

2.7 while Loops

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Recognize when a task needs a loop because a step must repeat, possibly an unknown number of times.
- Write and read a while loop, naming its condition and its body.
- Trace a while loop with a trace table, including the final check that ends it.
- Explain why a while loop can run zero times, and what starting values a loop needs to work as intended.
- Spot infinite loops and off-by-one errors, explain their cause, and fix them.

What to remember from earlier

- Repetition is a process that repeats until a desired outcome is reached. ([Topic 2.1: Algorithms with Selection and Repetition](#), [2.1.3 Repetition: Repeating Until the Goal Is Reached](#))
- Relational operators compare values and give a boolean. ([Topic 2.2: Boolean Expressions](#))
- An if statement runs its block once when its condition is true and skips it otherwise. ([Topic 2.3: if Statements](#), [2.3.1 Running Code Only When a Condition Is True](#))
- Use equals, not ==, to compare the contents of two String objects. ([Topic 2.6: Comparing Boolean Expressions](#), [2.6.6 Comparing Objects with equals](#))
- `x += 3` adds 3 to x, and `x++` adds 1. ([Topic 1.6: Compound Assignment Operators](#))

2.7.1 Repeating Code with a while Loop

Suppose a running app must announce each lap of a five-lap race. With only the tools from Unit 1, you would write one print statement per lap:

```
1 System.out.println("Lap 1");
2 System.out.println("Lap 2");
3 System.out.println("Lap 3");
4 System.out.println("Lap 4");
5 System.out.println("Lap 5");
```

It prints Lap 1 through Lap 5, one per line. That works for five laps. Now imagine a marathon training run of 40 laps, or a race where the runner chooses the number of laps before starting. Copying the line 40 times is tedious and easy to get wrong, and if the number of laps is not known when you write the program, no fixed number of copies can be right.

In Topic 2.1 you met **repetition** in everyday algorithms: "keep stirring until the sauce thickens," "keep walking laps until you have done five." The step is written once, and a decision about whether to do it again is made each time. Java needs the same thing: a way to write a step once and have it run again and again as long as some condition holds.

Repeating a segment of code is called **iteration**. A statement that does this is an **iteration statement**, usually called a **loop**. A loop changes the **flow of control**, the order in which Java runs

statements. Normally Java runs statements top to bottom, each one once. A loop sends Java back up to run the same statements again, as long as a boolean expression is `true`.

Java's simplest loop is the **while loop**. Here is the five-lap program written with one:

```
1 int lap = 1;
2 while (lap <= 5)
3 {
4     System.out.println("Lap " + lap);
5     lap++;
6 }
7 System.out.println("Race over");
```

Take the loop apart piece by piece:

Piece	What it is
<code>while</code>	The keyword that starts the loop. Read it as "as long as."
<code>(lap <= 5)</code>	The loop condition : a boolean expression in parentheses. The body runs again each time it is <code>true</code> .
<code>{ ... }</code>	The braces mark the loop body , the statements that repeat. Here that is lines 4 and 5.
<code>lap++;</code>	The update. It changes the variable the condition tests, so the condition can eventually become <code>false</code> .
line 7	The first statement after the loop. Java goes here once the condition is <code>false</code> . It is not part of the body.

Read line 2 aloud as "while `lap` is at most 5, do the body." The shape is the same as an `if` statement from Topic 2.3: keyword, condition in parentheses, block in braces. The difference is what happens after the block. An `if` runs its block at most once and moves on. A `while` loop goes back to the condition and asks again.

One run through the loop body is called an **iteration**. This loop does five iterations, one for each value of `lap` from 1 to 5. Look at the work the variable `lap` does: it starts at 1 before the loop (line 1), the condition tests it (line 2), the body prints it and adds 1 to it (lines 4 and 5). That start, test, and update are what make the loop stop at the right time.

KEY IDEA

A while loop repeats its body as long as its condition is true. The condition is a boolean expression, and something in the body must change it so that it eventually becomes false.

WORKED EXAMPLE Tracing the lap loop

A **trace table** follows a loop one step at a time. Give it a column for each variable, a column for the condition's value, and a column for what is printed. Read across one row per step.

Step	lap	lap <= 5?	Printed
1	1	true	Lap 1
2	2	true	Lap 2
3	3	true	Lap 3

4	4	true	Lap 4
5	5	true	Lap 5
6	6	false	Race over

In each of the first five rows, Java checks the condition with the current value of `lap`, finds it true, prints the lap, and then `lap++` adds 1. In row 6, `lap` is 6 and `6 <= 5` is false, so Java skips the body and goes to line 7. The output is:

```
Lap 1
Lap 2
Lap 3
Lap 4
Lap 5
Race over
```

To run 40 laps instead of 5, you would change only the `5` in the condition. The body is still written once.

Now vary two things: count down instead of up, and use `>` instead of `<=`.

WORKED EXAMPLE A countdown

```
1 int seconds = 3;
2 while (seconds > 0)
3 {
4     System.out.println(seconds);
5     seconds--;
6 }
7 System.out.println("Liftoff!");
```

Step	seconds	seconds > 0?	Printed
1	3	true	3
2	2	true	2
3	1	true	1
4	0	false	Liftoff!

```
3
2
1
Liftoff!
```

This time the update is `seconds--`, which moves the value **down** toward the point where `seconds > 0` stops being true. The update must move the variable toward making the condition false. Counting up works with a condition like `lap <= 5`; counting down works with one like `seconds > 0`. Notice that `0` is never printed: when `seconds` reaches 0, the condition is false before the print can run.

WATCH OUT

Writing if when you need while. The two look alike, so it is easy to type the wrong keyword:

```
1 int lap = 1;
2 if (lap <= 5)
3 {
4     System.out.println("Lap " + lap);
5     lap++;
6 }
7 System.out.println("Race over");
```

This compiles and prints only:

```
Lap 1
Race over
```

An `if` checks its condition once. It runs the block (printing Lap 1 and making lap 2) and then moves on to line 7; it never goes back to ask whether `lap <= 5` is still true. If a step must happen again and again, you need `while`. Use `if` for a decision made once.

WATCH OUT

The condition must be a boolean. Writing `=` (assignment) where you mean a comparison does not compile:

```
1 int lap = 1;
2 while (lap = 5)
3 {
4     System.out.println("Lap " + lap);
5     lap++;
6 }
```

```
error: incompatible types: int cannot be converted to boolean
```

`lap = 5` stores 5 in `lap`, and its value is the `int` 5, not `true` or `false`. A loop condition must be a boolean expression, such as `lap <= 5` or `lap == 5`. (Exact error wording can vary between Java versions.)

PAUSE AND TRY**Question 1** Tracing

What does this code print? Make a trace table with columns for `step`, the condition, and the output.

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

Question 2 Multiple choice

In the lap program at the start of this subsection, which statements make up the loop body?

- A. Only line 2, while (lap <= 5)
- B. Lines 4 and 5, the print and lap++;
- C. Lines 4, 5, and 7, every statement after the condition
- D. Every line from 1 to 7

2.7.2 Check, Run, Repeat

To trace loops correctly you need a precise picture of the order Java follows. A common mental picture is "the loop keeps going until the condition becomes false," and it leads to mistakes, because it does not say **when** the condition is looked at.

Here is the exact order. Java evaluates the condition **before each iteration, including the very first one**:

1. Evaluate the condition.
2. If it is `true`, run the whole body, top to bottom, then go back to step 1.
3. If it is `false`, skip the body and continue with the first statement after the loop. The loop is over; this is called **terminating** the loop.

Think of a ticket checker at the entrance to a ride. Each time around, a rider must show a valid ticket at the gate **before** boarding. Once on the ride, the rider stays on until the ride ends, even if the ticket expires halfway through. The gate (the condition) is checked only at the entrance (the top of the loop).

Two facts follow from this order:

- The condition is checked one more time than the body runs. The extra, final check is the one that comes out `false`.
- When a loop ends, its condition is `false`. Whatever variables the condition uses hold the first values that made it fail.

KEY IDEA

Java checks a while loop's condition before every iteration, including the first. If it is true, the entire body runs; if it is false, the loop ends and Java goes to the statement after it.

WORKED EXAMPLE Doubling bacteria

A colony starts with 1 bacterium and doubles every hour. The loop below doubles the count as long as it is under 20. What does the print after the loop show?

```
1 int bacteria = 1;
2 while (bacteria < 20)
3 {
4     bacteria *= 2;
5 }
6 System.out.println(bacteria);
```

Check	bacteria before the check	bacteria < 20?	Body does
1	1	true	bacteria becomes 2
2	2	true	bacteria becomes 4
3	4	true	bacteria becomes 8
4	8	true	bacteria becomes 16
5	16	true	bacteria becomes 32
6	32	false	(skipped; loop ends)

Line 6 prints:

```
32
```

Two things to notice. First, the condition was checked six times, but the body ran five times. Second, the answer is not 16 or 20. When `bacteria` was 16, the condition `16 < 20` was `true`, so the body ran one more time and doubled it to 32. Only then did the check fail. The value after the loop is the first value that makes the condition `false`.

Now vary it: track two variables, one that the condition tests and one that counts how many iterations happen.

WORKED EXAMPLE Cooling tea

A cup of tea starts at 95 degrees and cools 8 degrees per minute. How many minutes until it is 70 degrees or cooler, and what temperature is it then?

```
1 int temperature = 95;
2 int minutes = 0;
3 while (temperature > 70)
4 {
5     temperature -= 8;
6     minutes++;
7 }
8 System.out.println("Minutes: " + minutes);
9 System.out.println("Temperature: " + temperature);
```

Check	temperature	minutes	temperature > 70?
1	95	0	true

2	87	1	true
3	79	2	true
4	71	3	true
5	63	4	false

Each row shows the values at the moment of the check. In each `true` row, the body subtracts 8 from `temperature` and adds 1 to `minutes`, giving the next row. At check 4, `71 > 70` is still `true`, so one more minute passes. The output is:

```
Minutes: 4
Temperature: 63
```

`minutes` was never in the condition. It is a counter that records how many times the body ran, which here is how many minutes passed. Notice the final temperature, 63, overshoots 70; the loop stops at the first value that fails the test, not at exactly 70.

WATCH OUT

Expecting the loop to stop in the middle of the body. The condition is checked only at the top. If the body makes the condition `false` partway through, the rest of the body still runs:

```
1 int pages = 10;
2 while (pages > 0)
3 {
4     pages -= 4;
5     System.out.println("Pages left: " + pages);
6 }
```

```
Pages left: 6
Pages left: 2
Pages left: -2
```

On the third iteration, `pages` starts at 2. The check `2 > 0` is `true`, so the body begins. Line 4 makes `pages` equal to -2, which would fail the condition, but Java does not look at the condition again until the body is finished. Line 5 runs and prints -2. Only then does the next check, `-2 > 0`, come out `false`.

If printing a negative number is wrong for your program, the fix is to change the logic, for example by checking before you subtract. A loop does not quietly stop the moment a value goes out of range.

PAUSE AND TRY

Question 3 Tracing

What does this code print? Show `tickets` and the condition at each check.

```
1 int tickets = 13;
2 while (tickets >= 5)
3 {
4     tickets -= 5;
5     System.out.println(tickets);
6 }
7 System.out.println("Done");
```

Question 4 Multiple choice

In this code, how many times is the condition `value < 100` evaluated, and what is printed?

```
1 int value = 1;
2 while (value < 100)
3 {
4     value *= 3;
5 }
6 System.out.println(value);
```

- A. 5 times; prints 81
- B. 5 times; prints 243
- C. 6 times; prints 243
- D. 6 times; prints 100

2.7.3 When the Body Never Runs

Since the condition is checked before the first iteration too, ask: what if it is already `false` the very first time? Then the body is skipped completely. A while loop can run its body **zero times**.

This is not a flaw. It is often what you want. Picture a coffee station that should have 8 cups ready. A helper's rule is "while there are fewer than 8 cups, pour another." If 10 cups are already there, the right number of pours is zero. Any rule that always poured at least once would be wrong.

So a loop runs its body **zero or more times**: zero when the condition starts out `false`, and some number of times otherwise. Compare this with the `if` statement from Topic 2.3:

Statement	Condition checked	Body runs
<code>if</code>	Once	0 or 1 times
<code>while</code>	Before every iteration	0, 1, 2, ... times (zero or more)

KEY IDEA

If a while loop's condition is `false` the first time it is checked, the body never runs, and Java goes straight to the statement after the loop.

WORKED EXAMPLE No pouring needed

```
1 int cups = 10;
2 while (cups < 8)
3 {
4     System.out.println("Pour a cup");
5     cups++;
6 }
7 System.out.println("Cups ready: " + cups);
```

At the first check, `cups` is 10 and `10 < 8` is `false`. Java skips lines 3 to 6 entirely and runs line 7:

```
Cups ready: 10
```

`Pour a cup` never appears, and `cups++` never runs, so `cups` is still 10.

Now change only the starting value, from 10 to 6.

WORKED EXAMPLE Two cups to pour

```
1 int cups = 6;
2 while (cups < 8)
3 {
4     System.out.println("Pour a cup");
5     cups++;
6 }
7 System.out.println("Cups ready: " + cups);
```

Check	cups	cups < 8?	Printed
1	6	true	Pour a cup
2	7	true	Pour a cup
3	8	false	

```
Pour a cup
Pour a cup
Cups ready: 8
```

The loop is identical; only the value it started with changed, and the number of iterations went from zero to two. How many times a loop runs depends on the values its variables hold **when the loop begins**. These are the loop's **initial conditions**. Before you trust a loop, ask: what must be true of these variables before the loop starts for it to do what was intended?

WATCH OUT

Reusing a counter without resetting it. A workout program should print three warm-ups and then two cool-downs, using the same counter for both loops:

```
1 int count = 1;
2 while (count <= 3)
3 {
4     System.out.println("Warm-up " + count);
5     count++;
6 }
7 while (count <= 2)
8 {
9     System.out.println("Cool-down " + count);
10    count++;
11 }
12 System.out.println("Workout finished");
```

```
Warm-up 1
Warm-up 2
Warm-up 3
Workout finished
```

The cool-downs are missing, and no error appears. The first loop ends when `count` reaches 4 (that is the value that made `count <= 3` false). The second loop starts with `count` still 4, so its first check, `4 <= 2`, is false, and its body runs zero times. The second loop's initial condition was wrong.

The fix is to give the counter the starting value the second loop needs before it begins:

```
1 int count = 1;
2 while (count <= 3)
3 {
4     System.out.println("Warm-up " + count);
5     count++;
6 }
7 count = 1;
8 while (count <= 2)
9 {
10    System.out.println("Cool-down " + count);
11    count++;
12 }
13 System.out.println("Workout finished");
```

```
Warm-up 1
Warm-up 2
Warm-up 3
Cool-down 1
Cool-down 2
Workout finished
```

PAUSE AND TRY**Question 5** Multiple choice

In the "Two cups to pour" code, which starting value of `cups` makes the loop body run zero times?

- A. 0
- B. 7
- C. 8
- D. -2

Question 6 Short answer

Look again at the loop in Question 3 (the `tickets` loop). Describe every starting value of `tickets` for which `Done` is the only line printed, and explain why.

2.7.4 Infinite Loops

Every loop so far ended because the body changed a variable until the condition became `false`. What if that never happens? Then the condition is `true` at every check, the body repeats forever, and the statement after the loop is never reached. This is an **infinite loop**: a loop whose condition always evaluates to `true`.

A program stuck in an infinite loop does not crash with an error message. It keeps running. Depending on the body, it may flood the screen with the same line over and over, or show nothing at all and seem frozen. You have to stop it yourself, with the stop button in your coding environment or by pressing Ctrl+C in a terminal. Some online tools stop a program automatically after a time limit.

Because these programs never finish, the code in this subsection is shown without output. Each one is explained by reasoning about the condition, which is exactly how you will spot them on paper.

KEY IDEA

An infinite loop happens when the loop condition is true every time it is checked. To avoid one, make sure the body changes a variable in the condition, in the direction that will eventually make the condition false.

WORKED EXAMPLE The forgotten update

This loop was meant to print three laps:

```
1 int lap = 1;
2 while (lap <= 3)
3 {
4     System.out.println("Lap " + lap);
5 }
```

Trace the first few checks:

Check	lap	lap <= 3?
1	1	true
2	1	true
3	1	true
...	1	true

Nothing in the body changes `lap`, so every check sees the same value, 1, and gets the same answer, `true`. The body prints a lap line, the check passes again, and this repeats forever. A trace table makes the problem jump out: the column for the condition's variable never changes.

The fix is to add the update inside the braces:

```
1 int lap = 1;
2 while (lap <= 3)
3 {
4     System.out.println("Lap " + lap);
5     lap++;
6 }
```

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

The update can also be present but still never make the condition `false`. Here are three more ways that happens.

The update moves the wrong way. This countdown adds instead of subtracting:

```
1 int seconds = 3;
2 while (seconds > 0)
3 {
4     System.out.println(seconds);
5     seconds++;
6 }
```

`seconds` goes 3, 4, 5, 6, ..., moving **away** from 0, so `seconds > 0` keeps coming out `true`. (Strictly speaking, an `int` cannot grow forever. After about two billion iterations, `seconds` would pass the largest `int` and wrap around to a negative number, the integer overflow from Topic 1.5, and the loop would end with a nonsense value. Either way the loop is broken; treat it as infinite.) The fix is `seconds--`, as in the countdown in 2.7.1.

The update skips over the stopping value. A condition written with `!=` stops only when the variable lands **exactly** on one value:

```
1 int distance = 7;
2 while (distance != 0)
3 {
4     distance -= 2;
5 }
```

`distance` goes 7, 5, 3, 1, -1, -3, ... It jumps from 1 straight to -1 and is never equal to 0, so `distance != 0` is always `true`. Every value is odd, and subtracting 2 from an odd number always gives another odd number, so not even overflow rescues it. Using `>` instead of `!=` catches every value past the stopping point:

```
1 int distance = 7;
2 while (distance > 0)
3 {
4     distance -= 2;
5 }
6 System.out.println(distance);
```

This prints `-1`: the loop ends at the first value that is not greater than 0.

The update is outside the body. Braces decide what the body is, not indentation. Without braces, only the **one** statement right after the condition is the body:

```
1 int laps = 0;
2 while (laps < 3)
3     System.out.println("Swim a lap");
4     laps++;
5 System.out.println("Pool closed");
```

The indentation suggests that `laps++` belongs to the loop, but Java ignores indentation. The body is only the print statement, so `laps` stays 0, and `Swim a lap` repeats forever. `laps++` and `Pool closed` would run only after the loop, which never comes.

A related slip is a semicolon right after the condition:

```
1 int laps = 0;
2 while (laps < 3);
3 {
4     System.out.println("Swim a lap");
5     laps++;
6 }
```

That `;` is an empty statement, and it becomes the whole loop body. Java checks `laps < 3`, does nothing, checks again, does nothing, forever. The block in braces below it never runs, so this program prints nothing and seems frozen. The fix for both slips is the same: no semicolon after the condition, and braces around every statement that should repeat. Always using braces, even for a one-line body, prevents both mistakes.

WATCH OUT

Thinking every bad loop is infinite. An infinite loop needs a condition that is `true` at **every** check. A loop whose condition is `false` from the start does not run forever; it runs zero times (see 2.7.3). For example, `while (lap > 10)` with `lap` starting at 1 and `lap++` in the body is wrong for counting to 10, but it is not infinite: the first check is `false` and the body is skipped. To decide whether a loop is infinite, ask two questions: is the condition `true` at the start, and does the body move the variable toward making it `false`?

PAUSE AND TRY**Question 7** Multiple choice

Which loop is infinite? Assume each is run on its own.

- A. `int n = 10; while (n > 0) { n -= 3; }`
- B. `int n = 10; while (n != 0) { n -= 5; }`
- C. `int n = 10; while (n != 0) { n -= 4; }`
- D. `int n = 10; while (n > 20) { n++; }`

Question 8 Debugging

This code should print `Swim a lap` three times and then `Pool closed`. Instead it never finishes. Find the error and correct it.

```
1 int laps = 0;
2 while (laps < 3)
3 {
4     System.out.println("Swim a lap");
5 }
6 laps++;
7 System.out.println("Pool closed");
```

2.7.5 Off-by-One Errors

A loop can end, and still be wrong by exactly one iteration. Maybe it prints laps 1 through 4 when you wanted 1 through 5, or six numbers when you wanted five. This is so common that it has a name: an **off-by-one error** is when a loop runs one time too many or one time too few.

Off-by-one errors come from the edges of the loop: the starting value, whether the condition uses `<` or `<=` (or `>` or `>=`), and where the update sits in the body. The middle iterations are almost always fine. That tells you where to look: check the **first** iteration and the **last** iteration.

KEY IDEA

An off-by-one error makes a loop run one time too many or one time too few. Check the edges: trace the first iteration and the last one, paying close attention to the starting value and to `<` versus `<=`.

WORKED EXAMPLE One too few

This loop was meant to announce laps 1 through 5:

```
1 int lap = 1;
2 while (lap < 5)
3 {
4     System.out.println("Lap " + lap);
5     lap++;
6 }
```

```
Lap 1
Lap 2
Lap 3
Lap 4
```

Check the edges. The first iteration is fine: `lap` starts at 1. The last one is the problem. When `lap` is 5, the check `5 < 5` is `false`, because 5 is not less than itself. So the body never runs with `lap` equal to 5, and Lap 5 is missing.

The fix is to include 5: change the condition to `lap <= 5`, as in the lap program from 2.7.1.

Now vary the other edge: the starting value.

WORKED EXAMPLE One too many

```
1 int lap = 0;
2 while (lap <= 5)
3 {
4     System.out.println("Lap " + lap);
5     lap++;
6 }
```

```
Lap 0
Lap 1
Lap 2
Lap 3
Lap 4
Lap 5
```

The last iteration is right (`5 <= 5` is `true`, so Lap 5 prints), but the first is wrong: `lap` starts at 0, so Lap 0 prints too. That is six iterations, one too many.

A quick counting rule helps here. When the variable goes up by 1 each iteration:

Start	Condition	Values the body sees	Iterations
1	<code>lap <= 5</code>	1, 2, 3, 4, 5	5
1	<code>lap < 5</code>	1, 2, 3, 4	4
0	<code>lap < 5</code>	0, 1, 2, 3, 4	5
0	<code>lap <= 5</code>	0, 1, 2, 3, 4, 5	6

With `<=`, the count is $(\text{end} - \text{start} + 1)$; with `<`, it is $(\text{end} - \text{start})$. Both "start at 1 with `<=`" and "start at 0 with `<`" run five times, but they hand different values to the body. Choose the pair that matches what the body should see.

WATCH OUT

Moving the update changes what the body sees. Here the update comes **before** the print:

```
1 int lap = 1;
2 while (lap <= 5)
3 {
4     lap++;
5     System.out.println("Lap " + lap);
6 }
```

Lap 2
Lap 3
Lap 4
Lap 5
Lap 6

The loop still runs five times (lap is 1, 2, 3, 4, 5 each time the body starts). But each iteration adds 1 before printing, so the printed values are one higher: 2 through 6. The first and last outputs are both off by one.

If the update must come first, shift the start and the condition to match. Starting one lower, at 0, and using < gives the intended output:

```
1 int lap = 0;
2 while (lap < 5)
3 {
4     lap++;
5     System.out.println("Lap " + lap);
6 }
```

Lap 1
Lap 2
Lap 3
Lap 4
Lap 5

The order of statements in the body matters as much as the condition. When you check the edges, trace the body line by line, in order.

PAUSE AND TRY

Question 9 Tracing

How many times does this code print Hop! ?

```
1 int jump = 3;
2 while (jump <= 10)
3 {
4     System.out.println("Hop!");
5     jump++;
6 }
```

Question 10 Debugging

This code should print the numbers 10 down to 1, one per line. It prints 10 down to 2 instead. Find the error and correct it.

```
1 int n = 10;
2 while (n > 1)
3 {
4     System.out.println(n);
5     n--;
6 }
```

2.7.6 Loops That Stop on a Condition

Every loop in this chapter so far could be described as "do this N times." But look back at the cooling tea in 2.7.2. Nobody told the program "repeat 4 times." It was told "repeat while the tea is hotter than 70," and the 4 came out of the loop. That is the real power of a while loop: it keeps going until something is achieved, even when you do not know in advance how many steps that takes.

That gives you a way to decide when a task needs a loop at all. A task needs iteration when a step must be **repeated**, and especially when the number of repetitions depends on data or on how a process unfolds: "keep asking until the answer is valid," "keep halving until it is small enough," "keep reading scores until the end marker." If a step happens once, or if the decision is made just once, a plain statement or an `if` is enough.

Task	Needs a loop?	Why
Print a warning if the temperature is above 100	No	One decision, made once: use <code>if</code> .
Keep doubling a number until it is at least 1000	Yes	The doubling repeats; count not given.
Read and print scores until the value -1 appears	Yes	Reading repeats until a marker is seen.
Add 5 bonus points to one score	No	One step, done once.

KEY IDEA

Use a loop when a step must repeat until a goal is reached. A while loop's condition describes when to keep going, so it works even when nobody knows ahead of time how many iterations it will take.

WORKED EXAMPLE A bouncing ball

A ball dropped from 100 cm bounces back to half its previous height each time. How many bounces until it rises less than 10 cm?

```

1 int height = 100;
2 int bounces = 0;
3 while (height >= 10)
4 {
5     height /= 2;
6     bounces++;
7 }
8 System.out.println("Bounces: " + bounces);
9 System.out.println("Final height: " + height);

```

Check	height	bounces	height >= 10?
1	100	0	true
2	50	1	true
3	25	2	true
4	12	3	true
5	6	4	false

```

Bounces: 4
Final height: 6

```

On the step from 25, `height /= 2` is integer division (Topic 1.3), so `25 / 2` gives 12, not 12.5. The program never needed to know "4" in advance; the condition described the goal, and the counter `bounces` reported how many iterations it took.

Now a different kind of stopping condition: stop when a special value shows up in the input. A value whose only job is to mark the end of the data is often called a **sentinel**. Here the sentinel is -1, which can never be a real score. The `Scanner` is built from a string, as in Topic 1.4, so the output is predictable; reading from the keyboard works the same way.

WORKED EXAMPLE Reading scores until -1

ScoresUntilStop.java

```

1 import java.util.Scanner;
2
3 public class ScoresUntilStop
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("12 7 30 -1 99");
8         int score = input.nextInt();
9         while (score != -1)
10        {
11            System.out.println("Score: " + score);
12            score = input.nextInt();
13        }
14        System.out.println("End of scores");
15    }
16 }

```

The pattern has three parts:

- **Line 8** reads the first value **before** the loop, so the condition has something to test the first time.
- **Line 9** checks it: keep going while it is not the sentinel.
- **Line 12**, the last line of the body, reads the **next** value. This is the update: it changes `score`, the variable the condition tests.

Check	score	score != -1?	Printed
1	12	true	Score: 12
2	7	true	Score: 7
3	30	true	Score: 30
4	-1	false	

Score: 12
 Score: 7
 Score: 30
 End of scores

The 99 after the sentinel is never read: once -1 arrives, the loop ends. With different input, such as "5 -1", the same code would run once; with "-1", zero times. The data decides the number of iterations.

WATCH OUT

Reading the next value at the wrong point in the body. Here the read moves to the top of the body, before the print:

ScoresReadFirstWrong.java

```

1  import java.util.Scanner;
2
3  public class ScoresReadFirstWrong
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("12 7 30 -1 99");
8          int score = input.nextInt();
9          while (score != -1)
10         {
11             score = input.nextInt();
12             System.out.println("Score: " + score);
13         }
14         System.out.println("End of scores");
15     }
16 }
```

Score: 7
 Score: 30
 Score: -1
 End of scores

Two things go wrong. The first score, 12, read on line 8, is replaced on line 11 before it is ever printed. And when line 11 reads the sentinel -1, line 12 prints it as if it were a score, because the condition is not checked again until the body ends (the rule from 2.7.2). Keep the read as the **last** statement of the body, so that every value is checked by the condition before the body uses it.

Leaving out that read entirely is worse: `score` would never change, and the loop would be infinite.

PAUSE AND TRY

Question 11 Multiple choice

Which task needs a loop?

- A. Print `Passing` if a single test score is at least 70
- B. Subtract a 5-dollar fee from an account balance once
- C. Keep subtracting 5 dollars a week from a balance until it drops below 20, and report how many weeks that took
- D. Convert one temperature from Celsius to Fahrenheit

Question 12 Tracing

A plant is 3 cm tall and grows 4 cm per week. What does this code print?

```
1 int heightCm = 3;
2 int weeks = 0;
3 while (heightCm < 20)
4 {
5     heightCm += 4;
6     weeks++;
7 }
8 System.out.println(weeks + " weeks, " + heightCm + " cm");
```

Putting it together

This program combines a loop with an `if` statement inside its body (Topic 2.3) and a compound condition (Topic 2.5). A hiker starts with 20 units of energy. Each mile uses 6 units. The hike continues while the hiker has energy left **and** has walked fewer than 6 miles. After each mile, the program prints the mile number, adding : `tired` once energy drops below 10.

WORKED EXAMPLE A hike with two ways to stop

```

1 int energy = 20;
2 int miles = 0;
3 while (energy > 0 && miles < 6)
4 {
5     miles++;
6     energy -= 6;
7     if (energy < 10)
8     {
9         System.out.println("Mile " + miles + ": tired");
10    }
11    else
12    {
13        System.out.println("Mile " + miles);
14    }
15 }
16 System.out.println("Miles: " + miles + ", energy: " + energy);

```

Read the loop's structure before tracing:

- **Line 3:** the condition uses `&&`, so the loop continues only while **both** parts are `true`. It stops as soon as either one is `false`: energy runs out, or 6 miles are done.
- **Lines 5 and 6:** the updates. `miles` goes up and `energy` goes down, so each part of the condition is moving toward `false`.
- **Lines 7 to 14:** an `if/else` that chooses which message to print. It is inside the body, so it runs once per iteration.

Each row shows the values at a check, then what that iteration prints. Inside the body, `miles` goes up by 1 and `energy` drops by 6 **before** the `if` tests `energy < 10`.

energy	miles	Condition	Printed (energy after body)
20	0	<code>true && true → true</code>	Mile 1 (14)
14	1	<code>true && true → true</code>	Mile 2: tired (8)
8	2	<code>true && true → true</code>	Mile 3: tired (2)
2	3	<code>true && true → true</code>	Mile 4: tired (-4)
-4	4	<code>false && ... → false</code>	

At the last check, `energy > 0` is `false`, so the whole `&&` expression is `false` (and with short-circuit evaluation from Topic 2.5, Java does not even evaluate `miles < 6`). The loop ends because energy ran out, even though `miles < 6` was still true. The output is:

```

Mile 1
Mile 2: tired
Mile 3: tired
Mile 4: tired
Miles: 4, energy: -4

```

Notice the ideas from each subsection at work. The condition is checked before every mile, including the first (2.7.2). Energy went negative inside the fourth iteration, but the rest of that iteration still ran and printed `Mile 4: tired`, because the check happens only at the top. If

energy had started at 0, the first check would be `false` and the body would run zero times (2.7.3). If line 6 were missing and `miles++` stayed, the `miles < 6` part would still end the loop; if both updates were missing, it would be infinite (2.7.4).

Vocabulary

Term	Definition	Example
iteration	Repeating a segment of code; also, one run through a loop's body	The loop did five iterations.
loop	A statement that repeats a block of code zero or more times while a boolean expression is true; also called an iteration statement	<code>while (lap <= 5) { ... }</code>
flow of control	The order in which a program's statements run	A loop sends control back up.
while loop	A loop that checks its condition before each iteration and runs the body when the condition is true	<code>while (cups < 8) { ... }</code>
loop condition	The boolean expression in a loop's header that decides whether the body runs again	<code>seconds > 0</code>
loop body	The statements inside the loop's braces that repeat	<code>{ lap++; }</code>
trace table	A table that records variable values, the condition, and output at each step of a loop	Columns: lap, check, output
terminate	For a loop, to end because its condition is false	Ends when lap is 6
initial conditions	The values a loop's variables must hold before it starts for it to work as intended	<code>count = 1;</code> before the loop
infinite loop	A loop whose condition is true at every check, so it never ends	No update in the body
off-by-one error	A loop that runs one time too many or one time too few	<code>lap < 5</code> when 5 was wanted
sentinel	A special input value that marks the end of the data	<code>-1</code> after the last score

Chapter summary

- A while loop repeats its body as long as its boolean condition is true. Use one when a step must repeat, especially when the number of repetitions is not known in advance.
- Java checks the condition before every iteration, including the first. If it is true, the whole body runs; if it is false, the loop ends and Java continues after it. The condition is checked once more than the body runs, and a loop ends at the first value that fails the test.
- If the condition is false at the very first check, the body runs zero times. How many times a loop runs depends on its variables' values when it starts.
- An infinite loop has a condition that is true every time it is checked: the update is missing, moves the wrong way, skips over a `!=` stopping value, or sits outside the braces.
- An off-by-one error runs a loop one time too many or too few. Check the edges: the starting value, `<` versus `<=`, and the order of statements in the body.

- Trace tables, with a column for the condition, are the main tool for following any loop step by step.

Review questions

Question 13 Multiple choice

What does this code print?

```
1 int low = 1;
2 int high = 20;
3 while (low < high && high > 10)
4 {
5     low *= 2;
6     high -= 3;
7 }
8 System.out.println(low + " " + high);
```

- A. 8 11
- B. 16 8
- C. 32 5
- D. 16 11

Question 14 Tracing

A heater warms a room quickly while it is cold, then slowly as it nears the target. What does this code print?

```
1 int temp = 50;
2 while (temp < 80)
3 {
4     if (temp < 65)
5     {
6         temp += 10;
7     }
8     else
9     {
10        temp += 4;
11    }
12    System.out.println(temp);
13 }
```

Question 15 Debugging

This code should print the even page numbers from 2 through 12. It stops at Page 10. Find the error and correct it.

```
1 int page = 2;
2 while (page < 12)
3 {
4     System.out.println("Page " + page);
5     page += 2;
6 }
```

Question 16 Multiple choice

A program should count how many tries a player needs. Which loop is infinite when `tries` starts at 0 and `attempts` starts at 0?

- A. `while (tries < 3) { tries++; }`
- B. `while (tries < 3) { attempts++; }`
- C. `while (tries > 3) { attempts++; }`
- D. `while (tries < 3) { tries += 2; }`

Question 17 Short answer

Look back at the "Doubling bacteria" loop in 2.7.2. Without changing the loop itself, a student sets `bacteria` to 0 instead of 1 on line 1. Explain what happens when the code runs. Then describe what must be true of the starting value of `bacteria` for this loop to end.

Question 18 Tracing

What does this program print?

RevWordLengths.java

```
1 import java.util.Scanner;
2
3 public class RevWordLengths
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("hop skip stop jump");
8         String word = input.next();
9         while (!word.equals("stop"))
10        {
11            System.out.println(word.length());
12            word = input.next();
13        }
14        System.out.println("Stopped");
15    }
16 }
```

Question 19 Multiple choice

This program uses the same loop as "Reading scores until -1" in 2.7.6, but with different input. What does it print?

RevSentinelFirst.java

```
1 import java.util.Scanner;
2
3 public class RevSentinelFirst
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("-1 12 7");
8         int score = input.nextInt();
9         while (score != -1)
10        {
11            System.out.println("Score: " + score);
12            score = input.nextInt();
13        }
14        System.out.println("End of scores");
15    }
16 }
```

- A. Score: -1, then End of scores
- B. Only End of scores
- C. Score: 12, Score: 7, then End of scores
- D. Nothing; the loop never ends

Reflection

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

- ☐ I can name the condition and the body of a while loop and explain how it differs from an if statement. Reread: 2.7.1 Repeating Code with a while Loop
- ☐ I can trace a while loop with a trace table, including the final check that ends it. Reread: 2.7.2 Check, Run, Repeat
- ☐ I can explain why a while loop may run zero times and describe the starting values a loop needs. Reread: 2.7.3 When the Body Never Runs
- ☐ I can spot an infinite loop and explain how to fix it. Reread: 2.7.4 Infinite Loops
- ☐ I can find and fix an off-by-one error by checking the first and last iterations. Reread: 2.7.5 Off-by-One Errors
- ☐ I can recognize when a task needs a loop and read a loop that stops on a sentinel value. Reread: 2.7.6 Loops That Stop on a Condition

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

It prints 2, 5, and 8 on separate lines.

step	step <= 8?	Printed
2	true	2
5	true	5
8	true	8
11	false	

The update is `step += 3`, so the variable jumps by 3 each iteration. When `step` is 8, `8 <= 8` is still true (`<=` includes equality), so 8 prints. Then `step` becomes 11, the condition is false, and the loop ends. If you stopped before printing 8, you treated `<=` like `<`.

Reread 2.7.1 *Repeating Code with a while Loop* if your answer differed.

Question 2

B. The loop body is the block of statements between the braces that follow the condition. In that program, that is the `print` on line 4 and `lap++`; on line 5.

- **A** is wrong: line 2 is the loop header, which holds the condition. It decides whether the body runs; it is not the body.
- **C** is wrong: line 7 comes after the closing brace, so it runs once, after the loop ends.
- **D** is wrong: line 1 runs once, before the loop starts, to give `lap` its first value, and line 7 runs once after.

Reread 2.7.1 *Repeating Code with a while Loop* if you chose differently.

Question 3

It prints:

```
8
3
Done
```

Check	tickets	tickets >= 5?	Body prints
1	13	true	8
2	8	true	3
3	3	false	

The body subtracts first and then prints, so the printed values are the ones **after** subtracting. At check 3, `3 >= 5` is false, so the loop ends and line 7 prints `Done`. If you printed 13 first, you printed before subtracting; read the body in order.

Reread 2.7.2 *Check, Run, Repeat* if your answer differed.

Question 4

C. `value` goes 1, 3, 9, 27, 81, 243. The condition is checked at each of those six values. It is `true` for the first five, so the body runs five times, and `false` for 243, which ends the loop. Line 6 prints 243.

- **A** is wrong: 81 still passes `81 < 100`, so the body runs once more. The loop ends at the first value that fails, not the last that passed. Also, there are six checks, not five.
- **B** is wrong: the body runs five times, but the condition is checked one extra time, the final check that comes out `false`.
- **D** is wrong: `value` only takes values the body produces by tripling, and 100 is not one of them. A loop does not stop at the number in its condition.

Reread 2.7.2 *Check, Run, Repeat* if you chose differently.

Question 5

C. The body runs zero times exactly when the first check is `false`. With `cups` at 8, `8 < 8` is `false` (8 is not less than itself), so Java skips straight to the print, which shows `Cups ready: 8`.

- **A** is wrong: `0 < 8` is `true`, so the body runs; it keeps running until `cups` reaches 8.
- **B** is wrong: `7 < 8` is `true`, so the body runs once before `cups` becomes 8. One is not zero.
- **D** is wrong: `-2 < 8` is `true`, so the body runs. Negative numbers are still less than 8.

Reread 2.7.3 *When the Body Never Runs* if you chose differently.

Question 6

`Done` is the only output whenever `tickets` starts **less than 5** (4, 3, 0, -10, and so on).

A complete answer includes these ideas:

- The condition `tickets >= 5` is checked before the first iteration.
- If `tickets` starts below 5, that first check is `false`, so the body, which holds the only other print, runs zero times.
- Java then continues with the statement after the loop, which prints `Done`.

A starting value of exactly 5 does not count: `5 >= 5` is `true`, so the body runs once and prints 0 before `Done`.

Reread 2.7.3 *When the Body Never Runs* if your answer differed.

Question 7

C. `n` goes 10, 6, 2, -2, -6, ... It skips from 2 to -2 and is never exactly 0, so `n != 0` is `true` at every check. (Like the odd numbers in the example, every value is 2 more than a multiple of 4, and 0 is a multiple of 4, so `n` can never land on it.)

- **A** is wrong: `n` goes 10, 7, 4, 1, -2, and `-2 > 0` is `false`, so the loop ends. A `>` condition catches the first value past the stopping point.
- **B** is wrong: `n` goes 10, 5, 0. It lands exactly on 0, so `n != 0` becomes `false` and the loop ends.
- **D** is wrong: the first check is `10 > 20`, which is `false`, so the body runs zero times. A loop that never starts cannot run forever.

Reread 2.7.4 *Infinite Loops* if you chose differently.

Question 8

The update `laps++`; is **after** the closing brace, so it is not part of the loop body. Inside the loop, nothing changes `laps`, so it stays 0, `laps < 3` is true at every check, and the loop is infinite. `laps++`; and `Pool closed` are never reached.

Move the update inside the braces:

```
1 int laps = 0;
2 while (laps < 3)
3 {
4     System.out.println("Swim a lap");
5     laps++;
6 }
7 System.out.println("Pool closed");
```

Now `laps` goes 0, 1, 2, 3, and the corrected code prints `Swim a lap` three times and then `Pool closed`.

Reread 2.7.4 *Infinite Loops* if your answer differed.

Question 9

8 times. The body runs with `jump` equal to 3, 4, 5, 6, 7, 8, 9, and 10. At `jump 10`, `10 <= 10` is true, so the eighth `Hop!` prints; then `jump` becomes 11 and `11 <= 10` is false.

Using the counting rule: with `<=`, the count is $\text{end} - \text{start} + 1 = 10 - 3 + 1 = 8$. If you answered 7, you used $\text{end} - \text{start}$, which is the rule for `<`. If you answered 10, you counted from 1 instead of from the starting value 3.

Reread 2.7.5 *Off-by-One Errors* if your answer differed.

Question 10

The condition `n > 1` is false when `n` is 1, so the body never runs with `n` equal to 1, and 1 is never printed. The loop runs one time too few.

Change line 2 so that 1 passes the check:

```
1 while (n >= 1)
```

With that change, the code prints 10 down to 1. (`n > 0` also works, since for whole numbers it is true for exactly the same values as `n >= 1`.) Checking the last iteration finds this kind of error: ask "what is the last value that should be printed, and does it pass the condition?"

Reread 2.7.5 *Off-by-One Errors* if your answer differed.

Question 11

C. The subtraction repeats, and how many weeks it takes is not given; it depends on the starting balance. A while loop with a condition like `balance >= 20` and a week counter fits.

- **A** is wrong: it is a single decision made once, which is an `if` statement.
- **B** is wrong: one step done once needs no repetition.
- **D** is wrong: one calculation, done once.

Reread 2.7.6 *Loops That Stop on a Condition* if you chose differently.

Question 12

It prints 5 weeks, 23 cm.

Check	heightCm	weeks	heightCm < 20?
1	3	0	true
2	7	1	true
3	11	2	true
4	15	3	true
5	19	4	true
6	23	5	false

At check 5, $19 < 20$ is still `true`, so one more week of growth happens. The loop ends at the first height that is not under 20, which is 23, not exactly 20. If you answered 4 weeks, you stopped at 19.

Reread 2.7.6 *Loops That Stop on a Condition* if your answer differed.

Question 13

B. The loop continues while **both** `low < high` and `high > 10` are true.

Check	low	high	low < high && high > 10?
1	1	20	true && true → true
2	2	17	true && true → true
3	4	14	true && true → true
4	8	11	true && true → true
5	16	8	false && ... → false

The values after the loop are 16 and 8, so it prints 16 8.

- **A** is wrong: at check 4, $8 < 11$ and $11 > 10$ are both still `true`, so the body runs once more. Stopping here ends the loop one iteration early.
- **C** is wrong: it runs the body a fifth time. At check 5, $16 < 8$ is `false`, so the loop has already ended.
- **D** is wrong: it updates `low` but not `high` on the last iteration. The body always runs completely, so both statements run.

Reread 2.7.2 *Check, Run, Repeat* if you chose differently.

Question 14

It prints 60, 70, 74, 78, 82, one per line.

temp at check	temp < 80?	Branch in body	Printed
50	true	50 < 65: add 10	60
60	true	60 < 65: add 10	70
70	true	else: add 4	74
74	true	else: add 4	78
78	true	else: add 4	82

82	false		
----	-------	--	--

The `if` is inside the body, so it decides again on every iteration which update to use. When `temp` is 78, `78 < 80` is `true`, so one more iteration runs, and 82 prints even though it is past 80: the print comes after the update, and the condition is not checked until the body ends.

Reread 2.7.2 *Check, Run, Repeat* if your answer differed.

Question 15

This is an off-by-one error at the last iteration. When `page` reaches 12, the check `12 < 12` is `false`, so `Page 12` never prints. The loop runs one time too few.

Change line 2 to include 12:

```
1 while (page <= 12)
```

Now `page` takes the values 2, 4, 6, 8, 10, 12 in the body, and the loop ends when `page` becomes 14. The corrected code prints `Page 2` through `Page 12`. The update goes up by 2 here instead of 1, but the way to find the bug is the same: ask whether the last value you want passes the condition.

Reread 2.7.5 *Off-by-One Errors* if your answer differed.

Question 16

B. The condition tests `tries`, but the body updates `attempts`. `tries` stays 0, so `tries < 3` is `true` at every check and the loop never ends. The update has to change a variable **in the condition**.

- **A** is wrong: `tries` goes 0, 1, 2, 3, and `3 < 3` is `false`, so the loop ends after three iterations.
- **C** is wrong: the first check, `0 > 3`, is `false`, so the body runs zero times. It is not infinite; it never starts.
- **D** is wrong: `tries` goes 0, 2, 4, and `4 < 3` is `false`. Skipping over 3 is harmless here because the condition uses `<`, not `!=`.

Reread 2.7.4 *Infinite Loops* if you chose differently.

Question 17

With `bacteria` at 0, the first check `0 < 20` is `true`, so the body runs. But `0 * 2` is 0, so `bacteria` stays 0, every check is `true`, and the loop is **infinite**; line 6 is never reached.

A complete answer includes these ideas:

- The condition is checked first and is `true` for 0.
- Doubling 0 never changes it, so the body never moves the variable toward making the condition `false`.
- For the loop to end, `bacteria` must start **positive** (1 or more). Then doubling makes it grow until it reaches 20 or more. (A starting value of 20 or more also ends it, by running the body zero times.)

This is an initial condition: the same loop works or runs forever depending only on the value it starts with.

Reread 2.7.3 *When the Body Never Runs* and 2.7.4 *Infinite Loops* if your answer differed.

Question 18

It prints:

```
3
4
Stopped
```

Check	word	!word.equals("stop")?	Printed
1	"hop"	true	3
2	"skip"	true	4
3	"stop"	false	

Line 8 reads "hop" before the loop. The condition uses `equals` (Topic 2.6) and `!` (Topic 2.5): it is `true` while the word is **not** "stop". Each iteration prints the word's length and reads the next word as its last step. When "stop" is read, the next check is `false`, so the loop ends and "jump" is never read. "stop" acts as the sentinel.

Reread 2.7.6 *Loops That Stop on a Condition* if your answer differed.

Question 19

B. Line 8 reads the first value, -1, before the loop. The first check, `-1 != -1`, is `false`, so the body runs zero times and Java goes straight to line 14, which prints `End of scores`. The 12 and 7 after the sentinel are never read.

- **A** is wrong: the condition is checked before the first iteration, so the print inside the body never gets a chance to show -1.
- **C** is wrong: the loop does not skip the sentinel and look for later scores; it stops as soon as the sentinel is the value being tested.
- **D** is wrong: an infinite loop needs a condition that is `true` at every check. This one is `false` at the very first check.

Reread 2.7.3 *When the Body Never Runs* and 2.7.6 *Loops That Stop on a Condition* if you chose differently.