

2.9 Implementing Selection and Iteration Algorithms

Self-Study Guide · About 60 minutes

By the end you will be able to

- Trace and explain the standard loop patterns for adding up a sum and computing an average.
- Count how many values meet a condition by putting an if statement inside a loop.
- Use the remainder operator to decide whether one integer divides another evenly.
- Pull an integer apart into its digits with % 10 and / 10 inside a while loop.
- Find the smallest or largest value in a series and explain how the starting value affects the result.
- Combine these patterns to predict the output of a longer program and to find and fix bugs in it.

What to remember from earlier

- int division keeps only the whole-number part, and % gives the remainder. ([Topic 1.3: Expressions and Output](#), [1.3.6 Integer Division](#))
- A cast such as (double) total turns an int into a double. ([Topic 1.5: Casting and Range of Variables](#), [1.5.3 Casting an int to a double](#))
- Integer.MAX_VALUE and Integer.MIN_VALUE are the largest and smallest int values. ([Topic 1.5: Casting and Range of Variables](#), [1.5.5 The Range of int Values](#))
- An if statement runs its body only when its condition is true, and else runs when it is false. ([Topic 2.3: if Statements](#), [2.3.3 Two-Way Selection with if-else](#))
- A while loop checks its condition before every pass; a for loop puts initialization, condition, and update in one header. ([Topic 2.8: for Loops](#), [2.8.1 The Three Parts of a for Loop Header](#))

2.9.1 Adding Up a Sum

You now have every building block you need to make a program decide (if statements) and repeat (while and for loops). This topic is about putting those blocks together to solve problems that come up again and again: What is the total of these prices? What is the average score? How many days were hot? What was the coldest night? Programmers solve these so often that each has a well-known shape. A **standard algorithm** is a step-by-step solution to a common problem that programmers reuse, adjusting the details each time. Once you recognize the shape, you can read, trace, and fix code that uses it quickly, and you can combine shapes to solve new problems.

The first and most common problem is adding up a series of numbers. Think about how you add a stack of receipts by hand. You do not look at all of them at once. You start at zero, look at one receipt, add its amount to what you have so far, then move to the next receipt and add again. The number in your head is a **running total**: the sum of every value you have looked at so far.

A program does the same thing with a variable. An **accumulator** is a variable that collects a result a little at a time as a loop runs; when the result is a sum, the accumulator holds the running total. Every sum algorithm has the same three parts:

Part	Code	Why it is there
Start the total at zero	<code>int total = 0;</code>	Before you have looked at anything, the sum is zero. Done before loop.
Add one value on each pass	<code>total += value;</code>	Each pass makes the running total include one more value. Done inside .
Use the total after the loop	<code>println(total)</code>	Only when the loop has finished does the total include every value.

The order matters. The total must be set to zero once, before the loop starts, so that the additions on later passes are not thrown away. The `+=` from Topic 1.6 does the adding: `total += value;` means `total = total + value;`, so each pass builds on what the earlier passes stored.

KEY IDEA

To add up a series of values, start an accumulator at 0 before the loop, add one value to it with `+=` on each pass, and use the finished total after the loop ends.

WORKED EXAMPLE Adding the numbers 1 through 5

Here the values to add come from the loop control variable itself, which counts 1, 2, 3, 4, 5. Watch `total` grow by the current value of `number` on each pass.

```

1 int total = 0;
2 for (int number = 1; number <= 5; number++)
3 {
4     total += number;
5 }
6 System.out.println(total);

```

Line 1 runs once. Then the for loop from Topic 2.8 takes over: it sets `number` to 1, checks `number <= 5`, runs the body (line 4), updates `number`, and checks again.

Pass	number	total before line 4	total after line 4
1	1	0	1
2	2	1	3
3	3	3	6
4	4	6	10
5	5	10	15

After pass 5, `number` becomes 6, the condition `6 <= 5` is false, and the loop ends. Line 6 prints:

```
15
```

Read the "before" column: each pass starts from exactly what the pass before it stored. That is why the total ends up as $1 + 2 + 3 + 4 + 5$.

Now change where the values come from. Without lists of data (those come in Unit 4), a program usually gets a series of values by reading them one at a time. This example reads them with a `Scanner`. The first number in the input says how many prices follow, so a for loop can run exactly that many times. A keyboard `Scanner` would work the same way; this guide builds the `Scanner` from a string so the output is always the same.

WORKED EXAMPLE Totaling a receipt

ReceiptTotal.java

```
1 import java.util.Scanner;
2
3 public class ReceiptTotal
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("4 3.25 1.50 2.75 4.00");
8         int itemCount = input.nextInt();
9         double total = 0.0;
10        for (int item = 1; item <= itemCount; item++)
11        {
12            double price = input.nextDouble();
13            total += price;
14        }
15        System.out.print("Total: ");
16        System.out.println(total);
17    }
18 }
```

Line 8 reads 4, the number of items. Line 9 starts the accumulator at 0.0; it is a `double` because prices have decimal parts. On each pass, line 12 reads the next price into `price`, and line 13 adds it to `total`.

item	price read on line 12	total after line 13
1	3.25	3.25
2	1.5	4.75
3	2.75	7.5
4	4.0	11.5

When `item` becomes 5, the condition `5 <= 4` is false. Lines 15 and 16 print:

```
Total: 11.5
```

Compare this with the first example. The loop and the accumulator have the same shape; only the source of each value changed, from the loop control variable to `input.nextDouble()`. Notice too that `price` is replaced on every pass, while `total` keeps growing. A variable that holds "the current value" and a variable that holds "the result so far" play very different roles.

WATCH OUT

Resetting the total inside the loop. Students sometimes put the "start at zero" step inside the loop, thinking each pass needs a fresh start:

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

This prints 5, not 15. On every pass, line 4 wipes out the running total and line 5 adds only the current number. After the last pass, `total` holds just `0 + 5`. Java gives no error, because setting a variable to zero is legal; the program simply answers a different question. The fix is to set the total to zero once, before the loop, as in the first worked example.

PAUSE AND TRY**Question 1** Tracing

What does this code print? Make a trace table with columns for `k` and `total`.

```
1 int total = 0;
2 for (int k = 3; k <= 12; k += 3)
3 {
4     total += k;
5 }
6 System.out.println(total);
```

Question 2 Debugging

This code is supposed to print the sum of the whole numbers from 1 through 10, which is 55. It prints 45 instead. Find the error and correct it.

```
1 int total = 0;
2 for (int number = 1; number < 10; number++)
3 {
4     total += number;
5 }
6 System.out.println(total);
```

2.9.2 Computing an Average

A teacher does not usually want the total of a class's quiz scores; she wants the average. A weather report gives the average rainfall, not the sum. To find an **average** (also called the mean), you add up the values and divide by how many values there are. So an average algorithm is the sum algorithm from 2.9.1 plus one division after the loop.

The division is where things go wrong, because of a rule from Topic 1.3: when both numbers are `int`, `/` throws away the decimal part. A sum of whole-number scores and a count of scores are both `int`, so `sum / count` can lose information. The fix, from Topic 1.5, is to cast one of them to `double` **before** dividing, so the division is done with decimals.

KEY IDEA

An average is the sum divided by the number of values. When both are `int`, write `(double) sum / count` so the division keeps its decimal part.

WORKED EXAMPLE Average rainfall over five days

The first input value is the number of days. Each pass of the loop reads one day's rainfall and adds it straight into the accumulator. Line 12 passes the value from `nextInt()` directly to `+=`; no separate variable is needed when the value is used only once.

RainfallAverage.java

```

1  import java.util.Scanner;
2
3  public class RainfallAverage
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("5 12 7 0 9 4");
8          int days = input.nextInt();
9          int totalRain = 0;
10         for (int day = 1; day <= days; day++)
11         {
12             totalRain += input.nextInt();
13         }
14         double average = (double) totalRain / days;
15         System.out.print("Average rainfall: ");
16         System.out.println(average);
17     }
18 }
```

day	Value read	totalRain after line 12
1	12	12
2	7	19
3	0	19
4	9	28
5	4	32

After the loop, line 14 computes the average. The cast applies only to `totalRain`, the value right after it, and it happens before the division. So Java works out `32.0 / 5`, a `double` divided by an `int`, which is `6.4`:

```
Average rainfall: 6.4
```

Day 3 had no rain, but it still counts as a day. The value `0` adds nothing to the total, yet the division is still by `5`.

Where the cast goes matters. This snippet tries three ways with the same total and count:

```
1 int totalRain = 32;
2 int days = 5;
3 System.out.println(totalRain / days);
4 System.out.println((double) (totalRain / days));
5 System.out.println((double) totalRain / days);
```

```
6
6.0
6.4
```

Expression	What Java does	Result
<code>totalRain / days</code>	<code>int</code> divided by <code>int</code> : decimal part dropped.	<code>6</code>
<code>(double) (totalRain / days)</code>	Parentheses first: <code>32 / 5</code> is <code>6</code> ; then cast.	<code>6.0</code>
<code>(double) totalRain / days</code>	Cast first: <code>32.0 / 5</code> is a <code>double</code> division.	<code>6.4</code>

The middle one is the trap. It looks like it uses a cast, but by the time the cast happens, the integer division has already thrown away the `.4`. Casting `6` to a `double` gives `6.0`, not `6.4`.

So far the program knew how many values were coming. Often it does not: a teacher types scores until she is done. You met the usual way to handle this in Topic 2.7: a **sentinel value**, a special value that is not real data and that means "stop reading." Quiz scores are never negative, so `-1` works as a sentinel. Because the program does not know in advance how many values there will be, it uses a while loop (Topic 2.7) and a second accumulator. A **counter** is an accumulator that adds one each time something happens; here it adds one for each real value read.

WORKED EXAMPLE Average quiz score with a sentinel

QuizAverage.java

```

1  import java.util.Scanner;
2
3  public class QuizAverage
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("88 92 75 90 -1");
8          int sum = 0;
9          int count = 0;
10         int score = input.nextInt();
11         while (score != -1)
12         {
13             sum += score;
14             count++;
15             score = input.nextInt();
16         }
17         if (count > 0)
18         {
19             double average = (double) sum / count;
20             System.out.print("Average: ");
21             System.out.println(average);
22         }
23         else
24         {
25             System.out.println("No scores");
26         }
27     }
28 }

```

The reading pattern has a special shape. Line 10 reads the first value **before** the loop, so the condition on line 11 has something to check. Inside the loop, line 15 reads the next value **last**, so the condition checks each new value before it is added.

Step	score	sum	count
Line 10 reads	88	0	0
Pass 1 (lines 13–15)	92	88	1
Pass 2	75	180	2
Pass 3	90	255	3
Pass 4	-1	345	4

The `score` column shows the value read at the end of each pass. After pass 4, `score` is `-1`, so `score != -1` is false and the loop ends. The `-1` is never added, which is the whole point of a sentinel. Line 19 computes `345.0 / 4`:

Average: 86.25

The if on line 17 guards the division. If the very first value were `-1`, the loop body would never run (Topic 2.7), `count` would still be `0`, and there would be nothing to average. With input `"-1"` this program prints `No scores` instead.

WATCH OUT

Dividing when there are no values. Without the `count > 0` check, an empty input means dividing by zero. Here is a version that uses `int` division and no guard:

AverageNoGuard.java

```
1 import java.util.Scanner;
2
3 public class AverageNoGuard
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("-1");
8         int sum = 0;
9         int count = 0;
10        int score = input.nextInt();
11        while (score != -1)
12        {
13            sum += score;
14            count++;
15            score = input.nextInt();
16        }
17        int average = sum / count;
18        System.out.println(average);
19    }
20 }
```

The loop never runs, so line 17 tries `0 / 0` and the program crashes:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at AverageNoGuard.main(AverageNoGuard.java:17)
```

(The exact wording can vary between Java versions.) With the `(double)` cast it would instead be a `double` divided by zero, a case this course leaves out. Either way there is no meaningful average, so the fix is the same: check that `count > 0` before dividing, as `QuizAverage.java` does.

PAUSE AND TRY

Question 3 Multiple choice

The `int` variables `total` and `count` hold `17` and `4`. Which expression evaluates to `4.25`?

- A. `total / count`
- B. `(double) (total / count)`
- C. `(double) total / count`
- D. `(int) ((double) total / count)`

Question 4 Tracing

What does this program print? Trace `score`, `sum`, and `count`.

SentinelZeroScore.java

```
1  import java.util.Scanner;
2
3  public class SentinelZeroScore
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("10 0 5 -1");
8          int sum = 0;
9          int count = 0;
10         int score = input.nextInt();
11         while (score != -1)
12         {
13             sum += score;
14             count++;
15             score = input.nextInt();
16         }
17         System.out.println((double) sum / count);
18     }
19 }
```

2.9.3 Counting Values That Meet a Condition

In 2.9.2 you counted every value. Often you need to count only some of them. How many days this week were over 90 degrees? How many students scored in the B range? How many numbers from 1 to 100 are even? Each question asks for a **frequency**: the number of times a particular condition is true.

The plan is the counter from the average algorithm, which added one for every value, with an if statement wrapped around the `count++`. Start it at 0 before the loop. Inside the loop, test the current value; only when the condition is true, add one.

```

1 int count = 0;
2 // loop over the values:
3     if (condition about the current value)
4     {
5         count++;
6     }
7 // after the loop, count holds the answer

```

Why 0 and not 1? Before the loop has checked anything, no value has met the condition. Starting at 1 would claim one match before seeing any data.

KEY IDEA

To count how often a condition is met, start a counter at 0 before the loop, and inside the loop add one only when an if statement finds the condition true.

WORKED EXAMPLE Counting hot days

The input holds seven daily high temperatures. A day is "hot" when its high is above 90.

HotDays.java

```

1 import java.util.Scanner;
2
3 public class HotDays
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("7 88 93 91 79 95 90 84");
8         int days = input.nextInt();
9         int hotCount = 0;
10        for (int day = 1; day <= days; day++)
11        {
12            int temp = input.nextInt();
13            if (temp > 90)
14            {
15                hotCount++;
16            }
17        }
18        System.out.print("Hot days: ");
19        System.out.println(hotCount);
20    }
21 }

```

day	temp	temp > 90?	hotCount after the if
1	88	false	0
2	93	true	1
3	91	true	2
4	79	false	2
5	95	true	3
6	90	false	3

7	84	false	3
---	----	-------	---

Hot days: 3

Look at day 6. A high of exactly 90 is not "above 90," so $90 > 90$ is false and the counter does not change. If the question had been "at least 90," the condition would be `temp >= 90` and the answer would be 4. The counting pattern never changes; the condition decides what is counted, so read it carefully at the boundary.

The condition can be anything you learned in Topics 2.2 through 2.6, including compound conditions. Here the question is how many scores fall from 70 to 89, including both ends.

WORKED EXAMPLE Counting scores in a range

ScoreBand.java

```

1  import java.util.Scanner;
2
3  public class ScoreBand
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("6 72 95 89 64 70 90");
8          int students = input.nextInt();
9          int inBand = 0;
10         for (int student = 1; student <= students; student++)
11         {
12             int score = input.nextInt();
13             if (score >= 70 && score <= 89)
14             {
15                 inBand++;
16             }
17         }
18         System.out.print("Scores from 70 to 89: ");
19         System.out.println(inBand);
20     }
21 }
```

A score is in the range only when it is at least 70 **and** at most 89, so line 13 joins two comparisons with `&&` (Topic 2.5).

score	score >= 70	score <= 89	Counted?
72	true	true	yes (1)
95	true	false	no
89	true	true	yes (2)
64	false	(skipped)	no
70	true	true	yes (3)
90	true	false	no

Scores from 70 to 89: 3

For 64, the first comparison is false, so && short-circuits and never checks the second (Topic 2.5); the whole condition is already false. Both 70 and 89 count because the comparisons use `>=` and `<=`.

WATCH OUT

Leaving out the braces. As in Topic 2.3, without braces an if controls only the one statement right after it, no matter how the lines are indented. Here a student wanted to print each hot temperature and count it:

HotDaysNoBraces.java

```
1 import java.util.Scanner;
2
3 public class HotDaysNoBraces
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("7 88 93 91 79 95 90 84");
8         int days = input.nextInt();
9         int hotCount = 0;
10        for (int day = 1; day <= days; day++)
11        {
12            int temp = input.nextInt();
13            if (temp > 90)
14                System.out.println(temp);
15                hotCount++;
16        }
17        System.out.print("Hot days: ");
18        System.out.println(hotCount);
19    }
20 }
```

```
93
91
95
Hot days: 7
```

The prints are right, but the count is 7, the number of days, not the number of hot days. Only line 14 belongs to the if. Line 15 is indented to look as if it belongs, but Java ignores indentation, so `hotCount++` runs on every pass. The fix is to put braces around both statements so that both are inside the if:

```
1 if (temp > 90)
2 {
3     System.out.println(temp);
4     hotCount++;
5 }
```

PAUSE AND TRY**Question 5** Tracing

What two numbers does this program print?

FreezeCount.java

```
1 import java.util.Scanner;
2
3 public class FreezeCount
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("5 32 15 40 32 28");
8         int nights = input.nextInt();
9         int freezing = 0;
10        int aboveFreezing = 0;
11        for (int night = 1; night <= nights; night++)
12        {
13            int temp = input.nextInt();
14            if (temp <= 32)
15            {
16                freezing++;
17            }
18            else
19            {
20                aboveFreezing++;
21            }
22        }
23        System.out.println(freezing);
24        System.out.println(aboveFreezing);
25    }
26 }
```

Question 6 Multiple choice

In `ScoreBand.java`, the scores were 72, 95, 89, 64, 70, and 90. Which condition on line 13 would count exactly the scores **outside** the range 70 to 89, giving a count of 3?

- A. `score < 70 && score > 89`
- B. `score < 70 || score > 89`
- C. `score <= 70 || score >= 89`
- D. `!(score > 70 && score < 89)`

2.9.4 Testing Whether One Integer Divides Another

Can 84 marbles be shared equally among 7 friends with none left over? Is this year a multiple of 4? Is a number even? All of these ask whether one integer is **evenly divisible** by another: dividing leaves a remainder of zero.

You already have the tool. From Topic 1.3, `a % b` gives the remainder when `a` is divided by `b`. If the remainder is `0`, then `b` goes into `a` exactly. So the test "is `a` evenly divisible by `b`?" is the Boolean expression

```
1 a % b == 0
```

and "is `a` not evenly divisible by `b`?" is `a % b != 0`. In this course, the number being divided is never negative and the number you divide by is always positive, so these are the only cases you need.

KEY IDEA

`a % b == 0` is true exactly when `a` is evenly divisible by `b`.

WORKED EXAMPLE Sharing marbles

```
1 int marbles = 84;
2 int friends = 7;
3 System.out.println(marbles % friends);
4 if (marbles % friends == 0)
5 {
6     System.out.println("7 friends: shares evenly");
7 }
8 friends = 5;
9 System.out.println(marbles % friends);
10 if (marbles % friends == 0)
11 {
12     System.out.println("5 friends: shares evenly");
13 }
14 else
15 {
16     System.out.println("5 friends: some left over");
17 }
```

```
0
7 friends: shares evenly
4
5 friends: some left over
```

Line 3 prints `84 % 7`. Seven goes into 84 exactly twelve times ($7 \times 12 = 84$), so the remainder is `0` and the condition on line 4 is true. After line 8 changes `friends` to 5, line 9 prints `84 % 5`: five goes into 84 sixteen times (80), leaving 4. The condition on line 10 is false, so the else runs.

The most common divisibility test is checking whether a number is even, which means "divisible by 2." Every whole number leaves a remainder of either `0` or `1` when divided by 2:

```

1 for (int n = 6; n <= 9; n++)
2 {
3     System.out.print(n);
4     System.out.print(" % 2 is ");
5     System.out.println(n % 2);
6 }

```

```

6 % 2 is 0
7 % 2 is 1
8 % 2 is 0
9 % 2 is 1

```

So `n % 2 == 0` means `n` is even, and `n % 2 != 0` means `n` is odd.

A divisibility test becomes much more powerful inside a loop. Combine it with the counting pattern from 2.9.3, and you can find every number that divides another.

WORKED EXAMPLE Finding the divisors of 12

A **divisor** of a number is a whole number that divides it evenly. This loop tries every candidate `d` from 1 up to the number itself and keeps the ones that leave no remainder.

```

1 int number = 12;
2 int divisorCount = 0;
3 for (int d = 1; d <= number; d++)
4 {
5     if (number % d == 0)
6     {
7         System.out.print(d);
8         System.out.print(" ");
9         divisorCount++;
10    }
11 }
12 System.out.println();
13 System.out.print("Divisors: ");
14 System.out.println(divisorCount);

```

d	12 % d	Divisor?	divisorCount
1	0	yes	1
2	0	yes	2
3	0	yes	3
4	0	yes	4
5	2	no	4
6	0	yes	5
7–11	not 0	no	5
12	0	yes	6

Each divisor is printed with a space after it on one line, and line 12 then ends that line:

```
1 2 3 4 6 12
Divisors: 6
```

Change only `number` to 13 and the same code prints `1 13` and `Divisors: 2`. A number greater than 1 whose only divisors are 1 and itself is called a prime number, so this one loop can also tell you whether a number is prime: it is prime exactly when the count is 2.

WATCH OUT

Writing the two numbers in the wrong order. "84 is divisible by 7" puts 84 first, so the test is `84 % 7 == 0`. It is easy to swap them:

```
1 int marbles = 84;
2 int friends = 7;
3 System.out.println(friends % marbles);
4 if (friends % marbles == 0)
5 {
6     System.out.println("Shares evenly");
7 }
8 else
9 {
10    System.out.println("Some left over");
11 }
```

```
7
Some left over
```

`friends % marbles` is `7 % 84`. Since 84 does not go into 7 even once, the whole 7 is left over, so the remainder is 7 and the test fails. Read `a % b == 0` as "a divided by b leaves nothing," with the bigger whole (the thing being shared) on the left. The fix is `marbles % friends == 0`.

PAUSE AND TRY

Question 7 Debugging

This code is supposed to count the even numbers from 1 to 10, which should be 5. It prints 1. Find the error and correct it.

```
1 int evenCount = 0;
2 for (int n = 1; n <= 10; n++)
3 {
4     if (n / 2 == 0)
5     {
6         evenCount++;
7     }
8 }
9 System.out.println(evenCount);
```

Question 8 Multiple choice

What does this code print?

```

1 for (int n = 1; n <= 30; n++)
2 {
3     if (n % 4 == 0 && n % 6 == 0)
4     {
5         System.out.println(n);
6     }
7 }
```

- A. 12 and 24, each on its own line
- B. 4, 6, 8, 12, 16, 18, 20, 24, 28, and 30, each on its own line
- C. 24 only
- D. Nothing

2.9.5 Pulling Apart the Digits of an Integer

A credit card number has a check digit that catches typos. A puzzle asks for the sum of the digits of a number. A program reports how many digits a number has. Each needs the individual **digits** of an `int`, but an `int` is a single number, not a row of separate digits. How do you get at them?

The answer uses the two kinds of integer division from Topic 1.3, with 10:

- `number % 10` is the **last digit** (the ones digit). Dividing by 10 leaves the ones digit as the remainder: `4072 % 10` is 2.
- `number / 10` **drops the last digit**. Integer division by 10 throws away the ones digit: `4072 / 10` is 407.

Why 10? Our numbers are written in base ten: each place is worth ten times the place to its right. Dividing by 10 shifts every digit one place to the right, and whatever falls off the end is the remainder. You can combine the two to reach any digit:

```

1 int number = 4072;
2 System.out.println(number % 10);
3 System.out.println(number / 10);
4 System.out.println(number / 10 % 10);
5 System.out.println(number / 100 % 10);
```

Line	Expression	Work	Result
2	<code>number % 10</code>	ones digit of 4072	2
3	<code>number / 10</code>	drop the 2	407
4	<code>number / 10 % 10</code>	407 % 10 (tens digit)	7
5	<code>number / 100 % 10</code>	40 % 10 (hundreds)	0

Line 4 has no parentheses but still works left to right: `/` and `%` have the same precedence, so `number / 10` happens first and `% 10` is applied to its result.

To visit **every** digit without knowing how many there are, put the two steps in a while loop: take the last digit, then drop it, and repeat until nothing is left. When the number reaches 0, every digit has been removed.

KEY IDEA

`number % 10` gives the last digit and `number / 10` removes it. Repeating both in a `while (number > 0)` loop visits every digit, from right to left.

WORKED EXAMPLE Printing the digits of 4072

```
1 int number = 4072;
2 while (number > 0)
3 {
4     int digit = number % 10;
5     System.out.println(digit);
6     number = number / 10;
7 }
```

Pass	number at start	digit (line 4)	number after line 6
1	4072	2	407
2	407	7	40
3	40	0	4
4	4	4	0

```
2
7
0
4
```

After pass 4, `number` is 0, so `0 > 0` is false and the loop ends. The digits come out **right to left**, because `% 10` always takes the rightmost one. Notice that the zero in the middle is found like any other digit. Line 6 is what moves the loop toward ending; if it were missing, `number` would stay 4072 forever and `number > 0` would always be true, an infinite loop (Topic 2.7).

Because the loop destroys the number as it goes, a program that still needs the original value copies it first. This next example does that and uses two accumulators at once: a sum of the digits and a count of the digits.

WORKED EXAMPLE Adding and counting the digits of 9035

```

1 int number = 9035;
2 int remaining = number;
3 int digitSum = 0;
4 int digitCount = 0;
5 while (remaining > 0)
6 {
7     digitSum += remaining % 10;
8     digitCount++;
9     remaining /= 10;
10 }
11 System.out.print(number);
12 System.out.print(" has ");
13 System.out.print(digitCount);
14 System.out.print(" digits that add up to ");
15 System.out.println(digitSum);

```

remaining at start	remaining % 10	digitSum	digitCount
9035	5	5	1
903	3	8	2
90	0	8	3
9	9	17	4

9035 has 4 digits that add up to 17

The loop changes only `remaining`, so `number` still holds 9035 when lines 11 to 15 print it. Line 9 uses `/=` from Topic 1.6; `remaining /= 10;` means the same as `remaining = remaining / 10;`.

These digit algorithms assume the number is not negative. That is the usual situation (counts, years, ID numbers), and the remainder operator with negative numbers is outside this course.

WATCH OUT

Forgetting the number 0. How many digits does 0 have? One. But the digit-counting loop says zero:

```

1 int number = 0;
2 int digitCount = 0;
3 while (number > 0)
4 {
5     digitCount++;
6     number /= 10;
7 }
8 System.out.println(digitCount);

```

This prints 0. The condition `0 > 0` is false the very first time, so the body never runs (Topic 2.7). The loop is correct for every positive number but misses this edge case. One fix is to handle 0 separately with an if before the loop:

```
1 int number = 0;
2 int digitCount = 0;
3 if (number == 0)
4 {
5     digitCount = 1;
6 }
7 while (number > 0)
8 {
9     digitCount++;
10    number /= 10;
11 }
12 System.out.println(digitCount);
```

This prints 1. For any positive number, the if is skipped and the loop counts as before.

PAUSE AND TRY

Question 9 Tracing

What does this code print?

```
1 int n = 615;
2 while (n > 0)
3 {
4     System.out.print(n % 10);
5     n /= 10;
6 }
7 System.out.println();
```

Question 10 Multiple choice

The int variable n holds 4729. Which expression evaluates to the hundreds digit, 7?

- A. `n % 100`
- B. `n / 100`
- C. `n / 100 % 10`
- D. `n % 10 / 100`

2.9.6 Finding the Minimum or Maximum

Which player had the highest score? What was the coldest night of the week? These ask for the **maximum** (largest value) or the **minimum** (smallest value) in a series.

Think about how you would find the tallest person in a line if you could see only one person at a time. You would remember the tallest person so far. You look at the first person: so far, they are the tallest. For each next person, you compare: if this one is taller than the one you remember, you remember this one instead. When you reach the end, the person you remember is the tallest of all.

A program does exactly that with a variable that holds the **best so far**. Two decisions make up the algorithm:

1. **Where to start.** Before the loop, the best-so-far variable needs a starting value. The safest choice is the first value itself, since it is a real piece of data. Another choice is a value that every real value is guaranteed to beat: `Integer.MAX_VALUE` when looking for a minimum (every `int` is at most that), or `Integer.MIN_VALUE` when looking for a maximum.
2. **How to update.** Inside the loop, compare the current value with the best so far. For a maximum, `if (value > best);` for a minimum, `if (value < best)`. Only when the comparison is true, replace the best so far.

KEY IDEA

To find a maximum or minimum, keep a "best so far" variable. Start it at the first value (or at a value every real value will beat), and inside the loop replace it whenever the current value is larger (for a maximum) or smaller (for a minimum).

WORKED EXAMPLE The highest score

HighestScore.java

```
1  import java.util.Scanner;
2
3  public class HighestScore
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("5 71 88 64 93 80");
8          int players = input.nextInt();
9          int highest = input.nextInt();
10         for (int player = 2; player <= players; player++)
11         {
12             int score = input.nextInt();
13             if (score > highest)
14             {
15                 highest = score;
16             }
17         }
18         System.out.print("Highest: ");
19         System.out.println(highest);
20     }
21 }
```

Line 9 reads the first score straight into `highest`. Because the first value has already been used, the loop on line 10 starts `player` at 2 and runs for the remaining four players.

player	score	score > highest?	highest after the if
(line 9)	71		71
2	88	true	88
3	64	false	88
4	93	true	93
5	80	false	93

Highest: 93

If the loop had started `player` at 1, it would try to read six scores when only five remain, an off-by-one error. Starting at 2 matches the fact that one score was read before the loop.

Now vary three things: look for the **minimum**, start at `Integer.MAX_VALUE` instead of the first value, and also remember **where** the minimum happened. Recording the position is a small addition: whenever the best so far changes, store the day too.

WORKED EXAMPLE The coldest day and when it happened

ColdestDay.java

```

1  import java.util.Scanner;
2
3  public class ColdestDay
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("5 -3 4 -8 2 -5");
8          int days = input.nextInt();
9          int coldest = Integer.MAX_VALUE;
10         int coldestDay = 0;
11         for (int day = 1; day <= days; day++)
12         {
13             int temp = input.nextInt();
14             if (temp < coldest)
15             {
16                 coldest = temp;
17                 coldestDay = day;
18             }
19         }
20         System.out.print("Coldest: ");
21         System.out.print(coldest);
22         System.out.print(" on day ");
23         System.out.println(coldestDay);
24     }
25 }
```

day	temp	temp < coldest?	coldest, coldestDay
start			Integer.MAX_VALUE, 0
1	-3	true	-3, 1
2	4	false	-3, 1

3	-8	true	-8, 3
4	2	false	-8, 3
5	-5	false	-8, 3

Coldest: -8 on day 3

On day 1, $-3 < 2147483647$ is certainly true, so the first real temperature always replaces the starting value. That is why `Integer.MAX_VALUE` is safe for a minimum: no `int` is larger. This version reads every value inside the loop, so `day` starts at 1. Lines 16 and 17 are both inside the braces, so the day is recorded only when the temperature is a new low.

WATCH OUT

Starting the best so far at 0. Zero feels like a natural starting point, the way it is for a sum, but here it can be wrong. Look at a cold week where every high temperature is below zero:

`MaxStartZero.java`

```

1  import java.util.Scanner;
2
3  public class MaxStartZero
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("4 -6 -2 -9 -4");
8          int days = input.nextInt();
9          int warmest = 0;
10         for (int day = 1; day <= days; day++)
11         {
12             int temp = input.nextInt();
13             if (temp > warmest)
14             {
15                 warmest = temp;
16             }
17         }
18         System.out.println(warmest);
19     }
20 }
```

This prints 0, but no day had a high of 0. Every temperature (-6, -2, -9, -4) is less than 0, so `temp > warmest` is never true and the made-up starting value is never replaced. Starting at 0 works only if you know some value will beat it. The fix is to start with the first real value, as `HighestScore.java` does:

MaxStartFirst.java

```
1 import java.util.Scanner;
2
3 public class MaxStartFirst
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("4 -6 -2 -9 -4");
8         int days = input.nextInt();
9         int warmest = input.nextInt();
10        for (int day = 2; day <= days; day++)
11        {
12            int temp = input.nextInt();
13            if (temp > warmest)
14            {
15                warmest = temp;
16            }
17        }
18        System.out.println(warmest);
19    }
20 }
```

This prints `-2`, the warmest of the four.

PAUSE AND TRY**Question 11** Tracing

What does this program print?

MinUpdates.java

```
1 import java.util.Scanner;
2
3 public class MinUpdates
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("4 12 9 15 9");
8         int count = input.nextInt();
9         int lowest = input.nextInt();
10        int updates = 0;
11        for (int i = 2; i <= count; i++)
12        {
13            int value = input.nextInt();
14            if (value < lowest)
15            {
16                lowest = value;
17                updates++;
18            }
19        }
20        System.out.print("Lowest: ");
21        System.out.println(lowest);
22        System.out.print("Updates: ");
23        System.out.println(updates);
24    }
25 }
```

Question 12 Short answer

ColdestDay.java started coldest at Integer.MAX_VALUE. Would the program still work if it started coldest at 0 instead? Explain, using a week of summer temperatures such as 18, 25, and 21 as an example.

Putting it together

Real programs rarely use just one of these algorithms. They run several in the same loop, each with its own variable. This program reads a week of step counts from a fitness tracker and reports the total, the average, how many days met a 10,000-step goal, and the best day. Comments (Topic 1.8) label the job of each variable.

WORKED EXAMPLE A weekly step report

StepReport.java

```

1  import java.util.Scanner;
2
3  public class StepReport
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("7 8200 10450 6100 12000 9800 10450 6000");
8          int days = input.nextInt();
9
10         int total = 0;           // sum of all step counts
11         int goalDays = 0;       // days with at least 10000 steps
12         int best = 0;           // most steps in one day so far
13         int bestDay = 0;        // the day that happened
14
15         for (int day = 1; day <= days; day++)
16         {
17             int steps = input.nextInt();
18             total += steps;
19             if (steps >= 10000)
20             {
21                 goalDays++;
22             }
23             if (day == 1 || steps > best)
24             {
25                 best = steps;
26                 bestDay = day;
27             }
28         }
29
30         System.out.print("Total: ");
31         System.out.println(total);
32         System.out.print("Average: ");
33         System.out.println((double) total / days);
34         System.out.print("Goal days: ");
35         System.out.println(goalDays);
36         System.out.print("Best: ");
37         System.out.print(best);
38         System.out.print(" on day ");
39         System.out.println(bestDay);
40     }
41 }
```

Four algorithms share one loop:

Algorithm	Variable(s)	Lines
Sum	total	10, 18

Count with a test	goalDays	11, 19–22
Maximum (and when)	best, bestDay	12–13, 23–27
Average	total and days	33, after loop

Line 23 is a new twist on the maximum. Instead of reading the first value before the loop, it says "replace the best if this is day 1 **or** the steps beat the best so far." On day 1 the left side is true, so `||` short-circuits (Topic 2.5) and the first value always becomes the best. After that, `day == 1` is false and only `steps > best` matters.

day	steps	total	goalDays	best, bestDay
1	8200	8200	0	8200, 1
2	10450	18650	1	10450, 2
3	6100	24750	1	10450, 2
4	12000	36750	2	12000, 4
5	9800	46550	2	12000, 4
6	10450	57000	3	12000, 4
7	6000	63000	3	12000, 4

After the loop, line 33 computes `(double) 63000 / 7`, which is `9000.0`:

```
Total: 63000
Average: 9000.0
Goal days: 3
Best: 12000 on day 4
```

Day 5 (9800) is close to the goal but is not counted, because `9800 >= 10000` is false. Each variable did its one job on every pass, and none of them interfered with the others.

The algorithms also combine in less obvious ways. There is an old shortcut for checking whether a number is divisible by 9: add up its digits, and if that sum is divisible by 9, so is the number. This snippet uses the digit algorithm to build a sum, then two divisibility tests to compare the shortcut with the direct check.

WORKED EXAMPLE The divisible-by-9 shortcut

```

1 int number = 7848;
2 int remaining = number;
3 int digitSum = 0;
4 while (remaining > 0)
5 {
6     digitSum += remaining % 10;
7     remaining /= 10;
8 }
9 System.out.print("Digit sum: ");
10 System.out.println(digitSum);
11 if (digitSum % 9 == 0)
12 {
13     System.out.println("Digit sum is a multiple of 9");
14 }
15 if (number % 9 == 0)
16 {
17     System.out.println("Number is a multiple of 9");
18 }

```

The loop visits 8, 4, 8, 7, so `digitSum` is $8 + 4 + 8 + 7 = 27$. Then $27 \% 9$ is 0, and $7848 \% 9$ is also 0 ($7848 = 9 \times 872$), so both ifs run:

```

Digit sum: 27
Digit sum is a multiple of 9
Number is a multiple of 9

```

Change `number` to 7849 and the digit sum becomes 28. Both $28 \% 9$ and $7849 \% 9$ are 1, so neither message prints. The shortcut and the direct test agree, as they always do for 9. A program like this is an original algorithm: no single standard pattern solves it, but it is built entirely from the ones in this chapter.

Vocabulary

Term	Definition	Example
standard algorithm	A well-known step-by-step solution to a common problem that programmers reuse and adapt	Summing values in a loop
running total	The sum of every value looked at so far	<code>total</code> after pass 3
accumulator	A variable that collects a result a little at a time as a loop runs	<code>total += price;</code>
average	The sum of the values divided by how many values there are	<code>(double) sum / count</code>
sentinel value	A special value that is not real data and signals the end of the input	-1 after quiz scores
frequency	The number of times a condition is true	3 hot days

counter	An accumulator that adds one each time something happens	<code>hotCount++;</code>
evenly divisible	Divides with a remainder of zero	<code>84 % 7 == 0</code>
divisor	A whole number that divides another evenly	4 is a divisor of 12
digit	One of the single figures 0–9 that make up a written number; <code>% 10</code> gives the last one	4072 % 10 is 2
maximum	The largest value in a series	93 in 71, 88, 64, 93, 80
minimum	The smallest value in a series	-8 in -3, 4, -8, 2, -5
best so far	The variable in a minimum or maximum algorithm holding the smallest or largest value seen so far	highest in HighestScore

Chapter summary

- A sum uses an accumulator that starts at 0 before the loop, gains one value per pass with `+=`, and is used after the loop. Resetting it inside the loop throws away the running total.
- An average is the sum divided by the count. Write `(double) sum / count` so the division keeps its decimal part, and check that the count is greater than zero first. A sentinel value lets a while loop read an unknown number of values.
- Counting how often a condition is met puts `count++` inside an if inside the loop. The condition, including its boundary (`>` versus `>=`), decides what is counted.
- `a % b == 0` tests whether `a` is evenly divisible by `b`; `n % 2 == 0` tests for an even number. Inside a loop, it can find every divisor of a number.
- `% 10` gives the last digit of a nonnegative integer and `/ 10` removes it; a `while (number > 0)` loop repeating both visits every digit from right to left.
- A minimum or maximum keeps a best-so-far variable that starts at the first value (or a value every real value will beat) and is replaced whenever the current value is smaller or larger.

Review questions

Question 13 Multiple choice

What does this code print?

```
1 int sum = 0;
2 int count = 0;
3 for (int k = 1; k <= 10; k++)
4 {
5     if (k % 3 == 0)
6     {
7         sum += k;
8     }
9     else
10    {
11        count++;
12    }
13 }
14 System.out.println(sum);
15 System.out.println(count);
```

- A. 18 and then 7
- B. 18 and then 10
- C. 55 and then 7
- D. 18 and then 3

Question 14 Debugging

This code is supposed to count the numbers from 1 to 50 that are divisible by **both** 2 and 3. The correct answer is 8 (6, 12, 18, 24, 30, 36, 42, 48), but it prints 33. Find the error and correct it.

```
1 int count = 0;
2 for (int n = 1; n <= 50; n++)
3 {
4     if (n % 2 == 0 || n % 3 == 0)
5     {
6         count++;
7     }
8 }
9 System.out.println(count);
```

Question 15 Tracing

What does this code print? Trace `number` and `zeros`.

```
1 int number = 70300;
2 int zeros = 0;
3 while (number > 0)
4 {
5     if (number % 10 == 0)
6     {
7         zeros++;
8     }
9     number /= 10;
10 }
11 System.out.print("Zeros: ");
12 System.out.println(zeros);
```

Question 16 Multiple choice

A store's daily sales are read by this program. Sales of 55 happened on both day 2 and day 4. What does it print?

TieLastDay.java

```
1 import java.util.Scanner;
2
3 public class TieLastDay
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("5 40 55 30 55 20");
8         int days = input.nextInt();
9         int best = input.nextInt();
10        int bestDay = 1;
11        for (int day = 2; day <= days; day++)
12        {
13            int sales = input.nextInt();
14            if (sales >= best)
15            {
16                best = sales;
17                bestDay = day;
18            }
19        }
20        System.out.print("Best day: ");
21        System.out.println(bestDay);
22    }
23 }
```

- A. Best day: 2
- B. Best day: 4

- C. Best day: 55
- D. Best day: 5

Question 17 Short answer

A program reads quiz scores until it reaches a sentinel, then prints their average. A student suggests using 0 as the sentinel instead of -1. Explain why that is a bad choice, and say what makes a good sentinel.

Question 18 Tracing

What three lines does this code print? (Starting `largest` at 0 is safe here: why?)

```
1 int number = 2468;
2 int sum = 0;
3 int largest = 0;
4 while (number > 0)
5 {
6     int digit = number % 10;
7     sum += digit;
8     if (digit > largest)
9     {
10         largest = digit;
11     }
12     number /= 10;
13 }
14 System.out.println(sum);
15 System.out.println(largest);
16 System.out.println(sum % largest == 0);
```

Reflection

☐ I can trace a sum with an accumulator and explain why it starts at 0 before the loop. Reread:

2.9.1 Adding Up a Sum

☐ I can compute an average without losing the decimal part, and explain what a sentinel value is for. Reread: 2.9.2 Computing an Average

☐ I can trace a loop that counts how many values meet a condition, including at the boundary.

Reread: 2.9.3 Counting Values That Meet a Condition

☐ I can use % to test whether one integer is evenly divisible by another. Reread: 2.9.4 Testing

Whether One Integer Divides Another

☐ I can use % 10 and / 10 in a loop to visit every digit of an integer. Reread: 2.9.5 Pulling Apart the

Digits of an Integer

☐ I can find a minimum or maximum and explain how the starting value can change the answer.

Reread: 2.9.6 Finding the Minimum or Maximum

☐ I can trace a program that runs several of these algorithms in the same loop. Reread: Putting it together