

2.11 Nested Iteration

Self-Study Guide · About 40 minutes · With answer key

By the end you will be able to

- Recognize a nested loop and tell which loop is the outer loop and which is the inner loop.
- Explain why the inner loop runs all of its iterations, starting fresh, before the outer loop moves on.
- Trace nested loops step by step and predict their exact output, including how many times the inner body runs.
- Use `print` and `println` inside nested loops to predict the rows and columns of a printed pattern.
- Predict what happens when the inner loop's condition uses the outer loop's variable.
- Find and fix common nested-loop bugs, such as a reused loop variable or an inner counter that never resets.

What to remember from earlier

- A `while` loop checks its condition before every iteration, and the body never runs if the condition is false at the start. ([Topic 2.7: while Loops, 2.7.3 When the Body Never Runs](#))
- A `for` loop header has an initialization, a condition, and an update. ([Topic 2.8: for Loops, 2.8.1 The Three Parts of a for Loop Header](#))
- The initialization runs once, and the update runs after each pass through the body. ([Topic 2.8: for Loops, 2.8.2 The Order a for Loop Runs In](#))
- `print` stays on the same line, while `println` ends the line. ([Topic 1.3: Expressions and Output, 1.3.2 print Versus println](#))
- `substring(i, i + 1)` pulls one character out of a string. ([Topic 1.15: String Manipulation, 1.15.5 Extracting Text with substring](#))

2.11.1 A Loop Inside a Loop

A track coach plans two practice rounds, and in each round the runners do three laps. Writing the plan out by hand, you would write "Round 1", then lap 1, lap 2, lap 3, then "Round 2", then lap 1, lap 2, lap 3 again. Notice the shape: there is a repeated task (a round), and *inside* each repetition there is another repeated task (the laps). A single loop from Topics 2.7 and 2.8 can repeat the rounds, but each round itself needs repetition too.

Java handles this the way you would expect once you remember one fact: a loop's body can hold any statements at all, and a loop is a statement. So the body of one loop can contain a whole second loop.

A loop that appears inside the body of another loop is called a **nested loop**, and using loops this way is called **nested iteration**. The loop on the outside is the **outer loop**; the loop inside its body is the **inner loop**. The outer loop's body is everything between its braces, which here is a `print` statement *and* the entire inner loop.

KEY IDEA

A nested loop is a loop written inside the body of another loop. The outside one is the outer loop; the one inside its body is the inner loop.

WORKED EXAMPLE Rounds and laps

Look for two `for` headers. The second one (line 4) sits inside the braces of the first (lines 2–8), so it is part of the outer loop's body.

```
1 for (int round = 1; round <= 2; round++)
2 {
3     System.out.println("Round " + round);
4     for (int lap = 1; lap <= 3; lap++)
5     {
6         System.out.println("  lap " + lap);
7     }
8 }
```

- **Lines 1–8** are the outer loop. Its loop control variable, `round`, takes the values 1 and 2.
- **Line 3** is the first statement of the outer loop's body. It runs once per round.
- **Lines 4–7** are the inner loop, the second statement of the outer loop's body. Its variable, `lap`, takes the values 1, 2, 3.
- **Line 6** is the inner loop's body. It runs once per lap, and the laps happen once per round.

```
Round 1
  lap 1
  lap 2
  lap 3
Round 2
  lap 1
  lap 2
  lap 3
```

The two spaces at the start of `" lap "` only indent the output so the structure is easy to see. Line 3 ran 2 times (once per round), and line 6 ran 6 times (3 laps in each of 2 rounds).

Any kind of loop can be nested inside any kind of loop. The next program is the same shape built from two `while` loops, and it prints something on one line for each set of an exercise.

WORKED EXAMPLE Sets and reps with while loops

Watch the variable `rep`. It is declared and set to 1 on line 5, which is *inside* the outer loop's body.

```
1 int set = 1;
2 while (set <= 3)
3 {
4     System.out.print("Set " + set + ": ");
5     int rep = 1;
6     while (rep <= 4)
7     {
8         System.out.print("#");
9         rep++;
10    }
11    System.out.println();
12    set++;
13 }
```

The outer loop (lines 2–13) runs while `set <= 3`. Each time through its body:

1. Line 4 prints the label, such as `Set 1:` , without ending the line (`print`, not `println`).
2. Line 5 creates `rep` and sets it to 1.
3. The inner loop (lines 6–10) prints `#` and adds 1 to `rep` while `rep <= 4`, so it prints four `#` characters on the same line and stops when `rep` becomes 5.
4. Line 11 ends the line, and line 12 moves `set` to the next value.

```
Set 1: #####
Set 2: #####
Set 3: #####
```

Compared with the first example, the loop type changed, but the idea did not: the inner loop is just one more statement in the outer loop's body, so it runs once every time that body runs.

WATCH OUT

Reusing the outer loop's variable name. Many loops use `i` as the counter, so students often copy a loop header and paste it inside another loop without renaming anything:

SameName.java

```
1 public class SameName
2 {
3     public static void main(String[] args)
4     {
5         for (int i = 1; i <= 3; i++)
6         {
7             for (int i = 1; i <= 2; i++)
8             {
9                 System.out.println("Hi");
10            }
11        }
12    }
13 }
```

This does not compile. Java reports something like:

```
SameName.java:7: error: variable i is already defined in method main(String[])
```

(The exact wording can differ between Java versions.) The outer loop's `i` still exists while its body runs, and line 7 is inside that body, so line 7 tries to create a second variable with the same name in the same place. Java does not allow two variables with one name there, because it would have no way to know which `i` you meant. The fix is to give the inner loop its own name:

```
1 for (int i = 1; i <= 3; i++)
2 {
3     for (int j = 1; j <= 2; j++)
4     {
5         System.out.println(i + " " + j);
6     }
7 }
```

This version compiles and prints the pairs 1 1, 1 2, 2 1, 2 2, 3 1, 3 2, one per line. Section 2.11.2 explains why the pairs come out in that order.

PAUSE AND TRY

Question 1 Tracing

Write the exact output of this code. Pay attention to which statements are inside the inner loop and which are only inside the outer loop.

```
1 for (int song = 1; song <= 2; song++)
2 {
3     System.out.println("Song " + song);
4     for (int verse = 1; verse <= 2; verse++)
5     {
6         System.out.println("verse " + verse);
7     }
8     System.out.println("chorus");
9 }
```

Question 2 Multiple choice

How many lines does this code print in total?

```
1 for (int floor = 1; floor <= 4; floor++)
2 {
3     System.out.println("Floor " + floor);
4     for (int room = 1; room <= 3; room++)
5     {
6         System.out.println("room " + room);
7     }
8 }
```

- A. 7
- B. 12
- C. 16
- D. 4

2.11.2 The Inner Loop Finishes First

You now know *what* runs in a nested loop. The harder question is *in what order*. When the outer loop has done one iteration and the inner loop has done one, which goes next? Getting this right is the whole skill of tracing nested loops.

Think of a car's odometer. The rightmost digit rolls through 0, 1, 2, ... 9, and only after it has gone all the way around does the digit to its left move up by one. Then the rightmost digit starts over from 0. The inner loop is the fast digit; the outer loop is the slow one. (The analogy stops there: an odometer digit always has ten values, but an inner loop can run any number of times, even a different number each time, as you will see in 2.11.4.)

Here is the rule in Java terms. Each time the outer loop's body runs, Java reaches the inner loop as a brand-new statement:

1. The inner loop starts from the beginning. For a `for` loop, that means its initialization runs again, so the inner variable starts over.
2. The inner loop runs **all** of its iterations until its condition is false.
3. Only then does Java continue with the rest of the outer loop's body, and after that the outer loop does its update and checks its own condition.

The loops never take turns one step at a time. The outer loop's variable stays the same value the whole time the inner loop is running.

KEY IDEA

Each time the outer loop runs its body, the inner loop starts over and completes every one of its iterations before the outer loop can go on to its next iteration.

WORKED EXAMPLE Tracing the order step by step

Both loops here are short, so you can follow every check of every condition.

```

1  for (int outer = 1; outer <= 3; outer++)
2  {
3      for (int inner = 1; inner <= 2; inner++)
4      {
5          System.out.println(outer + " " + inner);
6      }
7  }
```

Each row of the table is one moment when a condition is checked. Read it top to bottom.

outer	inner	Condition checked	Printed
1		outer <= 3 is true	
1	1	inner <= 2 is true	1 1
1	2	inner <= 2 is true	1 2
1	3	inner <= 2 is false	
2		outer <= 3 is true	
2	1	inner <= 2 is true	2 1

2	2	inner <= 2 is true	2 2
2	3	inner <= 2 is false	
3		outer <= 3 is true	
3	1	inner <= 2 is true	3 1
3	2	inner <= 2 is true	3 2
3	3	inner <= 2 is false	
4		outer <= 3 is false	

Step by step: `outer` starts at 1 and its check passes. Java reaches the inner `for` loop and runs its initialization, so `inner` becomes 1. The inner loop prints 1 1, updates `inner` to 2, prints 1 2, updates `inner` to 3, and then its check fails. That inner loop is finished, which means the outer loop's body is finished too. Now the outer update runs (`outer` becomes 2) and the outer check passes. Java reaches the inner loop again, and its initialization sets `inner` back to 1. The same pattern repeats for `outer` equal to 3. Finally `outer` becomes 4, the outer check fails, and the whole nested loop is done.

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

Look at the printed pairs: the second number (inner) changes on every line, and the first number (outer) changes only after the second has gone through all its values.

Because the inner loop runs completely every time, you can count how many times the inner body runs: if the outer loop runs 3 times and the inner loop runs 2 times on each of those, the inner body runs $3 \times 2 = 6$ times. The next example lets the program do that counting.

WORKED EXAMPLE Counting the games

Four teams each play three games. Instead of printing, the inner body adds 1 to `count` (compound assignment, Topic 1.6).

```
1 int count = 0;
2 for (int team = 1; team <= 4; team++)
3 {
4     for (int game = 1; game <= 3; game++)
5     {
6         count++;
7     }
8 }
9 System.out.println("Games played: " + count);
```

team	game values that pass the check	count after the inner loop ends
1	1, 2, 3	3
2	1, 2, 3	6

3	1, 2, 3	9
4	1, 2, 3	12

Games played: 12

What changed from the last example is only what the inner body does, and the rule did the rest: `game` restarts at 1 for every team, so each team adds exactly 3 to `count`, and 4×3 is 12. Notice that `count` is declared *before* both loops, because it must keep its value across all the teams.

WATCH OUT

An inner counter that never starts over. With `while` loops, you write the inner loop's starting value yourself. If you put it before the outer loop, it runs only once:

```
1 int count = 0;
2 int team = 1;
3 int game = 1;
4 while (team <= 4)
5 {
6     while (game <= 3)
7     {
8         count++;
9         game++;
10    }
11    team++;
12 }
13 System.out.println("Games played: " + count);
```

A student expects 12 but sees:

Games played: 3

Trace it. On the first team, `game` goes 1, 2, 3, and the inner loop stops when `game` is 4, with `count` at 3. For teams 2, 3, and 4, nothing sets `game` back to 1. It is still 4, so `game <= 3` is false the first time it is checked, and the inner body runs zero times (a loop body never runs when its condition starts out false, Topic 2.7). The program compiles and runs, so only the wrong output reveals the bug.

The fix is to move the starting value into the outer loop's body, just before the inner loop, so it runs once per team:

```
1 int count = 0;
2 int team = 1;
3 while (team <= 4)
4 {
5     int game = 1;
6     while (game <= 3)
7     {
8         count++;
9         game++;
10    }
11    team++;
12 }
13 System.out.println("Games played: " + count);
```

Now each team starts with `game` equal to 1, and the program prints Games played: 12. A `for` inner loop avoids this mistake automatically, because its initialization is part of its header and runs every time the inner loop starts.

PAUSE AND TRY

Question 3 Tracing

Write the exact output. Notice that the inner loop counts down.

```
1 for (int a = 1; a <= 2; a++)
2 {
3     for (int b = 3; b >= 1; b--)
4     {
5         System.out.println(a + "-" + b);
6     }
7 }
```

Question 4 Multiple choice

What does this code print?

```
1 int count = 0;
2 int week = 0;
3 while (week < 5)
4 {
5     for (int day = 1; day <= 4; day++)
6     {
7         count++;
8     }
9     week++;
10 }
11 System.out.println(count);
```

- A. 9
- B. 20
- C. 4
- D. 25

2.11.3 Rows and Columns with print and println

Nested loops are the natural tool whenever something has rows *and* columns: seats in a theater, a calendar page, a grid of tiles, or a picture drawn with characters. Pattern printing is a favorite way to practice reading nested loops, because the output shows the structure of the loops directly.

The trick depends on the difference between the two print statements from Topic 1.3.

`System.out.print` prints and stays on the same line, so the next thing printed appears right after it. `System.out.println()` with nothing in the parentheses prints nothing and just ends the current line, so the next output starts on a new line.

Put those together with nested loops and you get a plan:

- The **inner loop** prints one row, one column at a time, using `print` so everything stays on that row.
- **After** the inner loop ends, but still inside the outer loop's body, a `println()` ends the row.
- The **outer loop** repeats this once per row.

So the outer loop decides *how many rows* there are, and the inner loop decides *how many columns* are in each row.

KEY IDEA

To print a grid, let the inner loop print one row with `print`, put `println()` after the inner loop to end the row, and let the outer loop repeat that for every row.

WORKED EXAMPLE A rectangle of stars

This prints 3 rows of 5 stars. Find the `println()` on line 7 and check which braces it is between.

```
1 for (int row = 1; row <= 3; row++)
2 {
3     for (int col = 1; col <= 5; col++)
4     {
5         System.out.print("*");
6     }
7     System.out.println();
8 }
```

Line 7 is after the inner loop's closing brace (line 6) but before the outer loop's closing brace (line 8), so it runs once per row.

row	What the inner loop does	Line 7	Output so far
1	col 1 to 5: prints * 5 times	ends line 1	1 row of *****
2	col restarts at 1: 5 more *	ends line 2	2 rows
3	col restarts at 1: 5 more *	ends line 3	3 rows

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

To make the rectangle wider, change the inner loop's condition (`col <= 5`); to make it taller, change the outer loop's condition (`row <= 3`).

The inner body does not have to print the same thing every time. It can print one of the loop variables, and which one you choose changes the pattern completely.

WORKED EXAMPLE Printing a loop variable instead of a star

Here is the same pair of loops, but line 5 prints `col`, the inner variable:

```
1 for (int row = 1; row <= 3; row++)
2 {
3     for (int col = 1; col <= 5; col++)
4     {
5         System.out.print(col);
6     }
7     System.out.println();
8 }
```

```
12345
12345
12345
```

Within a row, `col` goes 1, 2, 3, 4, 5, so each row shows 12345, and the row starts over because `col` restarts. Now change only line 5 to print `row`, the outer variable:

```
1 for (int row = 1; row <= 3; row++)
2 {
3     for (int col = 1; col <= 5; col++)
4     {
5         System.out.print(row);
6     }
7     System.out.println();
8 }
```

```
11111
22222
33333
```

During one row, `row` does not change while the inner loop runs, so the same digit is printed 5 times. Then `row` goes up by one for the next line. This pair of outputs is a quick way to see the rule from 2.11.2: the inner variable changes across a row; the outer variable changes only from row to row.

WATCH OUT

println in the wrong place. A student wants 2 rows of 4 stars, but puts `println()` inside the inner loop's braces:

```
1 for (int row = 1; row <= 2; row++)
2 {
3     for (int col = 1; col <= 4; col++)
4     {
5         System.out.print("*");
6         System.out.println();
7     }
8 }
```

Now the line ends after every star, and the program prints 8 lines with one star each:

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

Students do this because "end the row" feels like part of printing the stars. But anything inside the inner loop's braces runs once per column. Ending a row should happen once per row, so `println()` belongs after the inner loop's closing brace:

```
1 for (int row = 1; row <= 2; row++)
2 {
3     for (int col = 1; col <= 4; col++)
4     {
5         System.out.print("*");
6     }
7     System.out.println();
8 }
```

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

PAUSE AND TRY

Question 5 Tracing

Write the exact output of this code.

```
1 for (int r = 1; r <= 2; r++)
2 {
3     for (int c = 1; c <= 4; c++)
4     {
5         System.out.print(r * c);
6     }
7     System.out.println();
8 }
```

Question 6 Debugging

This code should print 4 rows of `###`, but it prints `#####` on a single line. Find the error and correct it.

```

1 for (int row = 1; row <= 4; row++)
2 {
3     for (int col = 1; col <= 3; col++)
4     {
5         System.out.print("#");
6     }
7 }
```

2.11.4 When the Inner Loop Depends on the Outer Loop

So far, every inner loop ran the same number of times on every pass. That gives rectangles. But many real tasks change from row to row: in a seating chart for a stadium section, the first row might have 1 seat, the next 2, and so on; a staircase gets one step longer each line.

The outer loop's variable can help here. Remember from 2.11.2 that while the inner loop runs, the outer variable holds steady. That means the inner loop's condition can use the outer variable, and on each pass of the outer loop the inner loop gets a different stopping point. When the outer variable is 1, the inner loop might run once; when it is 4, four times.

KEY IDEA

When the inner loop's condition (or starting value) uses the outer loop's variable, the inner loop can run a different number of times on each pass of the outer loop.

WORKED EXAMPLE A triangle of stars

Compare line 3 with the rectangle in 2.11.3. The inner condition is now `star <= row` instead of a fixed number.

```

1 for (int row = 1; row <= 4; row++)
2 {
3     for (int star = 1; star <= row; star++)
4     {
5         System.out.print("*");
6     }
7     System.out.println();
8 }
```

row	Inner check	Times inner body runs	Line printed
1	<code>star <= 1</code>	1	<code>*</code>
2	<code>star <= 2</code>	2	<code>**</code>
3	<code>star <= 3</code>	3	<code>***</code>

4	star <= 4	4	****
---	-----------	---	------

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

The inner body ran $1 + 2 + 3 + 4 = 10$ times in total. When the inner loop's count changes, you cannot multiply two numbers anymore; add up the count for each pass of the outer loop instead.

WORKED EXAMPLE A shrinking staircase of numbers

Two things vary from the triangle: the outer loop counts *down* from 4, and the inner body prints `col` instead of a star.

```
1 for (int row = 4; row >= 1; row--)
2 {
3     for (int col = 1; col <= row; col++)
4     {
5         System.out.print(col);
6     }
7     System.out.println();
8 }
```

row	col values printed	Line printed
4	1, 2, 3, 4	1234
3	1, 2, 3	123
2	1, 2	12
1	1	1

```
1234
123
12
1
```

The outer loop runs for `row` equal to 4, 3, 2, 1 and stops when `row` becomes 0. Each pass, `col` starts over at 1 and goes up to the current `row`, so every line is one shorter than the one before.

WATCH OUT

An off-by-one inner condition. Off-by-one errors (Topic 2.7) are easy to make in an inner loop, and a nested loop repeats the mistake on every row. Here a student meant to print the triangle above but wrote `<` instead of `<=`:

```
1 for (int row = 1; row <= 4; row++)
2 {
3     for (int star = 1; star < row; star++)
4     {
5         System.out.print("*");
6     }
7     System.out.println();
8 }
```

The output begins with an empty line:

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

When `row` is 1, the check `star < 1` is false right away, so the inner body runs zero times and line 7 prints an empty line. Each later row gets one star fewer than intended, and the four-star row never appears. Changing the condition to `star <= row` (or starting `star` at 0) gives every row the right number of stars. Whenever you write an inner condition that uses the outer variable, test it by hand on the first row and the last row.

PAUSE AND TRY

Question 7 Tracing

Here the inner loop's *starting value* uses the outer variable. Write the exact output.

```
1 for (int i = 1; i <= 3; i++)
2 {
3     for (int j = i; j <= 3; j++)
4     {
5         System.out.print(j);
6     }
7     System.out.println();
8 }
```

Question 8 Multiple choice

What does this code print?

```
1 int count = 0;
2 for (int row = 1; row <= 4; row++)
3 {
4     for (int k = 1; k < row; k++)
5     {
6         count++;
7     }
8 }
9 System.out.println(count);
```

- A. 10
- B. 16
- C. 6
- D. 12

Putting it together

In Topic 2.10 you used one loop to look at each character of a string. Some questions need two loops: "Does any letter appear more than once in this word?" To answer it, you have to compare every letter with every letter that comes after it. The outer loop picks a letter; the inner loop walks through the letters after it. This program combines everything in the chapter: an inner loop that runs completely for each outer pass, an inner starting value that depends on the outer variable, and an `if` statement (Topic 2.3) inside the inner body.

WORKED EXAMPLE Finding matching letters in a word

The word is "level", with indexes 0 to 4. Line 9 saves the outer letter; line 12 gets the inner letter, using `substring(i, i + 1)` to get one character as a string (Topic 1.15).

RepeatedLetters.java

```

1 public class RepeatedLetters
2 {
3     public static void main(String[] args)
4     {
5         String word = "level";
6         int pairs = 0;
7         for (int i = 0; i < word.length(); i++)
8         {
9             String first = word.substring(i, i + 1);
10            for (int j = i + 1; j < word.length(); j++)
11            {
12                String second = word.substring(j, j + 1);
13                if (first.equals(second))
14                {
15                    System.out.println(first + " at " + i + " and " + j);
16                    pairs++;
17                }
18            }
19        }
20        System.out.println("Matching pairs: " + pairs);
21    }
22 }
```

The inner loop starts at `j = i + 1` (line 10), so each letter is compared only with the letters after it. That way no letter is compared with itself, and no pair is checked twice.

i	first	j values compared	Matches found
0	l	1 (e), 2 (v), 3 (e), 4 (l)	j = 4: prints l at 0 and 4
1	e	2 (v), 3 (e), 4 (l)	j = 3: prints e at 1 and 3

2	v	3 (e), 4 (l)	none
3	e	4 (l)	none
4	l	none: j starts at 5, and 5 < 5 is false	none

Each time the `if` condition on line 13 is true, lines 15 and 16 print the pair and add 1 to `pairs`. After the outer loop ends, line 20 prints the total:

```
l at 0 and 4
e at 1 and 3
Matching pairs: 2
```

Three details to notice. First, `pairs` is declared before both loops (line 6) because it counts across the whole word; `first` is declared inside the outer loop because it belongs to one outer pass. Second, the inner loop made $4 + 3 + 2 + 1 + 0 = 10$ comparisons, fewer each time because its start depends on `i`. Third, on the last pass the inner loop runs zero times, which is not an error; the condition is simply false from the start.

Vocabulary

Term	Definition	Example
nested loop	A loop written inside the body of another loop	a <code>for</code> loop inside a <code>for</code> loop's braces
nested iteration	Repeating a task inside each repetition of another task, by placing one loop in another loop's body	rows of a pattern, each built column by column
outer loop	The loop whose body contains another loop; its variable changes only after the inner loop finishes	<code>for (int row = 1; row <= 3; row++)</code>
inner loop	The loop inside the outer loop's body; it starts over and runs completely on every outer pass	<code>for (int col = 1; col <= 5; col++)</code>

Chapter summary

- A nested loop is a loop inside the body of another loop. Any loop type can be nested in any other, and the inner loop is just one more statement in the outer loop's body.
- Each time the outer loop's body runs, the inner loop starts over and completes all of its iterations before the outer loop continues. The inner variable changes fastest; the outer variable stays the same while the inner loop runs.
- If the inner loop runs the same number of times on every pass, the inner body runs (outer passes) $\times$ (inner passes) times. If its count changes, add up the count for each pass.
- To print a grid, the inner loop prints one row with `print`, a `println()` after the inner loop ends the row, and the outer loop repeats for each row.

- When the inner loop's condition or starting value uses the outer variable, each row can be a different length, as in a triangle.
- Watch for a reused loop variable name (a compile error), an inner `while` counter set before the outer loop (it never restarts), a misplaced `println()`, and off-by-one inner conditions.

Review questions

Question 9 Tracing

This code combines a nested loop with a running total. Write the exact output.

```
1 for (int row = 1; row <= 4; row++)
2 {
3     int sum = 0;
4     for (int n = 1; n <= row; n++)
5     {
6         sum += n;
7     }
8     System.out.println("Row " + row + ": " + sum);
9 }
```

Question 10 Multiple choice

The blanks in this code are filled in to produce exactly this output:

```
1
22
333
```

```
1 for (int row = 1; row <= 3; row++)
2 {
3     for (int col = 1; /* condition */; col++)
4     {
5         System.out.print(/* value */);
6     }
7     System.out.println();
8 }
```

Which condition and value produce it?

- A. condition `col <= 3`, value `row`
- B. condition `col <= row`, value `col`
- C. condition `col <= row`, value `row`
- D. condition `col <= 3`, value `col`

Question 11 Debugging

This code should print a triangle with rows of 1, 2, and 3 stars. Instead it prints a single star on each of three lines. Find the error and correct it.

```
1 int row = 1;
2 int star = 1;
3 while (row <= 3)
4 {
5     while (star <= row)
6     {
7         System.out.print("*");
8         star++;
9     }
10    System.out.println();
11    row++;
12 }
```

Question 12 Short answer

A student predicts that this code prints 11 and then 22, because "the outer loop does one step, then the inner loop does one step." Explain what it actually prints and what is wrong with the student's reasoning.

```
1 for (int a = 1; a <= 2; a++)
2 {
3     for (int b = 1; b <= 2; b++)
4     {
5         System.out.println(a + " " + b);
6     }
7 }
```

Question 13 Multiple choice

What does this code print?

```
1 int count = 0;
2 for (int a = 1; a <= 4; a++)
3 {
4     for (int b = 1; b <= 4; b++)
5     {
6         if (a + b == 5)
7         {
8             count++;
9         }
10    }
11 }
12 System.out.println(count);
```

- A. 16
- B. 4
- C. 2
- D. 1

Question 14 Tracing

This code uses nested loops on a string. Write the exact output.

```
1 String word = "cat";
2 for (int i = 0; i < word.length(); i++)
3 {
4     for (int j = 0; j <= i; j++)
5     {
6         System.out.print(word.substring(i, i + 1));
7     }
8     System.out.println();
9 }
```

Reflection

- ☐ I can point to the outer loop and the inner loop in a nested loop and say what is in the outer loop's body. Reread: 2.11.1 A Loop Inside a Loop
- ☐ I can explain why reusing the outer loop's variable name for the inner loop does not compile. Reread: 2.11.1 A Loop Inside a Loop
- ☐ I can explain why the inner loop finishes all its iterations before the outer loop moves on, and trace the order of the output. Reread: 2.11.2 The Inner Loop Finishes First
- ☐ I can count how many times an inner loop body runs, and spot an inner while counter that never starts over. Reread: 2.11.2 The Inner Loop Finishes First
- ☐ I can predict the rows and columns a nested loop prints, and say where println() belongs. Reread: 2.11.3 Rows and Columns with print and println
- ☐ I can trace an inner loop whose condition or starting value uses the outer loop's variable. Reread: 2.11.4 When the Inner Loop Depends on the Outer Loop

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Song 1
verse 1
verse 2
chorus
Song 2
verse 1
verse 2
chorus
```

The outer loop's body has three statements: line 3, the inner loop on lines 4–7, and line 8. For `song` equal to 1, line 3 prints `Song 1`, the inner loop prints `verse 1` and `verse 2`, and then line 8 prints `chorus`. Then the same three steps happen for `song` equal to 2. Line 8 is after the inner loop's closing brace on line 7, so it runs once per song, not once per verse.

Reread 2.11.1 *A Loop Inside a Loop* if your output differed.

Question 2

C. The outer loop runs 4 times. Each time, line 3 prints one line and the inner loop prints 3 lines, so each floor produces 4 lines: $4 \times 4 = 16$. The output starts `Floor 1`, `room 1`, `room 2`, `room 3`, `Floor 2`, and so on through `room 3` for floor 4.

- **A** is wrong: it adds the loop counts ($4 + 3$), as if the inner loop ran only once in total. The inner loop runs again for every floor.
- **B** is wrong: 12 counts only the `room` lines (4×3) and forgets the `Floor` line printed on line 3 for each floor.
- **D** is wrong: 4 counts only the `Floor` lines, as if the inner loop were not part of the outer loop's body.

Reread 2.11.1 *A Loop Inside a Loop* if you chose differently.

Question 3

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

For `a` equal to 1, the inner loop starts `b` at 3 and runs while `b >= 1`, printing `1-3`, `1-2`, `1-1`. When `b` reaches 0, the inner loop ends, `a` becomes 2, and the inner loop starts over with `b` back at 3, printing `2-3`, `2-2`, `2-1`. The inner value changes on every line; the outer value changes only after the inner loop has finished.

Reread 2.11.2 *The Inner Loop Finishes First* if your output differed.

Question 4

B. The outer `while` loop runs for `week` equal to 0, 1, 2, 3, and 4: five times (it stops when `week` becomes 5). Each time, the inner `for` loop starts `day` at 1 and runs for `day` equal to 1 through 4: four times. The inner body runs $5 \times 4 = 20$ times, adding 1 to `count` each time.

- **A** is wrong: it adds the counts ($5 + 4$) instead of multiplying. The inner loop's 4 iterations happen again for every week.
- **C** is wrong: it counts the inner loop only once, as if it did not start over. Because `day` is initialized in the `for` header, it restarts at 1 every week.
- **D** is wrong: it treats the inner loop as running 5 times too. `day <= 4` allows only 1, 2, 3, 4.

Reread 2.11.2 *The Inner Loop Finishes First* if you chose differently.

Question 5

```
1234
2468
```

For `r` equal to 1, the inner loop prints $1 * 1$, $1 * 2$, $1 * 3$, $1 * 4$, which are 1, 2, 3, 4, all on one line, and line 7 ends the line. For `r` equal to 2, `c` starts over at 1 and the inner loop prints 2, 4, 6, 8. Because `print` adds no spaces, the numbers run together.

Reread 2.11.3 *Rows and Columns with print and println* if your output differed.

Question 6

Nothing ever ends a row. The inner loop prints three `#` with `print`, and the outer loop repeats that four times, but no statement moves to a new line, so all 12 characters land on one line. Add `System.out.println();` after the inner loop's closing brace, still inside the outer loop:

```
1 for (int row = 1; row <= 4; row++)
2 {
3     for (int col = 1; col <= 3; col++)
4     {
5         System.out.print("#");
6     }
7     System.out.println();
8 }
```

Line 7 now runs once per row, after each row's three `#` characters:

```
###
###
###
###
```

Putting the `println()` inside the inner loop instead would print 12 lines with one `#` each, and putting it after the outer loop would end only the single long line.

Reread 2.11.3 *Rows and Columns with print and println* if you missed it.

Question 7

```
123
23
3
```

i	j values printed	Line printed
1	1, 2, 3	123
2	2, 3	23
3	3	3

The inner loop always stops after 3, but it starts at the current value of `i`, so each row starts one number later and is one number shorter.

Reread 2.11.4 *When the Inner Loop Depends on the Outer Loop* if your output differed.

Question 8

C. The inner loop runs while `k < row`, so it runs `row - 1` times on each pass: 0 times when `row` is 1, then 1, 2, and 3 times. The total is $0 + 1 + 2 + 3 = 6$.

- **A** is wrong: 10 is $1 + 2 + 3 + 4$, which would be right for `k <= row`. With `<`, each pass runs one time fewer.
- **B** is wrong: 16 is 4×4 , which treats the inner loop as running a fixed 4 times. Its stopping point depends on `row`.
- **D** is wrong: 12 is 4×3 , which assumes the inner loop runs 3 times on every pass. It runs 3 times only on the last pass.

Reread 2.11.4 *When the Inner Loop Depends on the Outer Loop* if you chose differently.

Question 9

```
Row 1: 1
Row 2: 3
Row 3: 6
Row 4: 10
```

row	sum reset to	n values added	sum printed
1	0	1	1
2	0	1, 2	3
3	0	1, 2, 3	6
4	0	1, 2, 3, 4	10

Line 3 is inside the outer loop, so `sum` starts at 0 on every pass, and line 8 runs once per row, after the inner loop has finished adding. The inner loop runs `row` times, so each row adds one more number than the row before.

Reread 2.11.4 *When the Inner Loop Depends on the Outer Loop* if your output differed.

Question 10

C. Each line has one more digit than the line before, so the inner loop must run `row` times (`col <= row`). Every digit on a line is the same and equals the line number, so the value printed must be the outer variable, `row`, which stays fixed while the inner loop runs.

- **A** is wrong: with `col <= 3` every row has 3 digits, printing 111, 222, 333.
- **B** is wrong: printing `col` gives the digits counting up within a row, printing 1, 12, 123.
- **D** is wrong: this is a 3-by-3 rectangle of `col` values, printing 123 on all three lines.

Reread 2.11.3 *Rows and Columns with print and println* and 2.11.4 *When the Inner Loop Depends on the Outer Loop* if you chose differently.

Question 11

star is set to 1 only once, before the outer loop (line 2), so it never starts over. Trace it:

row	star when inner loop starts	Stars printed	star when inner loop ends
1	1	1	2
2	2	1	3
3	3	1	4

Each row, `star` picks up where it left off, which happens to be exactly `row`, so the check `star <= row` passes just once. The output is:

```
*
*
*
```

Corrected code: move the declaration into the outer loop's body, right before the inner loop, so every row starts with `star` equal to 1.

```
1 int row = 1;
2 while (row <= 3)
3 {
4     int star = 1;
5     while (star <= row)
6     {
7         System.out.print("*");
8         star++;
9     }
10    System.out.println();
11    row++;
12 }
```

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

Reread 2.11.2 *The Inner Loop Finishes First* if you missed it.

Question 12

Model answer: It prints four lines: 11, 12, 21, 22. The loops do not take turns. When `a` is 1, the inner loop runs all of its iterations (`b` is 1, then 2) before `a` changes, giving 11 and 12. Only then does `a` become 2; the inner loop starts over at `b` equal to 1 and runs completely again, giving 21 and 22. The inner body runs $2 \times 2 = 4$ times, not 2. (The `""` in `a + "" + b` makes Java join the numbers as text instead of adding them, Topic 1.15.)

A complete answer says:

- the actual output is 11, 12, 21, 22
- the inner loop finishes all its iterations before the outer loop moves on
- the inner loop starts over for each value of the outer variable

Reread 2.11.2 *The Inner Loop Finishes First* if your explanation differed.

Question 13

B. The inner body runs $4 \times 4 = 16$ times, once for every combination of `a` and `b` from 1 to 4. But `count` goes up only when the `if` condition is true, and `a + b == 5` is true for just four combinations: `a = 1` with `b = 4`, `2` with `3`, `3` with `2`, and `4` with `1`. So `count` ends at 4.

- **A** is wrong: 16 is how many times the condition is *checked*, not how many times it is true.
- **C** is wrong: it treats `a = 1` with `b = 4` and `a = 4` with `b = 1` as the same pair. The loops visit them as two separate combinations, and the condition is true for both.
- **D** is wrong: it stops after the first match, but nothing in the code stops the loops early; every combination is checked.

Reread 2.11.2 *The Inner Loop Finishes First* and *Putting it together* if you chose differently.

Question 14

```
c
aa
ttt
```

i	Letter at i	Inner j values	Line printed
0	c	0	c
1	a	0, 1	aa
2	t	0, 1, 2	ttt

The outer loop picks the letter at index `i`. The inner loop runs while `j <= i`, which is `i + 1` times, and every pass prints that same letter, because line 6 uses `i` (the outer variable), not `j`. Then line 8 ends the row.

Reread 2.11.4 *When the Inner Loop Depends on the Outer Loop* if your output differed.