

2.12 Informal Run-Time Analysis

Self-Study Guide · About 45 minutes · With answer key

By the end you will be able to

- Explain what a statement execution count is and find one for a statement by tracing.
- Work out how many times a for loop or while loop repeats by reading its header, including loops that count by steps other than 1.
- Count how often a statement inside an if, inside a loop that depends on its data, or inside a loop that never runs is executed.
- Count executions in nested loops, including inner loops whose length changes with the outer loop.
- Compare two code segments by how much work they do and describe how that work grows as the input gets bigger.

What to remember from earlier

- A while loop checks its condition before every pass and stops the first time it is false. ([Topic 2.7: while Loops](#), [2.7.2 Check, Run, Repeat](#))
- A loop that repeats one time too many or too few has an off-by-one error. ([Topic 2.7: while Loops](#), [2.7.5 Off-by-One Errors](#))
- 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](#))
- In nested loops, the inner loop runs completely for every pass of the outer loop. ([Topic 2.11: Nested Iteration](#), [2.11.2 The Inner Loop Finishes First](#))
- The remainder operator % gives what is left after integer division. ([Topic 1.3: Expressions and Output](#), [1.3.7 The Remainder Operator](#))

2.12.1 Counting How Many Times a Statement Runs

Two programs can print exactly the same answer while one of them does far more work to get there. Before you can say which program is faster, you need a way to measure "work" that does not depend on how fast a particular computer happens to be. The simplest measure is to count: how many times does each statement actually run?

Until Topic 2.7, that question had a boring answer. In a program made only of statements one after another, every statement runs exactly once, from top to bottom. Loops changed that. A statement inside a loop body runs once for **every** pass through the loop, so a single line of code can run three times, a thousand times, or not at all.

The number of times a statement is executed while a program runs is called its **statement execution count**. You find it the same way you have traced loops since Topic 2.7: follow the program step by step and keep track of each time the statement runs. Once you see the pattern, you can often skip the full trace and reason about the count from the loop itself. The **run time** of a program is how long it takes to run. Estimating and comparing run times by counting statement executions, instead of timing programs with a stopwatch, is called **informal run-time analysis**.

KEY IDEA

A statement execution count is how many times a statement runs. A statement outside any loop runs once; a statement in a loop body runs once for each pass through the loop.

WORKED EXAMPLE Counting the statements in a while loop

Look for which lines are inside the braces of the `while` loop (lines 6–8) and which are outside it. The variable `bodyCount` is a **counter variable**: it starts at 0 and goes up by one each time the body runs, so at the end it holds the body's execution count.

```

1  int bodyCount = 0;
2  System.out.println("Ready");
3  int lap = 1;
4  while (lap <= 4)
5  {
6      System.out.println("Lap " + lap);
7      bodyCount++;
8      lap++;
9  }
10 System.out.println("Done");
11 System.out.println("lap is now " + lap);
12 System.out.println("Loop body ran " + bodyCount + " times");

```

Trace it. The condition `lap <= 4` is checked before each pass:

Check	lap	lap <= 4	What happens
1st	1	true	Body runs: prints Lap 1, bodyCount becomes 1
2nd	2	true	Body runs: prints Lap 2, bodyCount becomes 2
3rd	3	true	Body runs: prints Lap 3, bodyCount becomes 3
4th	4	true	Body runs: prints Lap 4, bodyCount becomes 4
5th	5	false	Loop ends; Java moves on to line 10

```

Ready
Lap 1
Lap 2
Lap 3
Lap 4
Done
lap is now 5
Loop body ran 4 times

```

Now read off the execution counts:

Statement	Lines	Execution count
Statements before the loop	1–3	1 each
Each statement in the body	6–8	4 each
The condition <code>lap <= 4</code>	4	5
Statements after the loop	10–12	1 each

The condition is checked **one more time** than the body runs. Four checks come out true and let the body run; the fifth check, when `lap` is 5, comes out false and ends the loop. The printed `lap is now 5` confirms that final check happened with `lap` equal to 5.

The counter variable is a tool for checking your reasoning, not part of the work being measured. When you want to know how often a statement runs, put a counter increment right next to it, in the same block, and print the counter after the loop. If the printed number matches the count you worked out by hand, your reasoning was right.

The same counting works for a `for` loop. Remember from Topic 2.8 that the header's three parts run at different times: the initialization once at the start, the condition before every pass, and the update after every pass.

```
1 int cheers = 0;
2 for (int round = 1; round <= 3; round++)
3 {
4     System.out.println("Hip hip");
5     cheers++;
6 }
7 System.out.println("Hooray!");
8 System.out.println("cheers is " + cheers);
```

```
Hip hip
Hip hip
Hip hip
Hooray!
cheers is 3
```

Part of the loop	When it runs	Execution count
<code>int round = 1</code>	Once, before anything else	1
<code>round <= 3</code>	Before every pass, plus the last	4
Body (lines 4–5)	Once per pass	3
<code>round++</code>	After every pass	3
<code>System.out.println("Hooray!");</code>	Once, after the loop ends	1

When a question asks how many times a statement "is executed," it almost always means a statement in the body, and the answer is the number of passes the loop makes. The rest of this chapter is about working out that number.

WATCH OUT

Trusting the indentation instead of the braces. A loop with no braces repeats only the **one** statement right after its header, no matter how the lines are indented. Here `cheers++;` looks like part of the body, but it is not:

```
1 int cheers = 0;
2 for (int round = 1; round <= 3; round++)
3     System.out.println("Hip hip");
4     cheers++;
5 System.out.println("cheers is " + cheers);
```

```
Hip hip
Hip hip
Hip hip
cheers is 1
```

The loop repeats line 3 three times. Line 4 is outside the loop, so it runs once, after the loop finishes, and the counter ends at 1 instead of 3. Students make this mistake because indentation is how people group lines, but Java ignores indentation completely. To count correctly, find the braces (or, with no braces, the single statement) that make up the body. The fix is to put both statements inside braces, as in the `CheerCount` example above.

PAUSE AND TRY

Question 1 Multiple choice

How many times is the statement `total += step;` executed?

```
1 int total = 0;
2 int step = 1;
3 while (step < 6)
4 {
5     total += step;
6     step++;
7 }
8 System.out.println(total);
```

- A. 5
- B. 6
- C. 4
- D. 1

Question 2 Short answer

A `while` loop's body runs 7 times and then the loop ends normally. How many times was its condition checked? Explain why the answer is not 7.

2.12.2 Reading the Count from a Loop Header

Tracing every pass works, but it gets slow. A loop that runs from 1 to 500 would need a 500-row trace table. You need a faster way to get the number of passes straight from the loop header.

The idea: a counting loop's variable steps through a list of values, and the body runs once for each value that makes the condition true. So the number of passes is the number of values in that

list. Write down the first value, the last value that still passes the condition, and the step size, and count how many values that makes.

For the most common shapes, this turns into two short rules:

Header shape	Values the variable takes	Number of passes
for (int k = a; k < b; k++)	a, a + 1, ..., b - 1	b - a
for (int k = a; k <= b; k++)	a, a + 1, ..., b	b - a + 1

Why the + 1? Counting from a to b **including both ends** gives one more value than the difference. From 3 to 10 the difference is 7, but the values 3, 4, 5, 6, 7, 8, 9, 10 are eight numbers. Think of fence posts: a fence 7 meters long with a post every meter needs 8 posts. With <, the end value b is left out, so the extra one disappears.

KEY IDEA

The number of passes is the number of values the loop variable takes while the condition is true. With $k < b$ counting up by 1 from a, that is $b - a$; with $k \leq b$, it is $b - a + 1$.

WORKED EXAMPLE Three headers, three counts

Before reading the output, use the table above to predict each count.

```

1 int countA = 0;
2 for (int k = 0; k < 10; k++)
3 {
4     countA++;
5 }
6 int countB = 0;
7 for (int k = 1; k <= 10; k++)
8 {
9     countB++;
10 }
11 int countC = 0;
12 for (int k = 3; k <= 10; k++)
13 {
14     countC++;
15 }
16 System.out.println("A: " + countA);
17 System.out.println("B: " + countB);
18 System.out.println("C: " + countC);

```

```

A: 10
B: 10
C: 8

```

Loop	Header	Values	Rule	Passes
A	k = 0; k < 10; k++	0 to 9	10 - 0	10
B	k = 1; k <= 10; k++	1 to 10	10 - 1 + 1	10
C	k = 3; k <= 10; k++	3 to 10	10 - 3 + 1	8

Loops A and B both run 10 times even though their headers look different: A starts one lower and stops one lower. That is why both `for (int k = 0; k < n; k++)` and `for (int k = 1; k <= n; k++)` are common ways to repeat something n times.

The rules above are only for counting by 1. When the update is something else, such as `k += 3` or `timer -= 2`, go back to the basic idea and list the values. The list is usually short, because big steps reach the end fast.

```
1 int countD = 0;
2 for (int k = 0; k < 10; k += 3)
3 {
4     countD++;
5 }
6 int countE = 0;
7 int timer = 9;
8 while (timer > 0)
9 {
10     timer -= 2;
11     countE++;
12 }
13 System.out.println("D: " + countD);
14 System.out.println("E: " + countE);
15 System.out.println("timer is " + timer);
```

```
D: 4
E: 5
timer is -1
```

Loop D's variable takes the values 0, 3, 6, 9. The next value, 12, fails `k < 10`, so the body runs 4 times. Loop E is a while loop counting down: the body runs when `timer` is 9, 7, 5, 3, and 1, which is 5 passes. After the fifth pass `timer` is -1, and the check `-1 > 0` is false. Notice that the loop never landed on exactly 0; it stopped because the condition is `timer > 0`, not because `timer` hit a particular value.

WATCH OUT

Off-by-one counts. In Topic 2.7 you saw that an off-by-one error makes a loop repeat one time too many or one time too few. It usually comes from mixing up the two header shapes. This loop is meant to print 10 stars:

```
1 int stars = 0;
2 for (int k = 0; k <= 10; k++)
3 {
4     System.out.print("*");
5     stars++;
6 }
7 System.out.println();
8 System.out.println(stars + " stars");
```

```
*****  
11 stars
```

The values 0 through 10 make 11 values, because `<=` includes the end. Students make this slip by reading "0 to 10" as ten steps. Fix it by changing the condition to `k < 10` (or by starting at 1 and keeping `k <= 10`):

```
1 int stars = 0;  
2 for (int k = 0; k < 10; k++)  
3 {  
4     System.out.print("*");  
5     stars++;  
6 }  
7 System.out.println();  
8 System.out.println(stars + " stars");
```

```
*****  
10 stars
```

When you count by hand, always check both ends: what is the first value, and what is the **last value that still makes the condition true**?

PAUSE AND TRY

Question 3 Multiple choice

How many times does the body of this loop run?

```
1 int count = 0;  
2 for (int j = 5; j <= 20; j += 5)  
3 {  
4     count++;  
5 }  
6 System.out.println(count);
```

- A. 4
- B. 3
- C. 16
- D. 5

Question 4 Tracing

What does this code print?

```
1 int fuel = 20;
2 int burns = 0;
3 while (fuel >= 4)
4 {
5     fuel -= 3;
6     burns++;
7 }
8 System.out.println("burns: " + burns);
9 System.out.println("fuel: " + fuel);
```

2.12.3 Counts That Depend on Conditions and Data

So far, every statement in a loop body ran on every pass. Real loops are not always that simple. In Topic 2.9 you wrote loops with an `if` inside them, for example to count only the values that meet some condition, and in Topic 2.7 you saw that a `while` loop can stop after an unpredictable number of passes. In these loops, different statements have different counts, and some counts depend on the data.

Statements inside an if

An `if` statement inside a loop is itself a statement in the body, so its condition is **checked** on every pass. The statements inside the `if` run only on the passes where that condition is true. So you need two counts: one for the check and one for the body of the `if`.

WORKED EXAMPLE Counting multiples of 3

The counter `checks` sits in the loop body, next to the `if`. The counter `hits` sits inside the `if`. Predict both before reading on.

```
1 int checks = 0;
2 int hits = 0;
3 for (int num = 1; num <= 30; num++)
4 {
5     checks++;
6     if (num % 3 == 0)
7     {
8         hits++;
9     }
10 }
11 System.out.println("if condition checked: " + checks);
12 System.out.println("if body ran: " + hits);
```

```
if condition checked: 30
if body ran: 10
```

The loop runs for `num` from 1 to 30, which is $30 - 1 + 1 = 30$ passes, so the condition `num % 3 == 0` is checked 30 times. It is true only when `num` is a multiple of 3: 3, 6, 9, ..., 30. That is $30 \div 3 = 10$ values, so `hits++` runs 10 times.

Loops whose length depends on a value

Some loops do not step by a fixed amount. This loop keeps cutting `size` in half with integer division until it reaches 1:

```
1 int size = 64;
2 int halvings = 0;
3 while (size > 1)
4 {
5     size /= 2;
6     halvings++;
7 }
8 System.out.println("From 64: " + halvings + " halvings");
9 size = 100;
10 halvings = 0;
11 while (size > 1)
12 {
13     size /= 2;
14     halvings++;
15 }
16 System.out.println("From 100: " + halvings + " halvings");
```

From 64: 6 halvings

From 100: 6 halvings

Starting from 64, `size` goes $64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. That is six halvings; the check with `size` equal to 1 is false. Starting from 100, integer division drops the remainder each time: $100 \rightarrow 50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1$, again six halvings. Here the count is not "how big is the starting number?" but "how many times can you halve it before reaching 1?" That number grows very slowly, which will matter in 2.12.5.

KEY IDEA

A statement inside an if inside a loop runs only on the passes where the if condition is true, so its count can be smaller than the loop's. When the loop's condition depends on changing data, trace the values to find the count.

WATCH OUT

Assuming every loop body runs at least once. It is tempting to think a loop always does at least one pass before it checks anything. A `while` loop checks its condition **first**:

```
1 int score = 120;
2 int passes = 0;
3 while (score < 100)
4 {
5     score += 15;
6     passes++;
7 }
8 System.out.println("Loop body ran " + passes + " times");
9 System.out.println("score is " + score);
```

```
Loop body ran 0 times  
score is 120
```

score starts at 120, so the very first check, `120 < 100`, is false. The body runs zero times and score never changes. The condition was checked once. A `for` loop behaves the same way: `for (int k = 5; k < 3; k++)` never runs its body. When you count, check the first value against the condition before you count anything.

PAUSE AND TRY

Question 5 Tracing

What does this code print? Also give the number of times the `if` condition `n % 2 == 0` is checked.

```
1 int evens = 0;  
2 int odds = 0;  
3 for (int n = 10; n <= 20; n++)  
4 {  
5     if (n % 2 == 0)  
6     {  
7         evens++;  
8     }  
9     else  
10    {  
11        odds++;  
12    }  
13 }  
14 System.out.println("evens: " + evens);  
15 System.out.println("odds: " + odds);
```

Question 6 Multiple choice

What does this code print?

```
1 int n = 50;  
2 int count = 0;  
3 while (n > 0)  
4 {  
5     n /= 3;  
6     count++;  
7 }  
8 System.out.println(count);
```

- A. 4
- B. 3
- C. 16
- D. 5

2.12.4 Counting in Nested Loops

In Topic 2.11 you put one loop inside another to work through rows and columns, or every pair of values. Nested loops are where execution counts climb quickly, and also where students most often miscount, so they deserve their own method.

The key fact from Topic 2.11: each time the outer loop's body runs, the **entire** inner loop runs, from its initialization to its final false check. So if the outer loop makes 3 passes and the inner loop makes 4 passes each time, the inner body runs $4 + 4 + 4 = 3 \times 4 = 12$ times. A theater with 3 rows of 4 seats has 12 seats, not 7.

KEY IDEA

When the inner loop always makes the same number of passes, the inner body's count is (outer passes) $\times$ (inner passes). Statements in the outer body but outside the inner loop run once per outer pass.

WORKED EXAMPLE A seating chart

Two counters measure two different places: `seatsDone` is inside the inner loop, and `rowsDone` is in the outer body, after the inner loop.

```

1  int seatsDone = 0;
2  int rowsDone = 0;
3  for (int row = 1; row <= 3; row++)
4  {
5      for (int seat = 1; seat <= 4; seat++)
6      {
7          System.out.print("*");
8          seatsDone++;
9      }
10     System.out.println();
11     rowsDone++;
12 }
13 System.out.println("Seats printed: " + seatsDone);
14 System.out.println("Rows finished: " + rowsDone);

```

```

****
****
****
Seats printed: 12
Rows finished: 3

```

Statement	Where it sits	Count
<code>System.out.print("*");</code>	Inner body	$3 \times 4 = 12$
<code>System.out.println();</code>	Outer body, after the inner loop	3
<code>rowsDone++;</code>	Outer body, after the inner loop	3
<code>int seat = 1</code> (inner init)	Runs each time the inner loop starts	3

The inner loop's initialization runs 3 times, once per row, because the inner loop starts over for every outer pass. That restart is the reason `seat` goes back to 1 on every row.

When the inner loop's length changes

Sometimes the inner loop's header uses the outer loop's variable, so the inner loop makes a different number of passes each time. Multiplying no longer works. Instead, find the inner count for each outer pass and add them up.

```

1  int stars = 0;
2  for (int row = 1; row <= 4; row++)
3  {
4      for (int col = 1; col <= row; col++)
5      {
6          System.out.print("*");
7          stars++;
8      }
9      System.out.println();
10 }
11 System.out.println("Stars: " + stars);

```

```

*
**
***
****
Stars: 10

```

The inner condition is `col <= row`, so the inner loop runs `row` times:

row	Inner values of col	Inner passes
1	1	1
2	1, 2	2
3	1, 2, 3	3
4	1, 2, 3, 4	4
Total		10

The total is $1 + 2 + 3 + 4 = 10$, and the picture shows it: each printed row is as long as that row's inner count. If you multiplied 4×4 you would get 16, the count for a full square of stars, which overcounts this triangle.

WATCH OUT

Adding instead of multiplying, or counting from the wrong place. For the seating chart, a common wrong answer is $3 + 4 = 7$. Adding the two loop counts would be right only if the loops ran one after the other. Nested means "4 inner passes for **each** outer pass," which is multiplication.

The second slip is miscounting a statement that sits in the outer body. `System.out.println();` comes right after the inner loop's closing brace, so it is easy to lump it in with the inner body and say it runs 12 times. It runs 3 times. Before you count a statement in nested loops, find which pair of braces it is directly inside.

PAUSE AND TRY**Question 7** Multiple choice

What does this code print?

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

- A. 10
- B. 25
- C. 15
- D. 5

Question 8 Tracing

What does this code print?

```
1 int inner = 0;
2 int outer = 0;
3 for (int i = 0; i < 10; i += 2)
4 {
5     for (int j = 3; j > 0; j--)
6     {
7         inner++;
8     }
9     outer++;
10 }
11 System.out.println("inner: " + inner);
12 System.out.println("outer: " + outer);
```

2.12.5 Comparing Run Times Informally

Counting statements is useful because it lets you compare two ways of solving the same problem without running them. If two code segments give the same result, the one that executes fewer statements does less work and, informally, has a shorter run time. A second, even more useful question is how the work **grows** when the input gets bigger. A program that is fine for 10 values might be painfully slow for a million.

WORKED EXAMPLE Two ways to add the even numbers from 2 to 100

Version A checks every number from 1 to 100 and adds the even ones. Version B visits only the even numbers by counting by 2.

```

1  int sumA = 0;
2  int loopsA = 0;
3  for (int num = 1; num <= 100; num++)
4  {
5      loopsA++;
6      if (num % 2 == 0)
7      {
8          sumA += num;
9      }
10 }
11 int sumB = 0;
12 int loopsB = 0;
13 for (int num = 2; num <= 100; num += 2)
14 {
15     loopsB++;
16     sumB += num;
17 }
18 System.out.println("Version A: sum " + sumA + ", loop ran " + loopsA + " times");
19 System.out.println("Version B: sum " + sumB + ", loop ran " + loopsB + " times");

```

```

Version A: sum 2550, loop ran 100 times
Version B: sum 2550, loop ran 50 times

```

Both versions compute 2550, so they are equally correct. Now count the work:

Version	Loop passes	if checks	Additions to the sum
A	100	100	50
B	50	0	50

Version A makes twice as many passes and checks a condition on each one. Half of those passes, for the odd numbers, do no useful work at all. Version B skips them by choosing a better update, `num += 2`. It does about half the work for the same answer.

Comparing at one size is a start. The bigger difference shows up when the size changes. Let `n` stand for the size of the problem: how many values to process, or how long a string is. The program below measures three loop shapes from this chapter at `n = 10`, `20`, and `40`: a single loop that runs `n` times, a nested loop that runs `n` times inside `n` times, and the halving loop from 2.12.3.

```

1  for (int n = 10; n <= 40; n *= 2)
2  {
3      int single = 0;
4      for (int i = 0; i < n; i++)
5      {
6          single++;
7      }
8      int nested = 0;
9      for (int i = 0; i < n; i++)
10     {
11         for (int j = 0; j < n; j++)
12         {
13             nested++;
14         }
15     }
16     int halving = 0;
17     int size = n;
18     while (size > 1)
19     {
20         size /= 2;
21         halving++;
22     }
23     System.out.println("n = " + n + ": single " + single + ", nested " + nested + ",
24     halving " + halving);
25 }

```

```

n = 10: single 10, nested 100, halving 3
n = 20: single 20, nested 400, halving 4
n = 40: single 40, nested 1600, halving 5

```

Look at what happens each time `n` doubles:

Loop shape	n = 10	n = 20	n = 40	When n doubles, the count...
Single loop, <code>n</code> passes	10	20	40	doubles
Nested, <code>n × n</code> passes	100	400	1600	multiplies by 4
Halving loop	3	4	5	goes up by just 1

Doubling `n` doubles both loop counts in the nested version, and $2 \times 2 = 4$, so its work quadruples. The halving loop barely notices: doubling the start only adds one more halving. For small inputs every version finishes almost instantly, but as `n` grows the nested loop falls further and further behind. This is the heart of informal run-time analysis: count the executions, and ask how the count changes as the input grows.

KEY IDEA

To compare code segments informally, compare their statement execution counts for the same input, and compare how those counts grow as the input gets bigger. A nested loop's count grows much faster than a single loop's.

WATCH OUT

Judging speed by how long the code looks. Fewer lines of code does not mean less work. This segment is only a few lines long:

```
1 int count = 0;
2 for (int i = 0; i < 100; i++)
3 {
4     for (int j = 0; j < 100; j++)
5     {
6         count++;
7     }
8 }
9 System.out.println(count);
```

10000

The `count++;` statement runs $100 \times 100 = 10,000$ times. Twenty separate lines with no loop would each run once, for 20 executions in total. Students judge by length because that is what they see on the page, but the run time depends on how many times statements **execute**, not on how many statements are written. Look for loops, and especially nested loops, before judging how much work code does.

PAUSE AND TRY**Question 9** Short answer

Segment P has a single loop whose body runs n times. Segment Q has a nested loop whose inner body runs $n \times n$ times. When n is 50, P's body runs 50 times and Q's inner body runs 2500 times. Give both counts when n is 100, and explain which segment's work grows faster.

Question 10 Multiple choice

Loop 1 is `for (int k = 1; k <= n; k++)`. Loop 2 starts with size equal to n and runs `while (size > 1)`, doing `size /= 2;` each pass. When n is 1000, how many times does each loop body run?

- A. Loop 1: 1000; Loop 2: 9
- B. Loop 1: 1000; Loop 2: 500
- C. Loop 1: 999; Loop 2: 9
- D. Loop 1: 1000; Loop 2: 1000

Putting it together

This program looks at every pair of different numbers from 1 to 4 and reports the pairs whose sum is even. It has a nested loop whose inner length changes, an `if` inside the inner loop, and three counters. Predict all three counts before reading the trace.

WORKED EXAMPLE Pairs with an even sum

```

1  int outerRuns = 0;
2  int innerRuns = 0;
3  int evenPairs = 0;
4  for (int first = 1; first <= 4; first++)
5  {
6      outerRuns++;
7      for (int second = first + 1; second <= 4; second++)
8      {
9          innerRuns++;
10         if ((first + second) % 2 == 0)
11         {
12             System.out.println(first + " + " + second + " is even");
13             evenPairs++;
14         }
15     }
16 }
17 System.out.println("Outer body: " + outerRuns);
18 System.out.println("Inner body: " + innerRuns);
19 System.out.println("Even pairs: " + evenPairs);

```

The inner loop starts at `first + 1`, so each pair is visited once and a number is never paired with itself. Trace the outer passes:

first	Values of second	Inner passes	Pairs with an even sum
1	2, 3, 4	3	1 + 3
2	3, 4	2	2 + 4
3	4	1	none
4	none (5 fails <= 4)	0	none

```

1 + 3 is even
2 + 4 is even
Outer body: 4
Inner body: 6
Even pairs: 2

```

- **Outer body:** `first` goes from 1 to 4, which is $4 - 1 + 1 = 4$ passes, even though the last one does no inner work.
- **Inner body:** the inner length changes, so add: $3 + 2 + 1 + 0 = 6$. The `if` condition is in the inner body, so it is also checked 6 times.
- **Inside the `if`:** only 2 of those 6 checks are true, so the `println` and `evenPairs++` run twice each.

Now the run-time question: what if the program checked the numbers from 1 to 8 instead? The inner counts become $7 + 6 + 5 + 4 + 3 + 2 + 1 + 0 = 28$. Doubling the range from 4 to 8 made the inner body run more than four times as often (6 to 28), the same fast growth you saw for nested loops in 2.12.5.

Vocabulary

Term	Definition	Example
statement execution count	The number of times a statement runs while the program runs	The body of <code>for (int k = 0; k < 10; k++)</code> has a count of 10
informal run-time analysis	Estimating and comparing how much work code does by counting statement executions, usually by tracing	A nested 10×10 loop does 100 inner passes; a single loop does 10
run time	How long a program takes to run	Version B in 2.12.5 does half the passes of Version A
counter variable	A variable that starts at 0 and goes up by 1 each time a statement runs, used here to check an execution count	<code>count++</code> ; placed next to the statement being counted

Chapter summary

- A statement execution count is how many times a statement runs. Statements outside loops run once; statements in a loop body run once per pass; a loop condition is checked one more time than the body runs.
- To find the number of passes, count the values the loop variable takes while the condition is true. Counting up by 1 from a , `k < b` gives $b - a$ passes and `k <= b` gives $b - a + 1$. For other steps, list the values.
- Statements inside an `if` run only on the passes where its condition is true, a loop's count can depend on changing data, and a loop whose condition starts false runs zero times.
- In nested loops, multiply when the inner loop's length is fixed, and add the inner counts pass by pass when it changes with the outer loop.
- Compare code segments by how many statements they execute for the same input and by how that count grows as the input gets bigger; a counter variable lets you check any count you work out by hand.

Review questions

Question 11 Multiple choice

How many times is `#` printed?

```
1 for (int j = 1; j <= 5; j += 2)
2 {
3     for (int k = 0; k < 3; k++)
4     {
5         System.out.print("#");
6     }
7     System.out.println();
8 }
```

- A. 9
- B. 15
- C. 6
- D. 12

Question 12 Tracing

What does this code print? Show how `num` changes.

```
1 int num = 37;
2 int steps = 0;
3 int ones = 0;
4 while (num > 0)
5 {
6     if (num % 2 == 1)
7     {
8         ones++;
9     }
10    num /= 2;
11    steps++;
12 }
13 System.out.println("steps: " + steps);
14 System.out.println("ones: " + ones);
```

Question 13 Debugging

A student wants the counter to report how many times the inner loop body runs. They expect Inner body ran 15 times, but the program prints Inner body ran 3 times. Find the error and correct it.

```
1 int count = 0;
2 for (int row = 0; row < 3; row++)
3 {
4     for (int col = 0; col < 5; col++)
5     {
6         System.out.print("o");
7     }
8     count++;
9     System.out.println();
10 }
11 System.out.println("Inner body ran " + count + " times");
```

Question 14 Multiple choice

This code checks a word for a letter that appears twice in a row. How many times is the `if` condition evaluated?

```
1 String word = "letter";
2 int matches = 0;
3 for (int i = 0; i < word.length() - 1; i++)
4 {
5     if (word.substring(i, i + 1).equals(word.substring(i + 1, i + 2)))
6     {
7         matches++;
8     }
9 }
10 System.out.println("matches: " + matches);
```

- A. 5
- B. 6
- C. 1
- D. 4

Question 15 Short answer

This code searches a string for the first "a", once in "rhythm and blues" and once in "jazz". It prints `rhythm and blues steps: 7` and `jazz steps: 1`. The first string is about four times as long as the second. Explain why the step counts are 7 and 1, and why the length of the string alone does not tell you how many times the loop body runs.

```
1 String text = "rhythm and blues";
2 int index = 0;
3 int steps = 0;
4 while (index < text.length() && !text.substring(index, index + 1).equals("a"))
5 {
6     index++;
7     steps++;
8 }
9 System.out.println(text + " steps: " + steps);
10 text = "jazz";
11 index = 0;
12 steps = 0;
13 while (index < text.length() && !text.substring(index, index + 1).equals("a"))
14 {
15     index++;
16     steps++;
17 }
18 System.out.println(text + " steps: " + steps);
```

Question 16 Multiple choice

In this segment, the inner loop starts at `j = i`, so it gets shorter as `i` grows:

```
1 for (int i = 0; i < n; i++)
2 {
3     for (int j = i; j < n; j++)
4     {
5         count++;
6     }
7 }
```

When n is 10, `count++` runs 55 times. About how many times does it run when n is 20?

- A. 210
- B. 110
- C. 65
- D. 400

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can find how many times a statement runs, and I know a loop condition is checked one more time than the body runs. Reread: [2.12.1 Counting How Many Times a Statement Runs](#)
- ☐ I can work out the number of passes from a loop header, including loops that count by steps other than 1. Reread: [2.12.2 Reading the Count from a Loop Header](#)
- ☐ I can count statements inside an if in a loop, loops that depend on their data, and loops that run zero times. Reread: [2.12.3 Counts That Depend on Conditions and Data](#)
- ☐ I can count executions in nested loops by multiplying or by adding the inner counts. Reread: [2.12.4 Counting in Nested Loops](#)
- ☐ I can compare two code segments by their execution counts and describe how the counts grow as the input grows. Reread: [2.12.5 Comparing Run Times Informally](#)

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

A. `step` takes the values 1, 2, 3, 4, and 5 while `step < 6` is true, so the body runs 5 times, and `total += step;` is in the body. When `step` becomes 6, the condition is false and the loop ends. (The program prints 15, which is $1 + 2 + 3 + 4 + 5$.)

- **B** is wrong: 6 is how many times the **condition** is checked, one more than the number of passes. The sixth check is false, so the body does not run then.
- **C** is wrong: it misses one pass, probably by stopping at 4 as if the condition were `step < 5`. Check the last value that makes the condition true: `5 < 6` is true.
- **D** is wrong: the statement appears once in the code, but it is inside the loop body, so it runs once per pass.

Reread 2.12.1 *Counting How Many Times a Statement Runs* if you chose differently.

Question 2

Model answer: 8 times. The condition is checked before every pass, so 7 checks came out true and let the body run 7 times. The loop can only stop when a check comes out false, so there is one more check at the end, the 8th, which is false and ends the loop.

A complete answer must:

- give 8,
- say the condition is checked before each pass, and
- explain that the final, false check is the extra one that ends the loop.

Reread the trace in 2.12.1 *Counting How Many Times a Statement Runs* if your answer differed.

Question 3

A. `j` takes the values 5, 10, 15, and 20. All four make `j <= 20` true. The next value, 25, makes it false. Four values means four passes, and the code prints 4.

- **B** is wrong: it leaves out 20, as if the condition were `j < 20`. With `<=`, the end value is included.
- **C** is wrong: 16 is $20 - 5 + 1$, the rule for counting **by 1**. This loop counts by 5, so list the values instead.
- **D** is wrong: it counts one value too many, perhaps by also counting 0 or 25. Neither one passes: the loop starts at 5, and 25 fails the condition.

Reread 2.12.2 *Reading the Count from a Loop Header* if you chose differently.

Question 4

```
burns: 6
fuel: 2
```

Check	<code>fuel</code>	<code>fuel >= 4</code>	After the body: <code>fuel</code> , <code>burns</code>
1	20	true	17, 1
2	17	true	14, 2

3	14	true	11, 3
4	11	true	8, 4
5	8	true	5, 5
6	5	true	2, 6
7	2	false	loop ends

The values that pass the condition are 20, 17, 14, 11, 8, and 5: six passes. Notice that `fuel` is 5 at the sixth check, which still passes `fuel >= 4`, so one more burn happens and `fuel` ends at 2.

Reread 2.12.2 *Reading the Count from a Loop Header* if your output differed.

Question 5

```
evens: 6
odds: 5
```

The loop runs for n from 10 to 20 including both ends: $20 - 10 + 1 = 11$ passes, so the `if` condition is checked **11** times. Every pass runs exactly one of the two branches. The even values are 10, 12, 14, 16, 18, 20 (six of them), and the odd values are 11, 13, 15, 17, 19 (five). Check: $6 + 5 = 11$, one branch per pass.

Reread 2.12.3 *Counts That Depend on Conditions and Data* if your answer differed.

Question 6

A. Integer division by 3 drops the remainder each time: n goes $50 \rightarrow 16 \rightarrow 5 \rightarrow 1 \rightarrow 0$. The checks with 50, 16, 5, and 1 are all true (each is greater than 0), so the body runs 4 times. The fifth check, with n equal to 0, is false.

- **B** is wrong: it stops when n reaches 1, as if the condition were $n > 1$. One is still greater than zero, so there is one more pass, and $1 / 3$ is 0.
- **C** is wrong: 16 is the value of n after the first pass, not the number of passes.
- **D** is wrong: it counts the final check, with n equal to 0, as a pass. That check is false, so the body does not run.

Reread 2.12.3 *Counts That Depend on Conditions and Data* if you chose differently.

Question 7

A. The inner condition is $b < a$, so the inner loop runs a times on each outer pass. a takes the values 0, 1, 2, 3, 4, giving inner counts $0 + 1 + 2 + 3 + 4 = 10$. When a is 0, the check $0 < 0$ is false right away, so that pass adds nothing.

- **B** is wrong: $5 \times 5 = 25$ assumes the inner loop always runs 5 times. Its length depends on a , so add the counts instead of multiplying.
- **C** is wrong: 15 is $1 + 2 + 3 + 4 + 5$, which treats the inner loop as running $a + 1$ times (as if it were $b \leq a$). With $<$, the inner loop runs a times.
- **D** is wrong: 5 is the number of outer passes, not the number of times the inner body runs.

Reread 2.12.4 *Counting in Nested Loops* if you chose differently.

Question 8

```
inner: 15
outer: 5
```

The outer loop's `i` takes the values 0, 2, 4, 6, 8 (10 fails `i < 10`), so it makes **5** passes. The inner loop counts down: `j` is 3, 2, 1 (0 fails `j > 0`), so it makes **3** passes every time. The inner length never changes, so multiply: $5 \times 3 = 15$. `outer++` is in the outer body after the inner loop, so it runs once per outer pass: 5.

Reread 2.12.4 *Counting in Nested Loops* if your output differed.

Question 9

Model answer: When `n` is 100, P's body runs 100 times and Q's inner body runs $100 \times 100 = 10,000$ times. Doubling `n` doubled P's count (50 to 100) but quadrupled Q's (2500 to 10,000), because both of Q's loops got twice as long and $2 \times 2 = 4$. Q's work grows much faster.

A complete answer must:

- give 100 for P and 10,000 for Q,
- say that P's count doubles while Q's multiplies by 4, and
- explain that Q's growth comes from both nested loops getting longer.

Reread 2.12.5 *Comparing Run Times Informally* if your answer differed.

Question 10

A. Loop 1 runs for `k` from 1 to 1000 including both ends: $1000 - 1 + 1 = 1000$ passes. Loop 2 halves with integer division: $1000 \rightarrow 500 \rightarrow 250 \rightarrow 125 \rightarrow 62 \rightarrow 31 \rightarrow 15 \rightarrow 7 \rightarrow 3 \rightarrow 1$. That is 9 halvings before `size > 1` is false.

- **B** is wrong: 500 is the value of `size` after one pass, not the number of passes. Halving shrinks the value very fast, so the count stays small.
- **C** is wrong: with `<=`, the end value 1000 is included, so Loop 1 makes 1000 passes, not 999.
- **D** is wrong: Loop 2 does not step by 1; each pass cuts `size` in half.

Reread 2.12.5 *Comparing Run Times Informally* and the halving loop in 2.12.3 if you chose differently.

Question 11

A. The outer loop's `j` takes the values 1, 3, 5 (7 fails `j <= 5`), so it makes 3 passes. The inner loop runs for `k` = 0, 1, 2, which is 3 passes every time. The inner length is fixed, so multiply: $3 \times 3 = 9$. The output is three lines of `###`.

- **B** is wrong: it treats the outer loop as running 5 times, as if it counted by 1. Counting by 2 from 1 to 5 gives only 1, 3, 5.
- **C** is wrong: $3 + 3 = 6$ adds the two loop counts. The loops are nested, so the inner loop's 3 passes happen for **each** of the 3 outer passes: multiply.
- **D** is wrong: $4 \times 3 = 12$ counts one outer pass too many, perhaps by including `j = 7`, which fails the condition.

Reread 2.12.4 *Counting in Nested Loops* and the step-size rule in 2.12.2 if you chose differently.

Question 12

steps: 6
ones: 3

Check	num	num % 2 == 1?	ones after	num after /= 2	steps after
1	37	yes	1	18	1
2	18	no	1	9	2
3	9	yes	2	4	3
4	4	no	2	2	4
5	2	no	2	1	5
6	1	yes	3	0	6
7	0	loop ends			

The loop body runs 6 times, so `steps++` runs 6 times. The `if` condition is checked 6 times too, but `ones++` runs only on the 3 passes where `num` is odd (37, 9, and 1). The loop does not stop at 1: `1 > 0` is true, so there is one more pass, and `1 / 2` is 0.

Reread 2.12.3 *Counts That Depend on Conditions and Data* if your output differed.

Question 13

The counter is in the wrong block. `count++`; on line 8 is after the inner loop's closing brace, so it is in the **outer** body. It runs once per outer pass, 3 times, so it measures the outer body, not the inner one. The program compiles and runs, so only the wrong number reveals the bug.

Corrected code: move `count++`; inside the inner loop's braces, next to the statement being counted.

```

1  int count = 0;
2  for (int row = 0; row < 3; row++)
3  {
4      for (int col = 0; col < 5; col++)
5      {
6          System.out.print("o");
7          count++;
8      }
9      System.out.println();
10 }
11 System.out.println("Inner body ran " + count + " times");

```

Now it runs once per inner pass, $3 \times 5 = 15$ times, and the program prints the same three rows of `ooooo` followed by `Inner body ran 15 times`.

Reread 2.12.4 *Counting in Nested Loops* and the counter-variable advice in 2.12.1 if you missed it.

Question 14

A. "letter" has length 6, so the condition is `i < 5` and `i` takes the values 0 through 4: $5 - 0 = 5$ passes. The `if` is in the body, so its condition is evaluated 5 times. (Only one check is true, the `t` at index 2 next to the `t` at index 3, so the code prints `matches: 1`.)

- **B** is wrong: it uses the full length, 6. The loop stops one early on purpose, because it compares each letter with the **next** one, and the last letter has no next letter.
- **C** is wrong: 1 is how many times the condition is **true**, which is how many times `matches++` runs, not how many times the condition is checked.
- **D** is wrong: it subtracts 1 twice. `i < word.length() - 1` with `i` starting at 0 gives $5 - 0 = 5$ passes, not 4.

Reread 2.12.2 *Reading the Count from a Loop Header* and the `if` counts in 2.12.3 if you chose differently.

Question 15

Model answer: The loop keeps going only while the character at `index` is not "a", so it stops as soon as it reaches the first "a". In "rhythm and blues" the first "a" is at index 7, after the six letters of `rhythm` and a space, so the body runs 7 times. In "jazz" the "a" is at index 1, so the body runs once. The count depends on **where** the "a" is, not just on how long the string is. A string with no "a" at all would make the loop run its full length, until `index < text.length()` becomes false.

A complete answer must:

- say the loop stops at the first "a", so its count equals the index of that "a",
- give the positions (index 7 and index 1), and
- explain that the count depends on the data, with the full length as the most it can be.

Reread 2.12.3 *Counts That Depend on Conditions and Data* if your answer differed.

Question 16

A. The inner loop runs `n - i` times, so the total is $n + (n - 1) + \dots + 1$. For `n = 10` that is $10 + 9 + \dots + 1 = 55$; for `n = 20` it is $20 + 19 + \dots + 1 = 210$. Doubling `n` multiplied the count by almost 4, the same fast growth as any nested loop where both loops get longer.

- **B** is wrong: doubling the count is how a **single** loop grows. Here both loops get longer when `n` doubles.
- **C** is wrong: adding 10 to the count ignores that every outer pass gets longer, and there are also twice as many outer passes.
- **D** is wrong: $20 \times 20 = 400$ would be right only if the inner loop always ran the full 20 times. It starts at `j = i`, so it gets shorter on each pass, which cuts the total roughly in half.

Reread 2.12.5 *Comparing Run Times Informally* and the changing inner loop in 2.12.4 if you chose differently.