

1.3 Expressions and Output

Self-Study Guide · About 65 minutes · With answer key

By the end you will be able to

- Predict exactly what `System.out.print` and `System.out.println` display, and where the next output will appear.
- Write string literals, and use the escape sequences for a double quote, a backslash, and a new line.
- Evaluate arithmetic expressions built from numbers, variables, and the operators `+`, `-`, `*`, `/`, and `%`, and say whether the result is an `int` or a `double`.
- Explain why dividing two `int` values keeps only the whole-number part, and use `%` to find a remainder.
- Use operator precedence and parentheses to work out the value of an expression with several operators.
- Explain why dividing an `int` by the `int` zero compiles but throws an `ArithmeticException` when the program runs.

What to remember from earlier

- Every statement ends with a semicolon. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.4 From Algorithm to Program](#))
- A syntax error is caught by the compiler, so nothing runs. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.6 Syntax Errors](#))
- A logic error lets the program run to the end with a wrong result. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.7 Logic Errors and Testing](#))
- An exception is a run-time error that stops the program partway. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.8 Run-Time Errors and Exceptions](#))
- An `int` holds a whole number and a `double` holds a real number. ([Topic 1.2: Variables and Data Types, 1.2.3 The Three Primitive Types: int, double, and boolean](#))
- A variable is a named storage location with a type, as in `int tickets = 3;` ([Topic 1.2: Variables and Data Types, 1.2.4 Declaring a Variable](#))

1.3.1 Displaying Output with `println`

A program that calculates a grade or a ticket price is useless if nobody can see the answer. Programs need a way to show information to the person using them. In Topic 1.1 you saw `System.out.println` display a line of text. This subsection explains exactly what it does, because the details decide where every piece of output lands.

When a Java program displays text, it goes to the **console**: the output window of your IDE, sometimes called the terminal. Picture the console as a page that the program writes on from top to bottom. A blinking marker called the **cursor** shows where the next character will be written. Every time the program displays something, it starts writing at the cursor, and the cursor moves along behind the text.

`System.out.println(...)` does two things, always in this order:

1. It displays the value inside the parentheses, starting at the cursor.
2. It moves the cursor to the beginning of the next line.

The name is short for "print line": print this, then finish the line. Displaying information like this is called **output**.

Part	What it means
<code>System.out</code>	The console. Read <code>System.out.println</code> as one command name for now; what the dots mean is taught in later topics.
<code>println</code>	"Print line": display the value, then move the cursor to the next line.
<code>(...)</code>	The value to display goes inside the parentheses.
<code>;</code>	Ends the statement, as in Topic 1.1.

KEY IDEA

`System.out.println` displays its value on the console and then moves the cursor to the start of the next line.

WORKED EXAMPLE A lunch menu, line by line

Watch where the cursor is after each statement, and notice line 7, which has nothing inside the parentheses.

MenuBoard.java

```

1  public class MenuBoard
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Today's lunch:");
6          System.out.println("Veggie tacos");
7          System.out.println();
8          System.out.println("Enjoy!");
9      }
10 }
```

Line	What is displayed	Where the cursor ends up
5	Today's lunch:	Start of line 2 of the console
6	Veggie tacos	Start of line 3
7	nothing	Start of line 4
8	Enjoy!	Start of line 5

Line 7 is `System.out.println();` with empty parentheses. There is nothing to display, but `println` still does its second job: it moves the cursor down. The result is an empty line on the console:

```

Today's lunch:
Veggie tacos

Enjoy!
```

The empty `println()` is the usual way to put a blank line in your output. Each `println` statement, with or without a value, ends exactly one line.

WATCH OUT

println with a capital I. In many fonts, a lowercase `l` (as in "line") and a capital `I` look almost the same, so students type `printIn`:

PrintlnTypo.java

```
1 public class PrintlnTypo
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Practice starts at 4.");
6     }
7 }
```

Java is case-sensitive and spells the command with lowercase letters only, so it looks for something named `printIn`, finds nothing, and the compiler reports:

```
PrintlnTypo.java:5: error: cannot find symbol
```

It is a syntax error, so nothing runs. The fix is `println`: p-r-i-n-t-l-n, "print line."

PAUSE AND TRY

Question 1 Tracing

Write the exact console output of this program, including any empty lines.

BusSchedule.java

```
1 public class BusSchedule
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Bus 12");
6         System.out.println();
7         System.out.println("Leaves at 7:45");
8         System.out.println("Arrives at 8:10");
9     }
10 }
```

Question 2 Multiple choice

What does the statement `System.out.println();` do?

- A. Nothing at all; it is skipped.
- B. It causes a compiler error because there is no value to display.
- C. It moves the cursor to the start of the next line, which can leave an empty line.
- D. It displays the word `println`.

1.3.2 print Versus println

Sometimes you do not want a new line after every piece of output. A scoreboard should show
Score: 95 on one line, even if the label and the number come from two separate statements. You need a way to display something and leave the cursor right where it stopped.

That is what `System.out.print(...)` does. It displays its value starting at the cursor and **does not** move the cursor to a new line. The cursor stays just after the last character displayed, so the next output continues on the same line.

Statement	Displays the value?	Then moves the cursor to a new line?
<code>System.out.print(...);</code>	Yes	No: the cursor stays on the same line
<code>System.out.println(...);</code>	Yes	Yes

KEY IDEA

`print` leaves the cursor at the end of what it displayed; `println` moves the cursor to the next line afterward. The difference shows up in the next output, not in the current one.

WORKED EXAMPLE Ready, set, go

Watch where the cursor is after each statement.

```
1 System.out.print("Ready ");
2 System.out.print("set ");
3 System.out.println("go!");
4 System.out.println("Race over.");
```

Statement	Console line 1	Console line 2	Cursor afterward
<code>print("Ready ")</code>	Ready		Line 1, just after the space
<code>print("set ")</code>	Ready set		Line 1, just after the space
<code>println("go!")</code>	Ready set go!		Start of line 2
<code>println("Race over.")</code>	Ready set go!	Race over.	Start of line 3

```
Ready set go!
Race over.
```

The spaces inside `"Ready "` and `"set "` matter. `print` adds nothing of its own, not even a space, so the only spaces in the output are the ones you put there.

The most common use of `print` is a label followed by a value. Change one thing: leave out the space at the end of a label.

```
1 System.out.print("Score: ");
2 System.out.println(95);
3 System.out.print("Level:");
4 System.out.println(3);
```

```
Score: 95
Level:3
```

"Score: " ends with a space, so 95 appears one space after the colon. "Level:" has no space, so 3 lands right against the colon. Both pairs end with `println`, so each label-and-value pair gets its own line. (Later, in Topic 1.15, you will learn a way to join text and numbers in one statement. For now, a `print` followed by a `println` does the job.)

WATCH OUT

Expecting `print` to start a new line. Students often write `print` for every statement and expect one line each:

```
1 System.out.print("Line one");
2 System.out.print("Line two");
```

Line oneLine two

`print` never moves to a new line, so the second text starts exactly where the first one stopped. Use `println` when a line is finished:

```
1 System.out.println("Line one");
2 System.out.println("Line two");
```

Line one
Line two

PAUSE AND TRY**Question 3** Tracing

Write the exact output.

```
1 System.out.print("A");
2 System.out.println("B");
3 System.out.print("C");
4 System.out.print("D");
5 System.out.println();
6 System.out.println("E");
```

Question 4 Tracing

What is displayed, and after the last statement, where is the cursor?

```
1 System.out.print("Name: ");
2 System.out.println("Ada");
3 System.out.print("Grade: ");
4 System.out.print(11);
5 System.out.println();
```

1.3.3 Literals and String Literals

You have now displayed text, and in Question 4 a number, 11. The two were written differently: the text had double quotes and the number did not. Why? Because Java needs to know what kind of value you mean, and the way you write a value in code tells it.

A **literal** is a fixed value written directly in the code. When Java sees 42, it does not look anything up; the value is exactly forty-two, every time the program runs. Each type from Topic 1.2 has its own way of writing literals:

Kind of literal	Examples	What tells Java the type
int literal	42, 0, 2025	A whole number with no decimal point
double literal	3.75, 7.0, 0.5	A number with a decimal point
boolean literal	true, false	One of the two words, with no quotes
string literal	"Hello", "42", ""	Characters inside double quotes

A **string literal** is text written between a pair of double quotes: any characters, one after another. "Characters" means anything you can type: letters, digits, spaces, punctuation. The double quotes are not part of the text. They are markers that tell Java where the text starts and stops, so they are not displayed. Everything between them is kept exactly, including spaces. A pair of quotes with nothing between, "", is called the empty string.

KEY IDEA

A literal is a fixed value written in code. A string literal is a sequence of characters in double quotes; the quotes mark its start and end and are not displayed.

WORKED EXAMPLE One of each kind

```
1 System.out.println(42);
2 System.out.println(3.75);
3 System.out.println(true);
4 System.out.println("42");
5 System.out.println("Room 101, 2nd floor!");
```

```
42
3.75
true
42
Room 101, 2nd floor!
```

- Line 1 displays the `int` literal 42.
- Line 2 displays the `double` literal 3.75.
- Line 3 displays the `boolean` literal `true`. There are no quotes, so this is the value `true`, not text.
- Line 4 displays the **string** literal `"42"`. It looks the same on the console as line 1, but to Java it is text: the character 4 followed by the character 2. In 1.3.5 you will see why that difference matters: Java calculates with numbers, while text in quotes is only displayed, never calculated.

- Line 5 shows that a string literal can hold digits, spaces, commas, and punctuation. Java does not interpret any of it; it displays it as written.

Change one thing: look at how `double` literals and spacing come out.

```
1 System.out.println(2.50);
2 System.out.println(7.0);
3 System.out.println(" spaced out");
4 System.out.println("");
5 System.out.println("end");
```

```
2.5
7.0
 spaced out

end
```

A `double` is displayed without extra zeros at the end, so `2.50` shows as `2.5`. But it always keeps at least one digit after the decimal point, so `7.0` shows as `7.0`, not `7`. That `.0` is how you can tell a `double` result from an `int` on the console. In the string literal, every space between the quotes is kept. The empty string `""` displays nothing, and `println` then ends the line, so you see an empty line.

WATCH OUT

Forgetting the quotes around text. Without quotes, Java does not treat a word as text. It thinks the word is the name of something, such as a variable from Topic 1.2:

MissingQuotes.java

```
1 public class MissingQuotes
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(Welcome);
6     }
7 }
```

```
MissingQuotes.java:5: error: cannot find symbol
```

Java looked for a variable named `Welcome`, found none, and stopped. Put the text in double quotes: `System.out.println("Welcome");`. (Leaving out only the closing quote gives the unclosed string literal error you saw in Topic 1.1.)

PAUSE AND TRY

Question 5 Multiple choice

Which of these is a string literal?

- A. `Hello`
- B. `"17"`

- C. 17
- D. true

Question 6 Tracing

Write the exact output.

```
1 System.out.println("3.0");
2 System.out.println(3.0);
3 System.out.println(3.50);
4 System.out.println("3.50");
```

1.3.4 Escape Sequences

A double quote marks where a string literal begins and ends. So what if the text itself needs a double quote, as in *She said "hi" to me.*? Try the obvious thing:

QuoteInside.java

```
1 public class QuoteInside
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("She said "hi" to me.");
6     }
7 }
```

Java reads `"She said "` as a complete string literal, because the second quote closes it. Then it finds `hi`, which is not text and not in a place that makes sense, and the compiler reports errors such as `' ) ' or ' , ' expected` and `';' expected` (the exact wording varies between Java versions). Nothing runs.

Java solves this with a code. Inside a string literal, a backslash `\` means "the next character is special; do not treat it normally." A backslash followed by a character is an **escape sequence**: a two-character code inside a string that stands for one character that would be hard or impossible to type there directly. This course uses three:

Escape sequence	Displays	Why it is needed
<code>\"</code>	a double quote <code>"</code>	A plain <code>"</code> would end the string literal
<code>\\</code>	one backslash <code>\</code>	A single <code>\</code> would start an escape sequence
<code>\n</code>	a new line (moves the cursor)	Pressing Enter inside a string literal is not allowed

Note the direction: escape sequences use the **backslash** `\`, which leans left, not the forward slash `/`.

KEY IDEA

An escape sequence starts with a backslash and stands for one special character: `\"` is a double quote, `\\` is a backslash, and `\n` is a new line.

WORKED EXAMPLE Quotes and backslashes inside text

```
1 System.out.println("She said \"hi\" to me.");
2 System.out.println("A backslash looks like this: \\");
3 System.out.println("\"\\\\\"");
```

```
She said "hi" to me.
A backslash looks like this: \
"\
```

- Line 1: each `\` displays one `"` and does **not** end the string. The only quote that ends it is the last one, which has no backslash in front.
- Line 2: `\\` displays one backslash. The second backslash is the character; the first one announces it.
- Line 3 looks strange, so read it slowly from the left: `"` opens the literal, `\` is a quote, `\\` is a backslash, `\` is a quote, `"` closes the literal. Three characters are displayed: `"\"`.

Change one thing: use the third escape sequence, `\n`, which moves the cursor to the next line in the middle of a string.

```
1 System.out.println("Roses are red,\nViolets are blue.");
2 System.out.print("one\ntwo\n");
3 System.out.println("three");
```

```
Roses are red,
Violets are blue.
one
two
three
```

Line 1 displays two lines from one statement: `\n` acts like pressing Enter between `red,` and `Violets,` and then `println` ends the second line as usual. Line 2 uses `print`, which never ends a line on its own. But its text ends with `\n`, so the cursor still ends up at the start of a new line, and `three` appears on its own line.

WATCH OUT

A single backslash in a file path. Computer paths such as `C:\data` contain backslashes, and students type them as they appear. Java treats each `\` as the start of an escape sequence. `\d` is not one Java knows, so this does not compile:

PathIllegal.java

```
1 public class PathIllegal
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Saved in C:\data");
6     }
7 }
```

```
PathIllegal.java:5: error: illegal escape character
```

Worse, if the letter after the backslash happens to make a real escape sequence, the program compiles and is quietly wrong, which is a logic error:

```
1 System.out.println("Saved in C:\new");
```

```
Saved in C:
ew
```

`\n` became a new line, so the `n` of `new` disappeared. The fix in both cases is to write the backslash as `\\`:

```
1 System.out.println("Saved in C:\\new");
```

```
Saved in C:\new
```

PAUSE AND TRY

Question 7 Tracing

Write the exact output, including any empty lines.

```
1 System.out.print("Title: \"Hopper\"\\n");
2 System.out.println("Use \\n for a new line");
3 System.out.println("Done\\n");
4 System.out.println("!");
```

Question 8 Debugging

This statement is supposed to display `Line 1` and `Line 2` on separate lines, but it displays `Line 1/nLine 2` on one line. Find the error and write the corrected statement.

```
1 System.out.println("Line 1/nLine 2");
```

1.3.5 Arithmetic Expressions

Displaying fixed text only goes so far. A ticket booth needs to show the cost of 3 tickets at \$12 each, and the answer depends on the numbers. You could work out 36 yourself and type it in, but then the program is no help when the numbers change. You want Java to do the calculation.

Java calculates with **arithmetic expressions**. An **expression** is a piece of code that Java evaluates (works out) to produce a single value. An arithmetic expression is made of numeric values (literals such as 12 or 2.5, and numeric variables), joined by **arithmetic operators**, the symbols that say which calculation to do. The values an operator works on are its **operands**. In `3 * 12`, the operator is `*` and the operands are 3 and 12.

Operator	Meaning	Example	Value
+	addition	7 + 2	9
-	subtraction	7 - 2	5
*	multiplication	7 * 2	14
/	division	8 / 2	4
%	remainder	7 % 2	1

Multiplication uses `*` because there is no `x` key on a keyboard. Division and remainder have surprises of their own, so each gets its own subsection (1.3.6 and 1.3.7). When you put an expression inside `println`, Java first evaluates it, then displays the value.

WORKED EXAMPLE Calculating instead of typing the answer

```
1 System.out.println(3 * 12);
2 System.out.println("3 * 12");
3 System.out.println(20 - 7 + 1);
4 System.out.println(6.5 + 2);
```

```
36
3 * 12
14
8.5
```

- Line 1: Java evaluates `3 * 12` to 36 and displays 36.
- Line 2: the same characters inside double quotes form a string literal, so Java displays them as text. No calculation happens.
- Line 3: Java works out `20 - 7 = 13`, then `13 + 1 = 14`.
- Line 4: `6.5 + 2` is 8.5. The result has a decimal point; the next rule explains why.

The type of the result

Every value in Java has a type, so the result of an arithmetic operation has a type too. Java decides it from the two operands:

- If **both operands are** `int` values, the result is an `int`.
- If **at least one operand is a** `double`, the result is a `double`.

The reason: an `int` can hold only whole numbers, so whenever a `double` is involved the answer might need a decimal part, and only a `double` can hold one. Java does not look at whether the answer happens to be a whole number. It decides from the types alone.

```
1 System.out.println(6 * 2);
2 System.out.println(6 * 2.0);
3 System.out.println(1.5 + 2.5);
4 System.out.println(10 - 0.25);
```

```
12
12.0
4.0
9.75
```

$6 * 2$ uses two `int` values, so the result is the `int` `12`. Change one operand to `2.0` and the result is the `double` `12.0`: the same amount, but a different type, which is why it displays with `.0`. $1.5 + 2.5$ is `4.0` and not `4`, because both operands are `double` values, even though the sum is a whole number.

KEY IDEA

An operation on two `int` values gives an `int`. An operation with at least one `double` gives a `double`, even when the answer is a whole number.

Variables in expressions

An operand can also be a variable. When Java evaluates the expression, it uses the value the variable holds at that moment. Here the variables are declared with starting values, as in Topic 1.2. (The `=` sign that gives a variable its value is studied fully in Topic 1.4.)

WORKED EXAMPLE Ticket costs with variables

```
1 int tickets = 3;
2 double price = 12.5;
3 int fee = 2;
4 System.out.println(tickets * price);
5 System.out.println(tickets * fee);
6 System.out.println(price - fee);
```

Expression	Values used	Operand types	Result
<code>tickets * price</code>	<code>3 * 12.5</code>	<code>int, double</code>	<code>37.5 (double)</code>
<code>tickets * fee</code>	<code>3 * 2</code>	<code>int, int</code>	<code>6 (int)</code>
<code>price - fee</code>	<code>12.5 - 2</code>	<code>double, int</code>	<code>10.5 (double)</code>

```
37.5
6
10.5
```

Evaluating an expression does not change the variables. After these lines, `tickets` still holds `3`, `price` still holds `12.5`, and `fee` still holds `2`.

WATCH OUT

Putting the expression inside quotes. Quotes turn anything into a string literal, including math and variable names:

```
1 int laps = 4;
2 System.out.println("laps * 400");
3 System.out.println(laps * 400);
```

```
laps * 400
1600
```

Line 2 displays the characters `\aps * 400` because they are inside quotes; Java never evaluates them. Line 3 has no quotes, so Java looks up `\aps` (4), multiplies, and displays 1600. If you want the value, leave the quotes off.

PAUSE AND TRY

Question 9 Tracing

Write the exact output.

```
1 int boxes = 5;
2 double weight = 2.5;
3 System.out.println(boxes + 10);
4 System.out.println(boxes * weight);
5 System.out.println(weight * 4);
6 System.out.println("boxes + 1");
```

Question 10 Multiple choice

What are the type and the value of the expression `4 * 2.5`?

- A. `int`, 10
- B. `double`, 10.0
- C. `int`, 8
- D. A string, `4 * 2.5`

1.3.6 Integer Division

Seventeen cookies are shared equally among 5 friends, and cookies cannot be broken. Each friend gets 3, with some left over. Java's division of two `int` values works exactly like this, and it surprises nearly every new programmer.

You just learned that an operation on two `int` values must give an `int`. But 17 divided by 5 is 3.4, and an `int` cannot hold 3.4. Java's answer: keep **only the whole-number part** of the quotient and throw the decimal part away. This is called **integer division**. It does not round: `19 / 5` is 3.8 in math, and Java gives 3, not 4. When at least one operand is a `double`, the result is a `double`, so Java keeps the full quotient, decimal part included.

Expression	Operand types	Result	Why
<code>17 / 5</code>	<code>int</code> , <code>int</code>	3	3.4 → whole part only
<code>19 / 5</code>	<code>int</code> , <code>int</code>	3	3.8 → whole part only (no rounding)
<code>2 / 3</code>	<code>int</code> , <code>int</code>	0	0.666... → whole part is 0
<code>17.0 / 5</code>	<code>double</code> , <code>int</code>	3.4	A <code>double</code> is involved: full quotient
<code>17 / 5.0</code>	<code>int</code> , <code>double</code>	3.4	It does not matter which side is <code>double</code>

KEY IDEA

Dividing an `int` by an `int` keeps only the whole-number part of the answer, with no rounding. If at least one operand is a `double`, the result is the full quotient as a `double`.

WORKED EXAMPLE Sharing cookies

```
1 System.out.println(17 / 5);
2 System.out.println(19 / 5);
3 System.out.println(2 / 3);
4 System.out.println(17.0 / 5);
5 System.out.println(17 / 5.0);
```

```
3
3
0
3.4
3.4
```

Look at line 3: `2 / 3` gives `0`. In math the quotient is about 0.67, and its whole-number part is 0. Whenever the first `int` is smaller than the second, integer division gives `0`.

Change one thing: average two test scores whose total is odd. The total of 85 and 90 is 175.

```
1 int total = 175;
2 System.out.println(total / 2);
3 System.out.println(total / 2.0);
```

```
87
87.5
```

`total / 2` divides two `int` values, so the `.5` is lost and the average shows as `87`. Writing the divisor as `2.0` makes one operand a `double`, so Java keeps the full answer, `87.5`. Same numbers, different types, different results.

WATCH OUT

Expecting `1 / 2` to be one half. A recipe calls for half a cup, and a student writes:

```
1 System.out.println(1 / 2);
```

```
0
```

The code compiles and runs, so nothing warns you; the wrong answer is a logic error. Both operands are `int`, so the result is the whole part of 0.5, which is `0`. Make at least one operand a `double`:

```
1 System.out.println(1.0 / 2);
```

0.5

PAUSE AND TRY**Question 11** Tracing

Write the exact output.

```
1 System.out.println(29 / 10);
2 System.out.println(29 / 10.0);
3 System.out.println(9 / 10);
4 System.out.println(30 / 10);
```

Question 12 Short answer

`System.out.println(7 / 2);` displays 3, but `System.out.println(7 / 2.0);` displays 3.5.
Explain why changing 2 to 2.0 changes the result.

1.3.7 The Remainder Operator

Back to the 17 cookies and 5 friends. Integer division told you each friend gets 3. But how many cookies are left over? Integer division throws that part away, and sometimes the leftover is exactly what you need: the minutes left after counting whole hours, or the people left after forming full teams.

The **remainder operator** `%` gives the amount left over when one whole number is divided by another. In `a % b`, Java divides `a` by `b` as many whole times as possible and reports what is left. For 17 and 5: 5 goes into 17 three whole times, which uses up 15, and 2 are left, so `17 % 5` is 2. The two operators fit together like this:

```
17 = 5 × (17 / 5) + (17 % 5)
17 = 5 ×   3   +   2
```

`/` tells you how many whole groups, `%` tells you how many are left over. The `%` sign has nothing to do with percent here; it is just the symbol Java uses. In this course, `a` is never negative and `b` is always positive.

KEY IDEA

`a % b` is the remainder when `a` is divided by `b`: what is left after taking out as many whole groups of `b` as possible.

WORKED EXAMPLE Cookies, and what is left over

```
1 System.out.println(17 / 5);
2 System.out.println(17 % 5);
```

```
3
2
```

Line 1 is integer division: 3 cookies each. Line 2 is the remainder: 2 cookies left over. Check: 5 friends $\times$ 3 cookies + 2 left over = 17.

Change things: try different first and second numbers to see what the remainder depends on.

```
1 System.out.println(20 % 5);
2 System.out.println(3 % 5);
3 System.out.println(1234 % 10);
4 System.out.println(125 / 60);
5 System.out.println(125 % 60);
6 System.out.println(14 % 2);
7 System.out.println(15 % 2);
```

Expression	Whole groups	Left over	Result	Common use
20 % 5	4 groups use 20	0	0	Remainder 0 means it divides evenly
3 % 5	0 groups (5 > 3)	3	3	A smaller first number is all left over
1234 % 10	123 groups use 1230	4	4	% 10 gives the last digit
125 / 60	2 whole hours		2	Hours in 125 minutes
125 % 60	2 hours use 120	5	5	Minutes left over
14 % 2	7 groups use 14	0	0	Remainder 0 when divided by 2: even
15 % 2	7 groups use 14	1	1	Remainder 1 when divided by 2: odd

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

Notice the second row. When the first number is smaller than the second, there is not even one whole group, so the whole first number is left over. So $3 \% 5$ is 3, not 0.

WATCH OUT

Reading % as "percent." A student wants 50 percent of 10 and writes:

```
1 System.out.println(50 % 10);
```

```
0
```

This is a remainder, not a percentage: 10 goes into 50 five whole times with nothing left over, so the result is 0. Java has no percent operator. Fifty percent of 10 would be written as a multiplication, such as $0.5 * 10$.

PAUSE AND TRY**Question 13** Tracing

A year has 365 days. Write the exact output, and say what each number means.

```
1 int days = 365;
2 System.out.println(days / 7);
3 System.out.println(days % 7);
```

Question 14 Multiple choice

What is the value of `8 % 12`?

- A. 0
- B. 4
- C. 8
- D. 12

1.3.8 Operator Precedence

What is `2 + 3 * 4`? If you work left to right you get 20, but if you multiply first you get 14. A program cannot be ambiguous, so Java needs a fixed rule for which operation happens first whenever an expression has more than one operator.

An expression with more than one operator is a **compound expression**. Java follows the same order you learned in math class. **Operator precedence** is the rule that decides which operators are grouped with their values first:

1. Anything in **parentheses** is evaluated first.
2. Next, `*`, `/`, and `%` (they share the same, higher precedence).
3. Last, `+` and `-` (they share the same, lower precedence).
4. When operators have the **same** precedence, they are evaluated from **left to right**.

The grouping is decided when the program is compiled. The compiler reads the expression and works out which operator belongs with which values, as if it were adding invisible parentheses: `2 + 3 * 4` is understood as `2 + (3 * 4)`. The actual calculation happens later, when the program runs.

KEY IDEA

`*`, `/`, and `%` are done before `+` and `-`. Operators with the same precedence go left to right. Parentheses change the order.

WORKED EXAMPLE Evaluating step by step

```

1 System.out.println(2 + 3 * 4);
2 System.out.println((2 + 3) * 4);
3 System.out.println(20 - 6 - 4);
4 System.out.println(24 / 4 * 2);
5 System.out.println(17 % 5 * 3);

```

Expression	Step 1	Step 2	Result
<code>2 + 3 * 4</code>	<code>3 * 4 → 12</code>	<code>2 + 12</code>	14
<code>(2 + 3) * 4</code>	<code>(2 + 3) → 5</code>	<code>5 * 4</code>	20
<code>20 - 6 - 4</code>	<code>20 - 6 → 14</code>	<code>14 - 4</code>	10
<code>24 / 4 * 2</code>	<code>24 / 4 → 6</code>	<code>6 * 2</code>	12
<code>17 % 5 * 3</code>	<code>17 % 5 → 2</code>	<code>2 * 3</code>	6

```

14
20
10
12
6

```

The last three rows have operators of equal precedence, so they go left to right. In `20 - 6 - 4`, doing the right side first (`6 - 4 = 2`, then `20 - 2`) would give 18, which is wrong. In `24 / 4 * 2`, `*` does not go before `/`; they share a level, so the left one goes first.

Change one thing: use precedence with integer division, in the classic average-of-two-quizzes problem.

```

1 int quiz1 = 85;
2 int quiz2 = 90;
3 System.out.println(quiz1 + quiz2 / 2);
4 System.out.println((quiz1 + quiz2) / 2);
5 System.out.println((quiz1 + quiz2) / 2.0);

```

Expression	Grouping Java uses	Result
<code>quiz1 + quiz2 / 2</code>	<code>85 + (90 / 2) = 85 + 45</code>	130
<code>(quiz1 + quiz2) / 2</code>	<code>175 / 2</code> , both int	87
<code>(quiz1 + quiz2) / 2.0</code>	<code>175 / 2.0</code> , one double	87.5

```

130
87
87.5

```

Without parentheses, only `quiz2` is divided, and the "average" is 130, which is higher than either quiz. Parentheses make the sum happen first. Dividing by `2.0` also keeps the `.5`.

The type is decided one operation at a time

Precedence also matters for the `int`/`double` rule. Java decides the type of each operation separately, using only that operation's two operands.

```
1 System.out.println(1 / 2 * 4.0);
2 System.out.println(1.0 / 2 * 4);
3 System.out.println(4.0 * 1 / 2);
```

Expression	Step 1	Step 2	Result
<code>1 / 2 * 4.0</code>	<code>1 / 2</code> → 0 (both <code>int</code>)	<code>0 * 4.0</code> → 0.0	0.0
<code>1.0 / 2 * 4</code>	<code>1.0 / 2</code> → 0.5	<code>0.5 * 4</code> → 2.0	2.0
<code>4.0 * 1 / 2</code>	<code>4.0 * 1</code> → 4.0	<code>4.0 / 2</code> → 2.0	2.0

```
0.0
2.0
2.0
```

In the first row, the `4.0` cannot rescue the answer. By the time Java reaches it, `1 / 2` has already been done as integer division and has given 0.

WATCH OUT

Forgetting parentheses when a sum should be divided. Four friends split a \$30 pizza and a \$6 tip:

```
1 int pizzaCost = 30;
2 int tip = 6;
3 int people = 4;
4 System.out.println(pizzaCost + tip / people);
```

```
31
```

Division goes first, so only the tip is divided: `6 / 4` is 1, and `30 + 1` is 31, which is more than the whole bill split four ways. Put the sum in parentheses: `(pizzaCost + tip) / people`, which is `36 / 4 = 9`.

PAUSE AND TRY

Question 15 Tracing

Write the exact output.

```
1 System.out.println(10 - 2 * 3);
2 System.out.println((10 - 2) * 3);
3 System.out.println(36 / 6 / 2);
4 System.out.println(7 + 10 % 4);
```

Question 16 Debugging

To change a Fahrenheit temperature to Celsius, subtract 32, then multiply by 5 and divide by 9. Water boils at 212°F, which is 100°C, but this code displays 195. Explain the error and write the corrected output statement.

```
1 int fahrenheit = 212;
2 System.out.println(fahrenheit - 32 * 5 / 9);
```

1.3.9 Dividing an Integer by Zero

In Topic 1.1 you saw `10 / 0` stop a program with an exception. Now that you know how `/` works, you can see exactly why, and why the compiler lets it through.

Dividing by zero has no answer: no number of zeros adds up to 10. Integer division must produce an `int`, and there is no `int` it could honestly give. Rather than make up a number and let the program continue with a wrong value, Java stops the program. When an `int` is divided by the `int` zero, Java throws an `ArithmeticException`, the exception for an impossible arithmetic operation.

This is a run-time error, not a syntax error. `snacks / teams` follows every rule of Java, so the compiler accepts it. The compiler only checks how the code is written and does not work out the values. The divisor is only known to be zero when the program actually runs and evaluates the expression. (This course only covers dividing by zero when both values are `int`. Dividing by zero when a `double` is involved behaves differently and is not part of this course.)

KEY IDEA

Dividing an `int` by the `int` zero compiles, but when that line runs, Java throws an `ArithmeticException` and the program stops.

WORKED EXAMPLE Splitting snacks among zero teams

TeamSplit.java

```
1 public class TeamSplit
2 {
3     public static void main(String[] args)
4     {
5         int snacks = 24;
6         int teams = 0;
7         System.out.println("Splitting snacks...");
8         System.out.println(snacks / teams);
9         System.out.println("Done!");
10    }
11 }
```

Moment	What happens	Console shows
Compile	Every statement follows the rules. Compiling succeeds.	(nothing)

Lines 5–6	snacks holds 24, teams holds 0.	(nothing)
Line 7	Displays the text.	Splitting snacks...
Line 8	Evaluates <code>24 / 0</code> . No int answer exists: <code>ArithmeticException</code> .	the exception message
Line 9	Never runs. The program has stopped.	(nothing)

```
Splitting snacks...
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TeamSplit.main(TeamSplit.java:8)
```

`/ by zero` names the problem and `TeamSplit.java:8` names the line. (The exact layout of this message can vary a little between Java versions.)

Change one thing: put the zero on the other side, and build a zero divisor from an expression.

```
1 System.out.println(0 / 24);
2 System.out.println(24 / (3 - 3));
3 System.out.println("Finished");
```

```
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Line 1, `0 / 24`, is fine: 24 goes into 0 zero times, so the result is 0. Zero is only a problem as the **second** operand, the divisor. Line 2 divides by `(3 - 3)`. Parentheses go first, the divisor works out to 0, and the exception is thrown. The compiler did not catch this either, because the divisor is not known until the expression is evaluated. `Finished` never appears.

WATCH OUT

"It compiled, so the division is safe." The compiler checks only that the code follows Java's rules, and dividing by a variable or an expression always does. Whether the divisor is zero depends on the values when the line runs. When you see a division, ask: *could the value after / be zero?* If it is zero, expect an `ArithmeticException` at that line, with the earlier output already displayed and nothing after it.

PAUSE AND TRY

Question 17 Multiple choice

Which statement throws an `ArithmeticException` when it runs?

- A. `System.out.println(0 / 7);`
- B. `System.out.println(7 / 4);`
- C. `System.out.println(7 / (4 - 4));`
- D. `System.out.println(7 * 0);`

Question 18 Tracing

Does this program compile? Write the exact output that appears before the program stops, and name the line where it stops.

ScoreBoard.java

```

1 public class ScoreBoard
2 {
3     public static void main(String[] args)
4     {
5         int points = 90;
6         int games = 6;
7         int forfeits = 6;
8         System.out.print("Average: ");
9         System.out.println(points / games);
10        System.out.println("Per played game:");
11        System.out.println(points / (games - forfeits));
12        System.out.println("Season over");
13    }
14 }

```

Putting it together

Maya is planning a pizza party: 3 pizzas, 8 slices each, 5 friends, and \$12.50 per pizza. Her program uses every idea from this chapter. Read it, predict the output, then check the trace.

PizzaParty.java

```

1 public class PizzaParty
2 {
3     public static void main(String[] args)
4     {
5         int pizzas = 3;
6         int slicesPerPizza = 8;
7         int friends = 5;
8         double pricePerPizza = 12.5;
9
10        System.out.println("*** \"Pizza Party\" Planner ***");
11        System.out.print("Total slices: ");
12        System.out.println(pizzas * slicesPerPizza);
13        System.out.print("Slices each: ");
14        System.out.println(pizzas * slicesPerPizza / friends);
15        System.out.print("Left over: ");
16        System.out.println(pizzas * slicesPerPizza % friends);
17        System.out.print("Total cost: $");
18        System.out.println(pizzas * pricePerPizza);
19        System.out.print("Cost each: $");
20        System.out.println(pizzas * pricePerPizza / friends);
21        System.out.println("\nThanks for planning!");
22    }
23 }

```

Line	Code (shortened)	Evaluation	Displays
10	println("*** \"Pizza Party\" Planner ***")	\" displays a quote	*** "Pizza Party" Planner ***

11-12	<code>print(...), println(pizzas * slicesPerPizza)</code>	<code>3 * 8 = 24 (int)</code>	Total slices: 24
13-14	<code>pizzas * slicesPerPizza / friends</code>	left to right: <code>24 / 5 = 4 (int division)</code>	Slices each: 4
15-16	<code>pizzas * slicesPerPizza % friends</code>	left to right: <code>24 % 5 = 4</code>	Left over: 4
17-18	<code>pizzas * pricePerPizza</code>	<code>3 * 12.5 = 37.5 (double)</code>	Total cost: \$37.5
19-20	<code>pizzas * pricePerPizza / friends</code>	<code>37.5 / 5 = 7.5 (double)</code>	Cost each: \$7.5
21	<code>println("\nThanks for planning!")</code>	<code>\n</code> first, so an empty line comes first	(empty line), Thanks for planning!

```
*** "Pizza Party" Planner ***
Total slices: 24
Slices each: 4
Left over: 4
Total cost: $37.5
Cost each: $7.5

Thanks for planning!
```

Connect each line to its subsection:

- **Output (1.3.1, 1.3.2):** each label uses `print`, so the value lands on the same line; the value uses `println`, so the next label starts a new line.
- **Literals and escape sequences (1.3.3, 1.3.4):** `\` puts quotes inside the title. The `\n` at the start of the last string creates the empty line. The `$` inside "Total cost: \$" is just a character.
- **Types (1.3.5):** slices are all `int`, so they stay whole. The cost involves the `double` `pricePerPizza`, so it shows a decimal point.
- **Integer division and remainder (1.3.6, 1.3.7):** 24 slices among 5 friends is 4 each with 4 left over: $5 \times 4 + 4 = 24$.
- **Precedence (1.3.8):** `*`, `/`, and `%` share a level, so `pizzas * slicesPerPizza / friends` multiplies first (left to right), then divides.
- **Dividing by zero (1.3.9):** every division here has `friends` (5) after the `/`. Before trusting a line like 14, ask whether that divisor could ever be zero.

Common mistakes

A lookup table for the messages and symptoms in this chapter. "Compiler" means nothing runs; the other rows run.

What you see	Likely cause	Fix
Compiler: cannot find symbol on <code>printIn</code>	Capital I instead of lowercase l	Spell it <code>println</code>
Compiler: cannot find symbol on a word	Text written without double quotes	Put the text in <code>"..."</code>

Compiler: <code>') ' expected or ' ; ' expected</code> near a quote	A plain <code>"</code> inside a string literal	Write the inner quote as <code>\"</code>
Compiler: <code>illegal escape character</code>	A single <code>\</code> before a letter, as in a file path	Write the backslash as <code>\\</code>
Output runs together on one line	<code>print</code> used where a line should end	Use <code>println</code> , or add <code>\n</code>
Output shows the formula, such as <code>laps * 400</code>	The expression is inside double quotes	Remove the quotes
Answer is missing its decimal part	Both operands of <code>/</code> are <code>int</code>	Make one operand a <code>double</code> , such as <code>2.0</code>
Answer is too large, such as an average of 130	Missing parentheses; <code>/</code> happened before <code>+</code>	Put the sum in parentheses
ArithmeticException: <code>/ by zero</code>	An <code>int</code> divisor was <code>0</code> when the line ran	Make sure the divisor cannot be zero

Vocabulary

Term	Definition	Example
console	The output window where a program displays text	Your IDE's output pane
cursor	The marker for where the next output will appear	After <code>print("Hi")</code> , it follows <code>Hi</code>
output	Information a program displays	<code>Score: 95</code>
<code>System.out.println</code>	Displays a value, then moves the cursor to the next line	<code>System.out.println("Hi");</code>
<code>System.out.print</code>	Displays a value and leaves the cursor on the same line	<code>System.out.print("Score: ");</code>
literal	A fixed value written directly in the code	<code>42</code> , <code>3.75</code> , <code>true</code>
string literal	A sequence of characters enclosed in double quotes	<code>"Room 101"</code>
escape sequence	A backslash plus a character, standing for one special character inside a string	<code>\"</code> , <code>\\</code> , <code>\n</code>
expression	Code that Java evaluates to produce a single value	<code>tickets * price</code>
arithmetic expression	An expression built from numeric values, variables, and arithmetic operators	<code>3 * 12 + 1</code>
arithmetic operator	A symbol for a calculation: <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code>	<code>*</code> multiplies
operand	A value an operator works on	In <code>3 * 12</code> : <code>3</code> and <code>12</code>
integer division	<code>/</code> with two <code>int</code> operands; keeps only the whole-number part of the quotient	<code>17 / 5</code> is <code>3</code>
remainder operator	<code>%</code> ; gives what is left over when one number is divided by another	<code>17 % 5</code> is <code>2</code>

compound expression	An expression with more than one operator	<code>2 + 3 * 4</code>
operator precedence	The rule for which operators are grouped first: <code>*</code> / <code>%</code> before <code>+</code> <code>-</code> , ties left to right	<code>2 + 3 * 4</code> is 14
ArithmeticException	The exception thrown when an <code>int</code> is divided by the <code>int</code> zero	<code>10 / 0</code>

Chapter summary

- `System.out.println` displays a value and then moves the cursor to the next line; `System.out.print` displays a value and leaves the cursor where it stopped, so the next output continues on the same line.
- A **literal** is a fixed value in code. A **string literal** is characters in double quotes; the quotes are not displayed. Inside one, the **escape sequences** `\`, `\\`, and `\n` stand for a double quote, a backslash, and a new line.
- **Arithmetic expressions** combine numeric values and variables with `+`, `-`, `*`, `/`, and `%`. Two `int` operands give an `int`; at least one `double` gives a `double`.
- `int` divided by `int` keeps only the whole-number part, with no rounding. `%` gives the remainder, what is left after taking out as many whole groups as possible.
- **Operator precedence:** parentheses first, then `*`, `/`, `%`, then `+`, `-`, with equal-precedence operators left to right. The type of each operation depends only on its own two operands.
- Dividing an `int` by the `int` zero compiles, but throws an `ArithmeticException` when the line runs, stopping the program.

Review questions

Question 19 Multiple choice

Which code segment displays exactly these two lines?

```
Go "Team"!
\o/
```

- `System.out.println("Go "Team"!\\n\\o/");`
- `System.out.print("Go \"Team\\\"!\\n\\o/");`
- `System.out.println("Go \"Team\\\"!/n\\o/");`
- `System.out.print("Go \"Team\\\"!");` followed by `System.out.print("\\o/");`

Question 20 Tracing

Write the exact output.

```
1 int pages = 7;
2 int days = 2;
3 double daysExact = 2.0;
4 System.out.println(pages / days);
5 System.out.println(pages / daysExact);
6 System.out.println(pages % days);
7 System.out.println(pages + days * 3);
8 System.out.println((pages + days) / 4);
```

Question 21 Multiple choice

Which expression evaluates to the `double` value 2.5?

- A. `5 / 2`
- B. `5 / 2 * 1.0`
- C. `5.0 / 2`
- D. `1.0 + 5 / 2`

Question 22 Debugging

This program should display the average of three quiz scores, 84.0. It displays `Average: 194`. Find the error and write the corrected line.

QuizAverage.java

```
1 public class QuizAverage
2 {
3     public static void main(String[] args)
4     {
5         int quiz1 = 80;
6         int quiz2 = 85;
7         int quiz3 = 87;
8         System.out.print("Average: ");
9         System.out.println(quiz1 + quiz2 + quiz3 / 3);
10    }
11 }
```

Question 23 Tracing

A movie is 200 minutes long. Write the exact output.

```
1 int totalMinutes = 200;
2 System.out.print(totalMinutes / 60);
3 System.out.print(" h ");
4 System.out.print(totalMinutes % 60);
5 System.out.println(" min");
6 System.out.println(totalMinutes / 60 * 60 + totalMinutes % 60);
```

Question 24 Short answer

A student makes two claims: (1) `System.out.println(24 / (3 - 3));` will not compile, because the compiler sees the division by zero. (2) `System.out.println(0 / 24);` will throw an exception, because there is a zero in it. Explain what really happens in each case.

Question 25 Debugging

This program should display:

```
Tickets: 4
Total: $50.0
```

Instead it displays `Tickets: 4Total: $tickets * price` on one line. Find both errors and write the corrected lines.

`TicketReceipt.java`

```
1 public class TicketReceipt
2 {
3     public static void main(String[] args)
4     {
5         int tickets = 4;
6         double price = 12.5;
7         System.out.print("Tickets: ");
8         System.out.print(tickets);
9         System.out.print("Total: $");
10        System.out.println("tickets * price");
11    }
12 }
```

Question 26 Tracing

Write the exact output. Work carefully: each line combines several rules.

```
1 System.out.print(3 / 4 * 8.0);
2 System.out.print(" ");
3 System.out.println(8.0 * 3 / 4);
4 System.out.println(2 + 12 % 5 * 3 - 1);
5 System.out.print("Done\n\"Bye\");
```

Reflection

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

- ☐ I can explain what `println` does after it displays its value. Reread: 1.3.1 Displaying Output with `println`
- ☐ I can predict where output lands when `print` and `println` are mixed. Reread: 1.3.2 `print` Versus `println`
- ☐ I can tell a string literal apart from a number literal and predict how each is displayed. Reread: 1.3.3 Literals and String Literals
- ☐ I can use escape sequences to put a double quote, a backslash, or a new line inside a string. Reread: 1.3.4 Escape Sequences
- ☐ I can evaluate an arithmetic expression and say whether the result is an `int` or a `double`. Reread: 1.3.5 Arithmetic Expressions
- ☐ I can predict the result of dividing two `int` values. Reread: 1.3.6 Integer Division
- ☐ I can use `%` to find what is left over after division. Reread: 1.3.7 The Remainder Operator
- ☐ I can evaluate a compound expression using precedence, left-to-right order, and parentheses. Reread: 1.3.8 Operator Precedence
- ☐ I can explain why dividing an `int` by zero compiles but throws an `ArithmeticException`. Reread: 1.3.9 Dividing an Integer by Zero

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Bus 12  
  
Leaves at 7:45  
Arrives at 8:10
```

Line 5 displays `Bus 12` and moves the cursor down. Line 6 has empty parentheses, so it displays nothing but still moves the cursor down, leaving an empty line. Lines 7 and 8 each display their text and end their line.

Reread *1.3.1 Displaying Output with `println`* if your output differed.

Question 2

C. `println` displays its value and then moves the cursor to the next line. With empty parentheses there is nothing to display, but the cursor still moves.

- **A** is wrong: the second job of `println`, moving the cursor, still happens.
- **B** is wrong: empty parentheses are allowed for `println`; `MenuBoard.java` compiles and runs.
- **D** is wrong: `println` is the name of the command, not text to display. Only a value inside the parentheses is displayed.

Reread *1.3.1 Displaying Output with `println`* if you chose differently.

Question 3

```
AB  
CD  
E
```

Statement	Effect
<code>print("A")</code>	Displays A; cursor stays after it
<code>println("B")</code>	Displays B right after A, then moves to the next line
<code>print("C")</code>	Displays C at the start of line 2
<code>print("D")</code>	Displays D right after C
<code>println()</code>	Displays nothing, moves to the next line, which ends line 2
<code>println("E")</code>	Displays E on line 3

The empty `println()` does not create an empty line here, because the cursor was in the middle of a line. It simply ends the line `CD`.

Reread *1.3.2 `print` Versus `println`* if your output differed.

Question 4

```
Name: Ada
Grade: 11
```

`print("Name: ")` leaves the cursor after the space, and `println("Ada")` finishes that line. `print("Grade: ")` and `print(11)` both stay on line 2, so 11 follows the label. The last statement, `println()`, displays nothing and moves the cursor to the **start of line 3**. Without it, the cursor would be left just after 11.

Reread 1.3.2 *print Versus println* if your answer differed.

Question 5

B. A string literal is a sequence of characters in double quotes. "17" is the text made of the characters 1 and 7.

- **A** is wrong: without quotes, `Hello` is read as a name, not as text, so Java looks for something called `Hello`.
- **C** is wrong: 17 is an `int` literal, a number.
- **D** is wrong: `true` with no quotes is a `boolean` literal.

Reread 1.3.3 *Literals and String Literals* if you chose differently.

Question 6

```
3.0
3.0
3.5
3.50
```

Lines 1 and 4 are string literals, so they are displayed exactly as typed, extra zero included. Lines 2 and 3 are `double` literals. A `double` is shown without extra trailing zeros but with at least one digit after the decimal point, so 3.0 stays 3.0 and 3.50 becomes 3.5.

Reread 1.3.3 *Literals and String Literals* if your output differed.

Question 7

```
Title: "Hopper"
Use \n for a new line
Done

!
```

- Line 1: `\"` twice gives the two quotes around `Hopper`. The `\n` at the end moves the cursor to the next line even though this is `print`.
- Line 2: `\\n` is `\\` (one backslash) followed by an ordinary `n`, so the characters `\n` are displayed instead of a new line.
- Line 3: `Done`, then `\n` moves to a new line, and then `println` moves down again. Two moves with nothing between them leave one empty line.
- Line 4: `!` appears after the empty line.

Reread 1.3.4 *Escape Sequences* if your output differed.

Question 8

Error: the code uses a forward slash, `/n`. Only a backslash starts an escape sequence, so `/n` is just two ordinary characters and is displayed as typed. The code compiles, so this is a logic error, not a syntax error.

Corrected statement:

```
1 System.out.println("Line 1\nLine 2");
```

Now `\n` moves the cursor to a new line, and the output is:

```
Line 1
Line 2
```

Reread 1.3.4 *Escape Sequences* if you missed the error.

Question 9

```
15
12.5
10.0
boxes + 1
```

- `boxes + 10` is `5 + 10`. Both are `int`, so the result is the `int` 15.
- `boxes * weight` is `5 * 2.5`. One operand is a `double`, so the result is the `double` 12.5.
- `weight * 4` is `2.5 * 4 = 10.0`. It is a whole amount, but one operand is a `double`, so it displays as 10.0.
- `"boxes + 1"` is a string literal, so it is displayed as text, not calculated.

Reread 1.3.5 *Arithmetic Expressions* if your output differed.

Question 10

B. One operand, 2.5, is a `double`, so the result is a `double`: 10.0.

- **A** is wrong: the value is ten, but the type comes from the operands, not from whether the answer is whole. With a `double` operand, the result is a `double`.
- **C** is wrong: Java does not drop the .5 from 2.5 before multiplying. Only the type rules decide the result's type, and here they give a `double`.
- **D** is wrong: there are no double quotes, so this is an arithmetic expression, not a string literal.

Reread 1.3.5 *Arithmetic Expressions* if you chose differently.

Question 11

```
2
2.9
0
3
```

- `29 / 10`: both `int`, quotient 2.9, keep the whole part: 2 (not rounded to 3).
- `29 / 10.0`: one `double`, full quotient: 2.9.
- `9 / 10`: both `int`, quotient 0.9, whole part 0.
- `30 / 10`: both `int`, divides evenly: 3. Since both are `int`, it is 3, not 3.0.

Reread 1.3.6 *Integer Division* if your output differed.

Question 12

Model answer: In $7 / 2$, both operands are `int` values, so the result must be an `int`. Java keeps only the whole-number part of 3.5, which is 3. In $7 / 2.0$, one operand is a `double`, so the result is a `double`, and Java keeps the full quotient, 3.5.

A complete answer must say that:

- the types of the operands decide the type of the result (two `int` values give an `int`; any `double` gives a `double`), and
- `int` division drops the decimal part instead of rounding.

Reread 1.3.6 *Integer Division* if your explanation differed.

Question 13

```
52
1
```

$365 / 7$ is integer division: 7 goes into 365 fifty-two whole times ($52 \times 7 = 364$), so a year has 52 whole weeks. $365 \% 7$ is what is left over: $365 - 364 = 1$ extra day.

Reread 1.3.7 *The Remainder Operator* if your output differed.

Question 14

C. 12 does not fit into 8 even once, so there are zero whole groups and all of 8 is left over.

- **A** is wrong: 0 is what $8 / 12$ gives (the number of whole groups), not what is left over.
- **B** is wrong: 4 is what $12 \% 8$ would give. The order matters: $8 \% 12$ divides 8 by 12, not 12 by 8.
- **D** is wrong: the leftover comes from the first number, 8. You cannot have more left over than you started with.

Reread 1.3.7 *The Remainder Operator* if you chose differently.

Question 15

```
4
24
3
9
```

Expression	Steps	Result
$10 - 2 * 3$	$2 * 3 = 6$ first, then $10 - 6$	4
$(10 - 2) * 3$	parentheses: 8, then $8 * 3$	24
$36 / 6 / 2$	left to right: $36 / 6 = 6$, then $6 / 2$	3
$7 + 10 \% 4$	$10 \% 4 = 2$ first, then $7 + 2$	9

Reread 1.3.8 *Operator Precedence* if your output differed.

Question 16

Error: `*` and `/` have higher precedence than `-`, so the subtraction happens last. Java evaluates $32 * 5 = 160$, then $160 / 9 = 17$ (integer division), and finally $212 - 17 = 195$. The code compiles and runs, so this is a logic error.

Corrected statement:

```
1 System.out.println((fahrenheit - 32) * 5 / 9);
```

The parentheses make the subtraction happen first: $212 - 32 = 180$. Then left to right, $180 * 5 = 900$ and $900 / 9 = 100$.

Reread *1.3.8 Operator Precedence* if your reasoning differed.

Question 17

C. Parentheses go first, so the divisor is $4 - 4 = 0$. Dividing the `int 7` by the `int 0` throws an `ArithmeticException`.

- **A** is wrong: zero is the first operand. 7 goes into 0 zero times, so the result is 0; nothing is thrown.
- **B** is wrong: $7 / 4$ is integer division, giving 1.
- **D** is wrong: multiplying by zero is fine and gives 0. Only dividing by zero has no answer.

Reread *1.3.9 Dividing an Integer by Zero* if you chose differently.

Question 18

Yes, it compiles. Every statement follows Java's rules; the compiler does not work out that `games - forfeits` is zero.

```
Average: 15
Per played game:
```

- Lines 8–9: `Average:` with `print`, then $90 / 6 = 15$ with `println`.
- Line 10: `Per played game:` on its own line.
- Line 11: parentheses first, $6 - 6 = 0$, so $90 / 0$ throws an `ArithmeticException` (`/ by zero`, at `ScoreBoard.java:11`). The program stops.
- Line 12 never runs, so `Season over` is never displayed.

Reread *1.3.9 Dividing an Integer by Zero* if your answer differed.

Question 19

B. `\` displays each quote, `\n` moves to a new line, and `\\` displays the backslash before `o/`. Using `print` is fine: the text itself contains the line break.

- **A** is wrong: the plain `"` before `Team` ends the string literal, and `\o` is not an escape sequence Java knows. It does not compile.
- **C** is wrong: `/n` uses a forward slash, so it is displayed as the characters `/n` and everything stays on one line: `Go "Team"!/n\o/`.
- **D** is wrong: `print` never moves to a new line, so the second text continues on the same line: `Go "Team"! \o/`.

Reread *1.3.4 Escape Sequences* and *1.3.2 print Versus println* if you chose differently.

Question 20

```
3
3.5
1
13
2
```

Expression	Steps	Result
pages / days	7 / 2, both int: whole part of 3.5	3
pages / daysExact	7 / 2.0, one double: full quotient	3.5
pages % days	2 goes into 7 three times (6), 1 left over	1
pages + days * 3	2 * 3 = 6 first, then 7 + 6	13
(pages + days) / 4	parentheses: 9, then 9 / 4, both int	2

Reread 1.3.6 *Integer Division* and 1.3.8 *Operator Precedence* if your output differed.

Question 21

C. One operand is a `double`, so the division keeps the full quotient: 2.5.

- **A** is wrong: both operands are `int`, so integer division gives the `int` 2.
- **B** is wrong: left to right, `5 / 2` happens first as integer division (2), and only then `2 * 1.0 = 2.0`. The `.5` was already lost.
- **D** is wrong: `/` has higher precedence, so `5 / 2` happens first and gives 2; then `1.0 + 2 = 3.0`.

Reread 1.3.8 *Operator Precedence* if you chose differently.

Question 22

Error: on line 9, `/` has higher precedence than `+`, so only `quiz3` is divided by 3: `87 / 3 = 29`, and `80 + 85 + 29 = 194`. It compiles and runs, so it is a logic error.

Corrected line:

```
1 System.out.println((quiz1 + quiz2 + quiz3) / 3.0);
```

The parentheses make the sum (252) happen first. Dividing by `3.0` makes the result a `double`, so the average keeps its decimal part when it has one. The program then displays `Average: 84.0`. Dividing by the `int` 3 would also work for these three scores, but it would drop the decimal part of an average such as 84.5.

Reread 1.3.8 *Operator Precedence* and 1.3.6 *Integer Division* if you missed part of the fix.

Question 23

```
3 h 20 min
200
```

- Line 2: `200 / 60` is integer division: 3 whole hours. `print` keeps the cursor on the line.
- Line 3: `" h "`, with a space on each side.
- Line 4: `200 % 60` is the leftover minutes: $3 \times 60 = 180$, so 20.
- Line 5: `" min"`, then `println` ends the line.

- Line 6: `/` and `*` go first, left to right: $200 / 60 = 3$, $3 * 60 = 180$. Then $200 \% 60 = 20$. Last, $180 + 20 = 200$. Whole groups times the group size, plus the remainder, gives back the original number.

Reread 1.3.7 *The Remainder Operator* if your output differed.

Question 24

Model answer: (1) It compiles, because the code follows Java's rules; the compiler checks how code is written, not what the values work out to. When it runs, the parentheses are evaluated to `0`, and dividing the `int` `24` by the `int` `0` throws an `ArithmeticException`, stopping the program. (2) It does not throw anything. The zero is the first operand, not the divisor; `24` goes into `0` zero times, so it displays `0`. Only a zero divisor (after the `/`) causes the exception.

A complete answer must:

- say that claim 1 compiles and fails at run time with an `ArithmeticException`, and
- say that claim 2 runs normally and displays `0`, because only the divisor matters.

Reread 1.3.9 *Dividing an Integer by Zero* if your reasoning differed.

Question 25

Error 1, line 8: `print` leaves the cursor after `4`, so the next label continues on the same line.

Change it to `println` to end the line.

Error 2, line 10: the expression is inside double quotes, so it is a string literal and is displayed as text. Remove the quotes so Java evaluates $4 * 12.5$, which is the `double` `50.0`.

Corrected lines:

```
1 System.out.println(tickets);
```

```
1 System.out.println(tickets * price);
```

Both are logic errors: the program compiles and runs, but the output is wrong.

Reread 1.3.2 *print Versus println* and 1.3.5 *Arithmetic Expressions* if you missed an error.

Question 26

```
0.0 6.0
7
Done
"Bye"
```

- Line 1: $3 / 4$ goes first (left to right) as integer division: `0`. Then $0 * 8.0 = 0.0$. `print` keeps the cursor on the line.
- Line 2: a space.
- Line 3: $8.0 * 3 = 24.0$, then $24.0 / 4 = 6.0$. The same numbers in a different order give a different answer, because here the `double` comes first. `println` ends the line.
- Line 4: $12 \% 5 = 2$, then $2 * 3 = 6$ (both before `+` and `-`). Then left to right: $2 + 6 = 8$, $8 - 1 = 7$.
- Line 5: `Done`, then `\n` starts a new line, then `\\"Bye\"` displays `"Bye"`.

Reread 1.3.8 *Operator Precedence* and 1.3.4 *Escape Sequences* if your output differed.