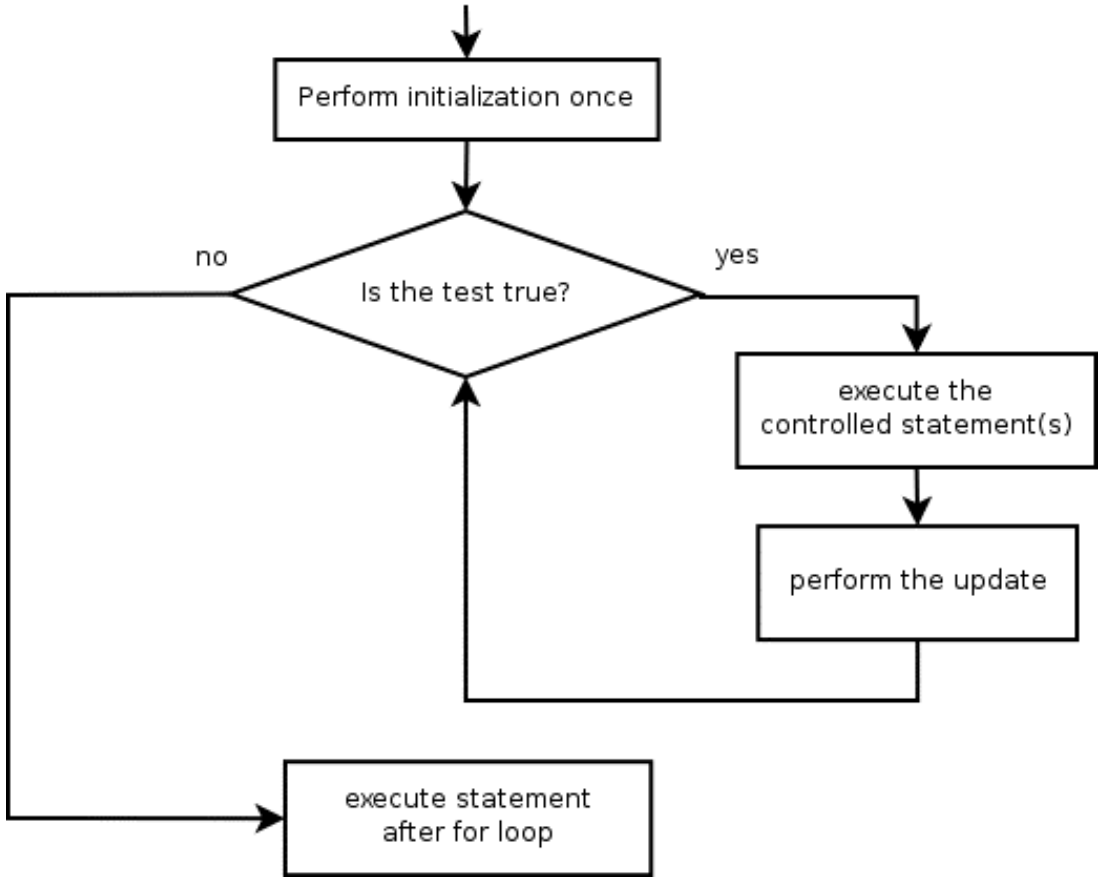
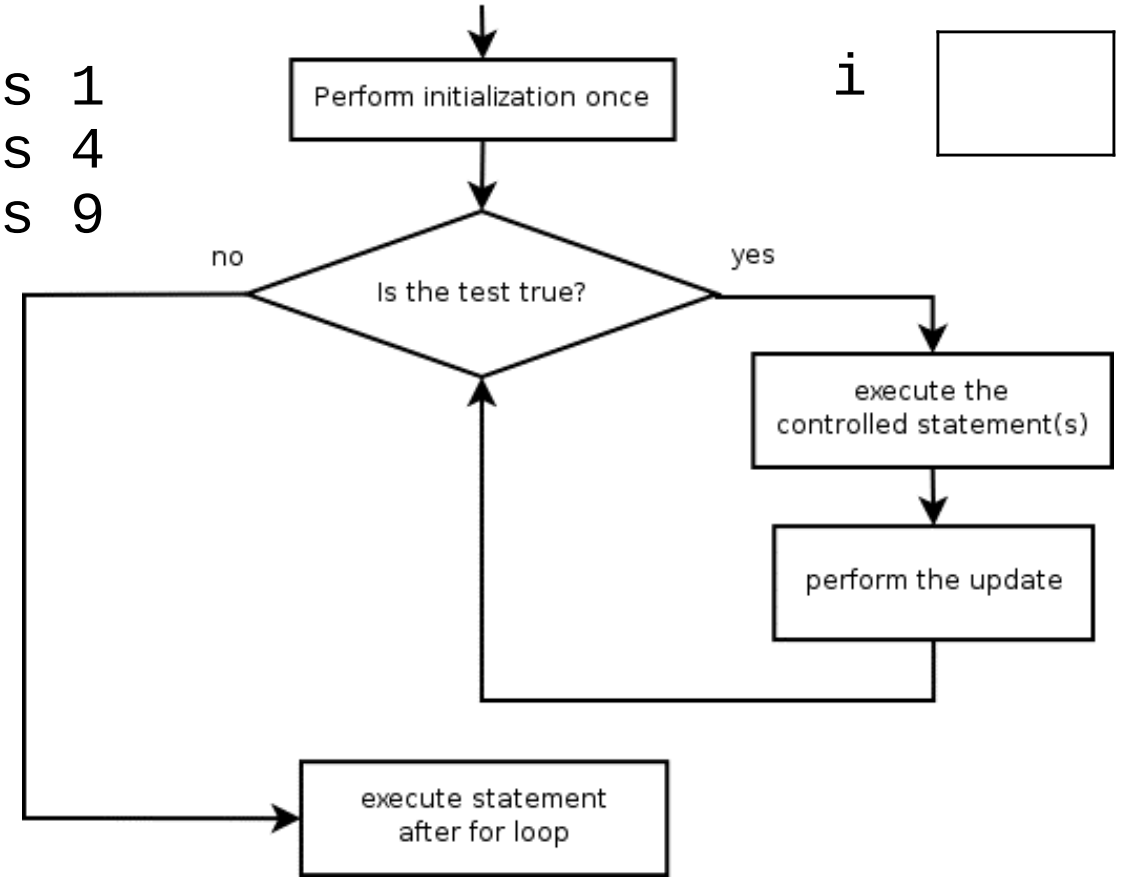
Shorthand	Equivalent longer version
<variable> += ***<value>*** ;	***<variable>*** = ***<variable>*** + ***<value>*** ;
<variable> -= ***<value>*** ;	***<variable>*** = ***<variable>*** - ***<value>*** ;
<variable> *= ***<value>*** ;	***<variable>*** = ***<variable>*** * ***<value>*** ;
<variable> /= ***<value>*** ;	***<variable>*** = ***<variable>*** / ***<value>*** ;
<variable> %= ***<value>*** ;	***<variable>*** = ***<variable>*** % ***<value>*** ;

- Examples:
  - `x += 3;` `// x = x + 3;`
  - `gpa -= 0.5;` `// gpa = gpa - 0.5;`
  - `number *= 2;` `// number = number * 2;`

System.out.print command

- Recall: System.out.println prints a line of output and then advances to a new line.
- System.out.print prints without moving to a new line.
  - This allows you to print partial messages on the same line.
- Example:
  - ```
    System.out.print("Kind of");
    System.out.print("Like a cloud,");
    System.out.println("I was up");
    System.out.print("Way up ");
    System.out.println("in the sky");
    ```

    Output:

    ```
    Kind ofLike a cloud,I was up
    Way up in the sky
    ```
    ```



The `for` loop

reading: 2.3

Repetition with for loops

- So far, when we wanted to perform a task multiple times, we have written redundant code:

```
System.out.println("I am so smart");
System.out.println("I am so smart");
System.out.println("I am so smart");
System.out.println("I am so smart");
System.out.println("I am so smart");
System.out.println("S-M-R-T");
System.out.println("I mean S-M-A-R-T");
```

- Java has a statement called a *for loop statement* that instructs the computer to perform a task many times.

```
for (int i = 1; i <= 5; i++) {   // repeat 5 times
    System.out.println("I am so smart");
}
System.out.println("S-M-R-T");
System.out.println("I mean S-M-A-R-T");
```

for loop syntax

- **for loop**: A Java statement that executes a group of statements repeatedly until a given test fails.
  - General syntax:

```
for (<initialization> ; <test> ; <update>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

header

body

  - Example:

```
for (int i = 1; i <= 10; i++) {
    System.out.println("His name is Robert Paulson");
}
```

for loop over range of ints

- We'll write `for` loops over integers in a given range.
  - The loop declares a *loop counter* variable that is used in the test, update, and body of the loop.

```
for (int <name> = 1; <name> <= <value>; <name>++)
```

  - Example:

```
for (int i = 1; i <= 6; i++) {
    System.out.println(i + " squared is " + (i * i));
}
```

  - Interpretation: "For each integer **i** from 1 through 6, ..."
  - Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
```

for loop flow diagram

- Behavior of the `for` loop:
  - Start out by performing the ***<initialization>*** once.
  - Repeatedly execute the ***<statement(s)>*** followed by the ***<update>*** as long as the ***<test>*** is still a true statement.



Loop walkthrough

Let's walk through the following `for` loop:

```
for (int i = 1; i <= 3; i++) {
    System.out.println(i + " squared is " + (i * i));
}
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
```



Another example for loop

- The body of a `for` loop can contain multiple lines.
  - Example:

```
System.out.println("+----+");
for (int i = 1; i <= 3; i++) {
    System.out.println("\\    /");
    System.out.println("/    \\");
}
System.out.println("+----+");
```

  - Output:

```
+----+
\    /
/    \
\    /
/    \
\    /
/    \
+----+
```

Some for loop variations

- The initial and final values for the loop counter variable can be arbitrary numbers or expressions:
  - Example:

```
for (int i = -3; i <= 2; i++) {
    System.out.println(i);
}
```

  - Output:

```
-3
-2
-1
0
1
2
```

  - Example:

```
for (int i = 1 + 3 * 4; i <= 5248 % 100; i++) {
    System.out.println(i + " squared is " + (i * i));
}
```

Downward-counting for loop

- The update can also be a `--` or other operator, to make the loop count down instead of up.
  - This also requires changing the test to say >= instead of <= .

```
System.out.print("T-minus ");
for (int i = 10; i >= 1; i--) {
     System.out.print(i + ", ");
}
System.out.println("blastoff!");
```

  - Output:

```
T-minus 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!
```

Single-line for loop

- When a `for` loop only has one statement in its body, the `{ }` braces may be omitted.

```
for (int i = 1; i <= 6; i++)
    System.out.println(i + " squared is " + (i * i));
```

- However, this can lead to mistakes where a line appears to be inside a loop, but is not:
  - ```
    for (int i = 1; i <= 3; i++)
        System.out.println("This is printed 3 times");
        System.out.println("So is this... or is it?");
    ```
  - Output:

```
This is printed 3 times
This is printed 3 times
This is printed 3 times
So is this... or is it?
```
```

for loop questions

- Write a loop that produces the following output.

```
On day #1 of Christmas, my true love sent to me
On day #2 of Christmas, my true love sent to me
On day #3 of Christmas, my true love sent to me
On day #4 of Christmas, my true love sent to me
On day #5 of Christmas, my true love sent to me
...
On day #12 of Christmas, my true love sent to me
```

- Write a loop that produces the following output.

```
2 4 6 8
Who do we appreciate
```

Mapping loops to numbers

- Suppose that we have the following loop:

```
for (int count = 1; count <= 5; count++) {
    ...
}
```

  - What statement could we write in the body of the loop that would make the loop print the following output?

```
3 6 9 12 15
```

- Answer:

```
for (int count = 1; count <= 5; count++) {
    System.out.print(3 * count + " ");
}
```

Mapping loops to numbers 2

- Now consider another loop of the same style:

```
for (int count = 1; count <= 5; count++) {
    ...
}
```

  - What statement could we write in the body of the loop that would make the loop print the following output?

```
4 7 10 13 16
```

- Answer:

```
for (int count = 1; count <= 5; count++) {
    System.out.print(3 * count + 1 + " ");
}
```

Loop number tables

- What statement could we write in the body of the loop that would make the loop print the following output?

```
2 7 12 17 22
```

- To find the pattern, it can help to make a table of the count and the number to print.
  - Each time count goes up by 1, the number should go up by 5.
  - But count * 5 is too great by 3, so we must subtract 3.

count	number to print	count * 5	count * 5 - 3
1	2	5	2
2	7	10	7
3	12	15	12
4	17	20	17
5	22	25	22

Loop table question

- What statement could we write in the body of the loop that would make the loop print the following output?

```
17 13 9 5 1
```

- Let's create the loop table together.
  - Each time count goes up 1, the number should ...
  - But this multiple is off by a margin of ...

count	number to print	count * -4	count * -4 + 21
1	17	-4	17
2	13	-8	13
3	9	-12	9
4	5	-16	5
5	1	-20	1

Degenerate loops

- Some loops execute 0 times, because of the nature of their test and update.

```
// a degenerate loop
for (int i = 10; i < 5; i++) {
    System.out.println("How many times do I print?");
}
```

- Some loops execute endlessly (or far too many times), because the loop test never fails.
- A loop that never terminates is called an *infinite loop*.

```
for (int i = 10; i >= 1; i++) {
    System.out.println("Runaway Java program!!!");
}
```

Nested loops

- **nested loop**: Loops placed inside one another.
  - The inner loop's counter variable must have a different name.

```
for (int i = 1; i <= 3; i++) {
    System.out.println("i = " + i);
    for (int j = 1; j <= 2; j++) {
        System.out.println("  j = " + j);
    }
}
```

Output:

```
i = 1
  j = 1
  j = 2
i = 2
  j = 1
  j = 2
i = 3
  j = 1
  j = 2
```

More nested loops

- In this example, all of the statements in the outer loop's body are executed 5 times.
  - The inner loop prints 10 numbers each of those 5 times, for a total of 50 numbers printed.

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print((i * j) + " ");
    }
    System.out.println();  // to end the line
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
```

Nested for loop exercise

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

- Output:

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

Nested for loop exercise

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

- Output:

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

Nested for loop exercise

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print(i);
    }
    System.out.println();
}
```

- Output:

```
1
22
333
4444
55555
666666
```

Nested for loop exercise

- What nested `for` loops produce the following output?

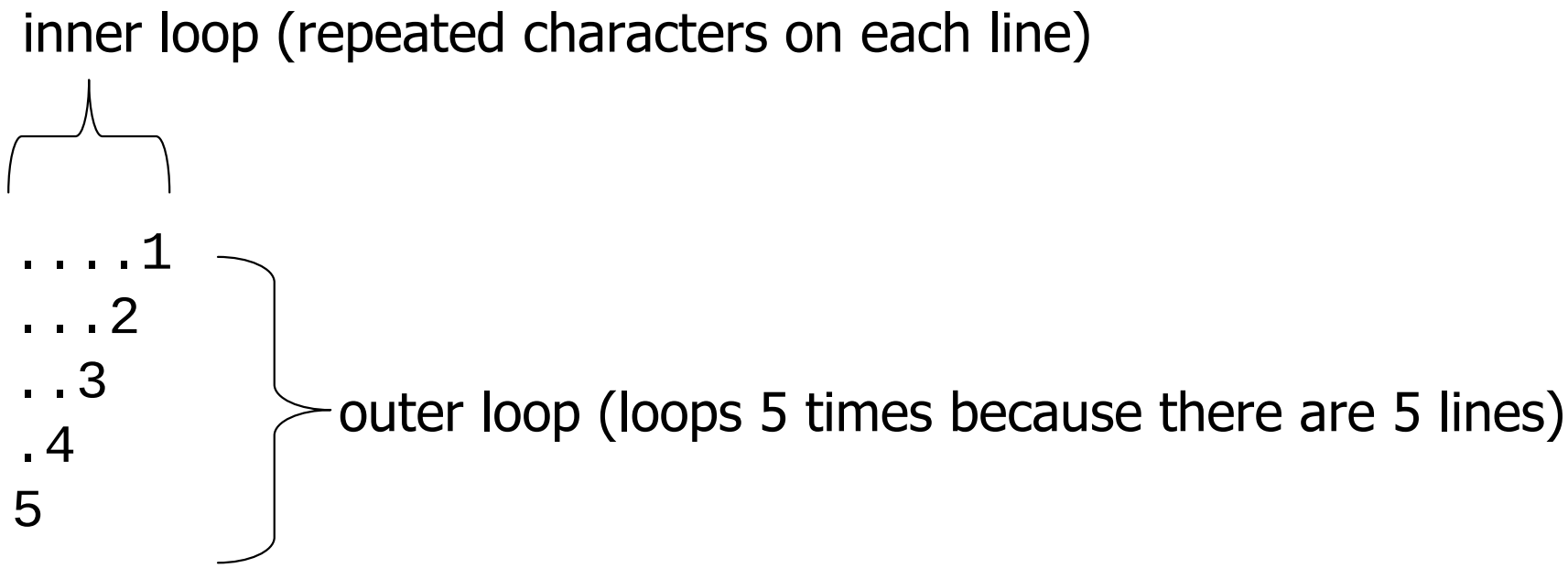
```
1, 1
2, 1
3, 1
1, 2
2, 2
3, 2
```

- Answer:

```
for (int y = 1; y <= 2; y++) {
    for (int x = 1; x <= 3; x++) {
        System.out.println(x + ", " + y);
    }
}
```

Nested for loop exercise

- What nested `for` loops produce the following output?



- This is an example of a nested loop problem where we build multiple complex lines of output:
  - outer "vertical" loop for each of the lines
  - inner "horizontal" loop(s) for the patterns within each line

Nested for loop exercise

- First we write the outer loop, which always goes from 1 to the number of lines desired:

```
for (int line = 1; line <= 5; line++) {
    ...
}
```

- We notice that each line has the following pattern:
  - some number of dots (0 dots on the last line)
  - a number

```
....1
...2
..3
.4
5
```

Nested for loop exercise

- Next we make a table to represent any necessary patterns on that line:

```
....1
...2
..3
.4
5
```

line	# of dots	value displayed	
1	4	1	
2	3	2	
3	2	3	
4	1	4	
5	0	5	

- Answer:

```
for (int line = 1; line <= 5; line++) {
    for (int j = 1; j <= (-1 * line + 5); j++) {
        System.out.print(".");
    }
    System.out.println(line);
}
```

Nested for loop exercise

- A for loop can have more than one loop nested in it.
- What is the output of the following nested for loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= (5 - i); j++) {
        System.out.print(" ");
    }
    for (int k = 1; k <= i; k++) {
        System.out.print(i);
    }
    System.out.println();
}
```

- Answer:

```
    1
   22
  333
 4444
55555
```

Nested for loop exercise

- Modify the previous code to produce this output:

```
....1
...2.
..3..
.4...
5....
```

line	# of dots	value displayed	# of dots
1	4	1	0
2	3	2	1
3	2	3	2
4	1	4	3
5	0	5	4

- Answer:

```
for (int line = 1; line <= 5; line++) {
    for (int j = 1; j <= (-1 * line + 5); j++) {
        System.out.print(".");
    }
    System.out.print(line);
    for (int j = 1; j <= (line - 1); j++) {
        System.out.print(".");
    }
    System.out.println();
}
```

Common nested loop bugs

- It is easy to accidentally type the wrong loop variable.
  - What is the output of the following nested loops?

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; i <= 5; j++) {
        System.out.print(j);
    }
    System.out.println();
}
```

  - What is the output of the following nested loops?

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 5; i++) {
        System.out.print(j);
    }
    System.out.println();
}
```

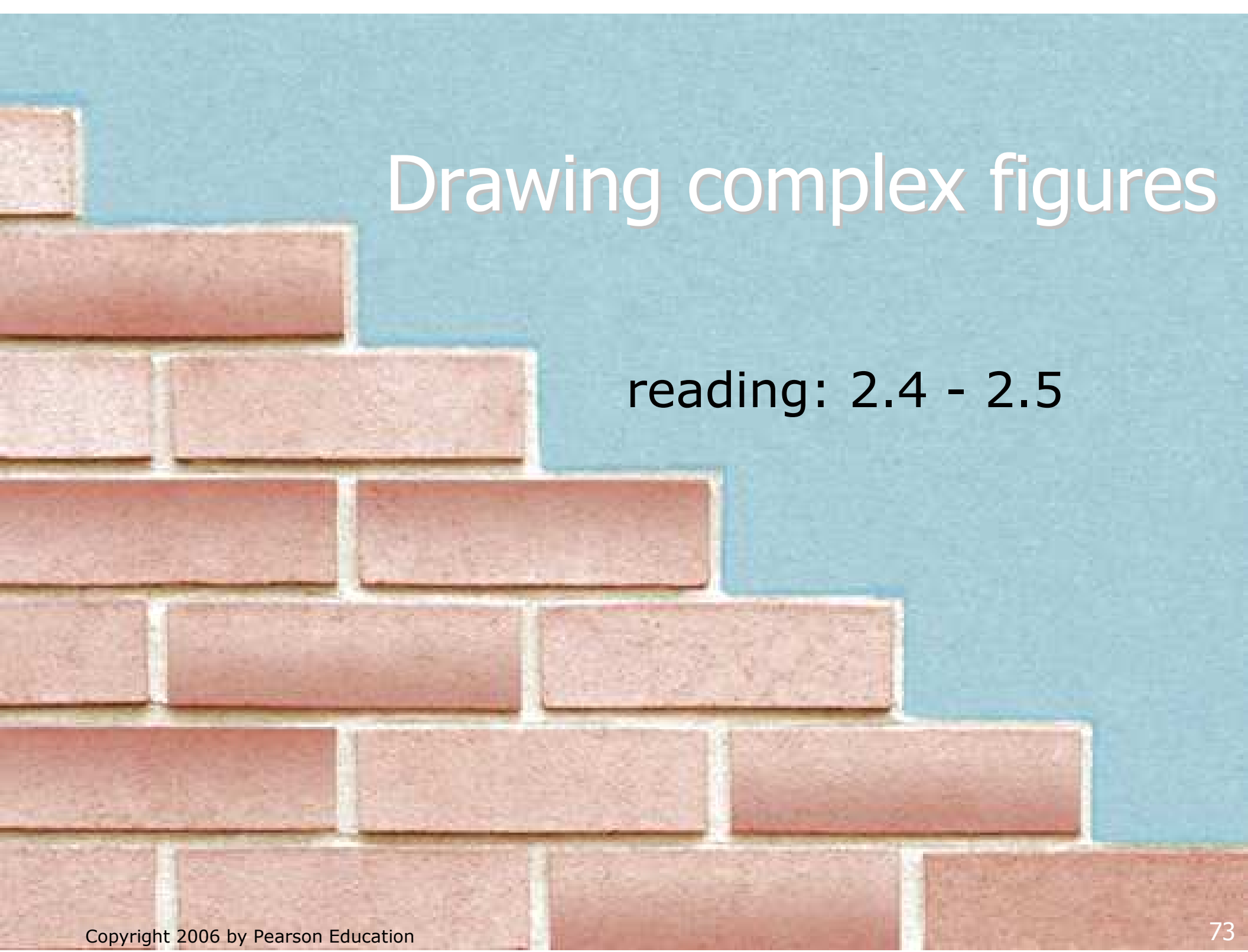
How to comment: for loops

- Place a comment on complex loops explaining *what* they do conceptually, not the mechanics of the syntax.
  - Bad:

```
// This loop repeats 10 times, with i from 1 to 10.
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 5; j++) {  // loop goes 5 times
        System.out.print(j);  // print the j
    }
    System.out.println();
}
```

  - Better:

```
// Prints 12345 ten times on ten separate lines.
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.print(j);
    }
    System.out.println();  // end the line of output
}
```



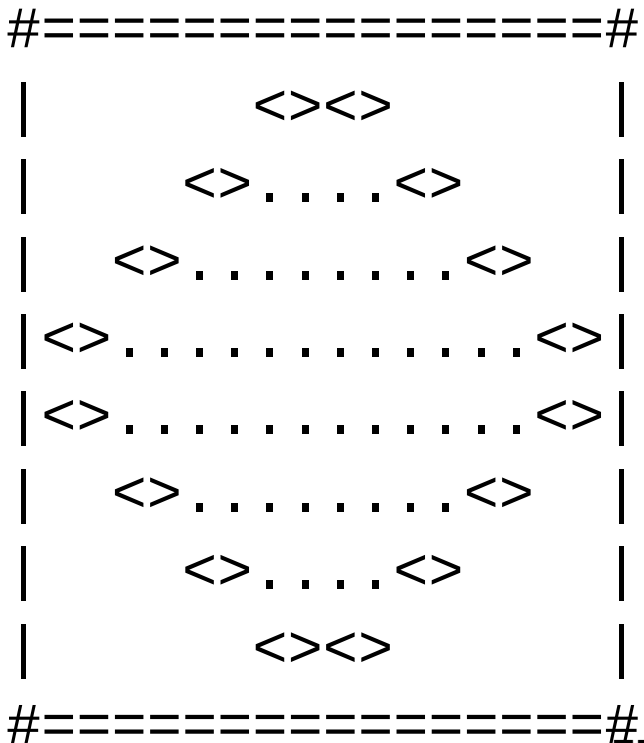
Drawing complex figures

- 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 #*

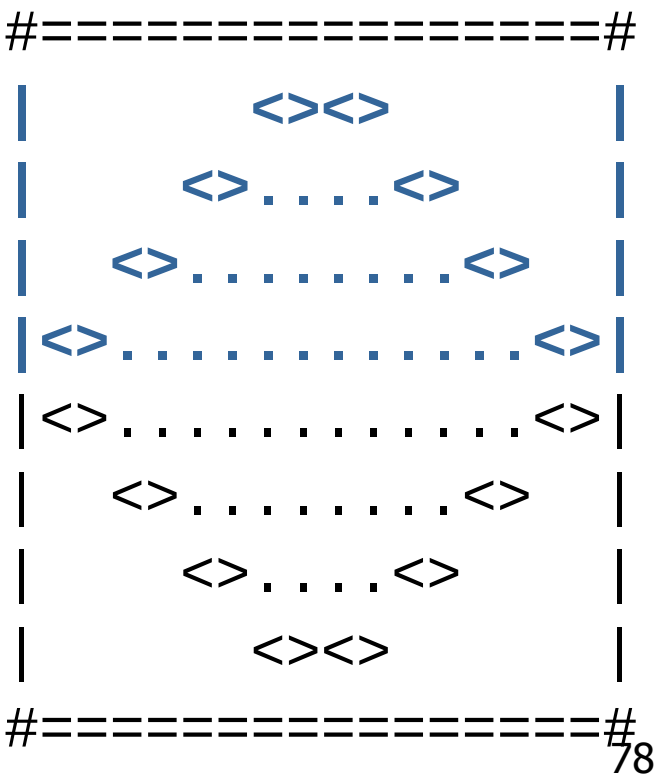
- Our pseudo-code suggests we should use a table to learn the pattern in the top and bottom halves of the figure.

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

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?

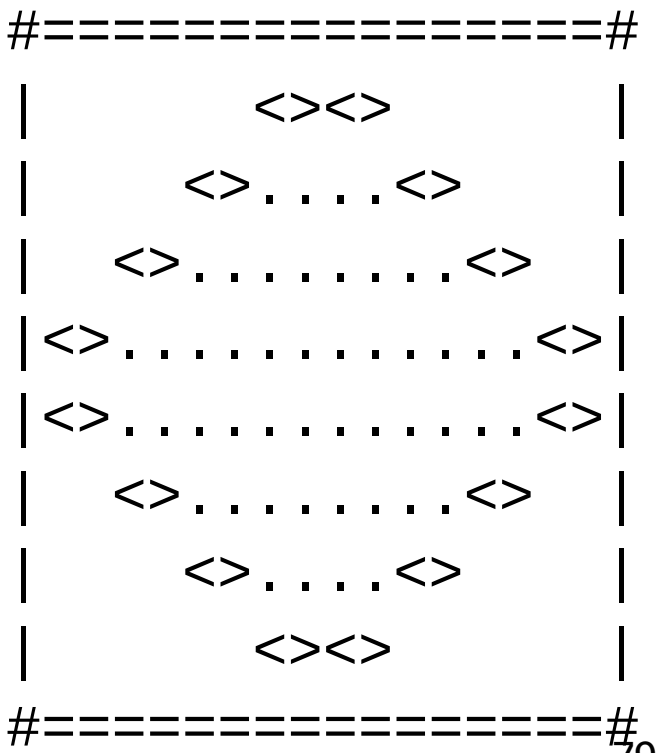
line	spaces	line * -2 + 8	dots	4 * line - 4
1	6	6	0	0
2	4	4	4	4
3	2	2	8	8
4	0	0	12	12



Implementing the figure

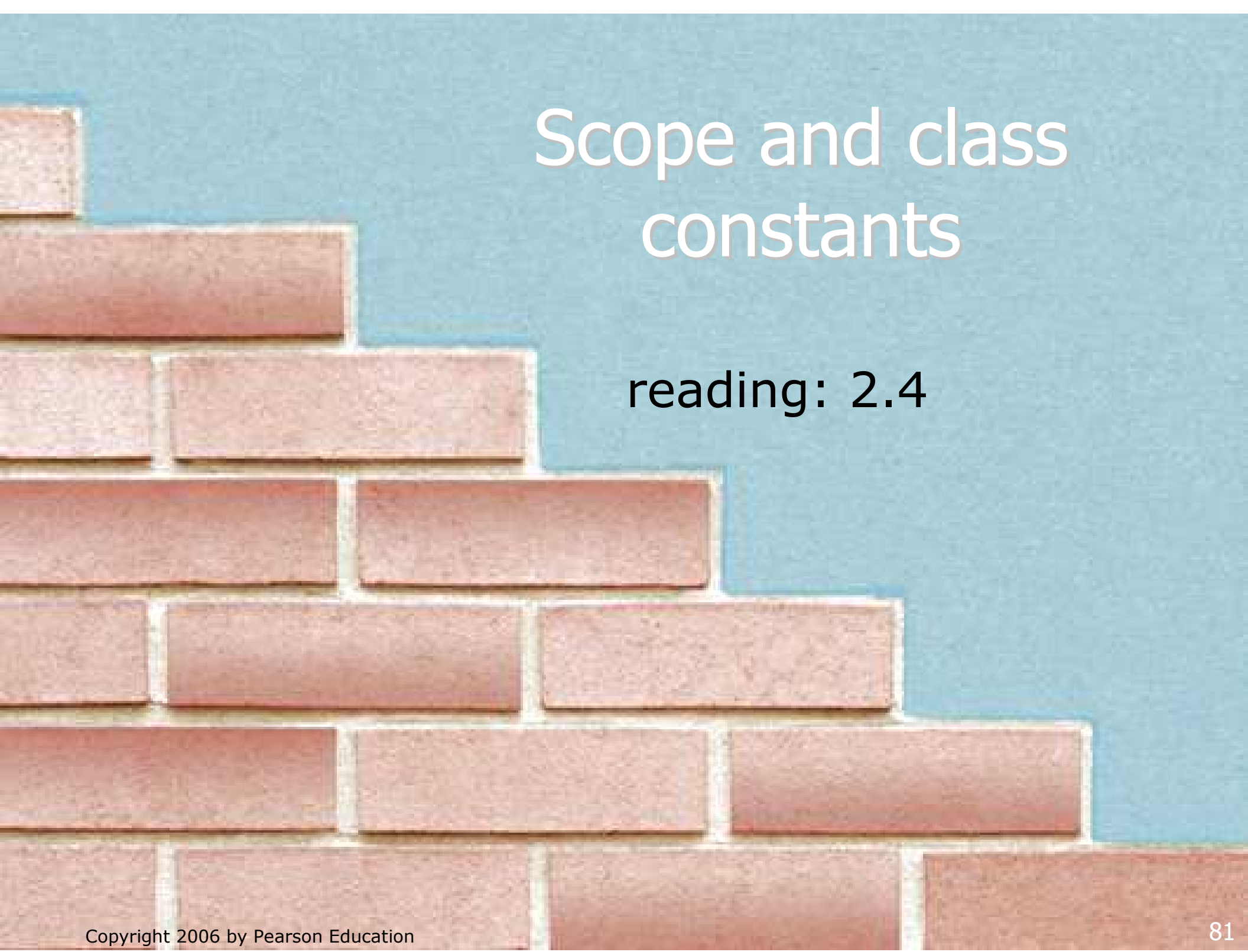
- Let's implement the code for this figure together.

- 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

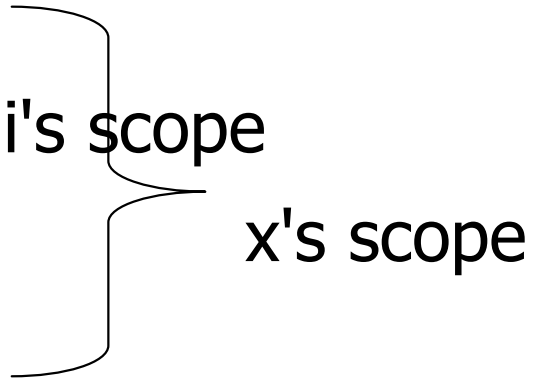
```
// Prints the expanding pattern of <> for the top half of the figure.
public static void drawTopHalf() {
    for (int line = 1; line <= 4; line++) {
        System.out.print("|");

        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;
```

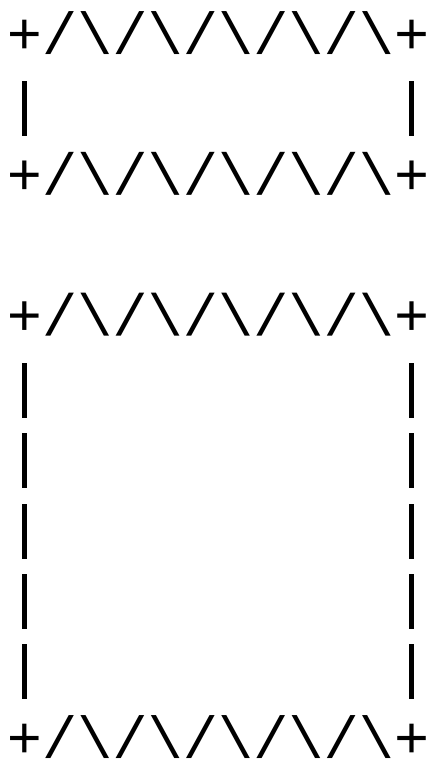
Class constant example

- 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.

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

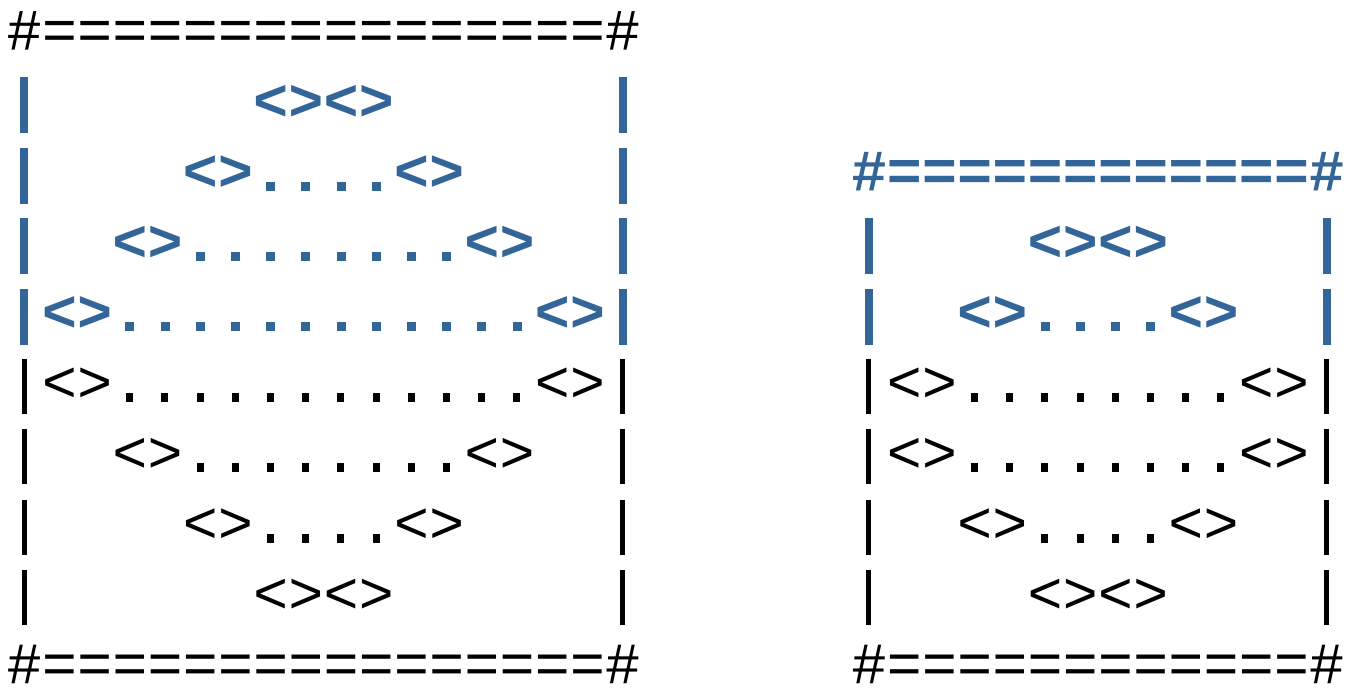
A figure of size 3:

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

Loop tables and constant

- Let's modify our loop table to take into account `SIZE`:

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



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 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.