

1.6 Compound Assignment Operators

Self-Study Guide · About 35 minutes · With answer key

By the end you will be able to

- Use `+=`, `-=`, `*=`, `/=`, and `%=` to update a number variable, and work out the value it holds afterward.
- Explain why the whole right side of a compound assignment is worked out before it is combined with the variable.
- Use `++` and `--` to add one to or subtract one from a variable.
- Spot common slips such as `+=` in place of `+=`, and predict what Java does with them.

What to remember from earlier

- A variable has a type and holds one value at a time. ([Topic 1.2: Variables and Data Types, 1.2.5 A Variable's Value Can Change](#))
- With two `int` values, `/` keeps the whole-number part and `%` gives the remainder. ([Topic 1.3: Expressions and Output](#))
- Multiplication, division, and remainder happen before addition and subtraction. ([Topic 1.3: Expressions and Output, 1.3.8 Operator Precedence](#))
- An assignment works out the right side first, then replaces the variable's old value. ([Topic 1.4: Assignment Statements and Input, 1.4.2 The Right Side Is Evaluated First](#))
- A variable must be given a value before it can be used. ([Topic 1.4: Assignment Statements and Input, 1.4.4 Assign Before You Use](#))
- Java widens an `int` to a `double` automatically, but a `double` to an `int` needs a cast. ([Topic 1.5: Casting and Range of Variables](#))

1.6.1 Adding to a Variable in One Step

Programs change the same variable over and over. A game adds points to a score each time you hit a target. A shopping cart adds each item's price to a running total. A bank app subtracts each purchase from a balance. In each case the new value depends on the old one.

You can already do this with the assignment statement from Topic 1.4:

```
1 score = score + 10;
```

Recall how Java runs that line. It works out the right side first, using the value `score` holds right now, and only then stores the result back into `score`, replacing the old value. If `score` held 40, the right side is `40 + 10`, which is 50, and 50 goes into `score`.

That works, but notice that the variable name appears twice. In a long program with names like `totalPointsThisRound`, writing the name twice is tiring and gives you two chances to mistype it. Updating a variable from its own value is so common that Java gives it a shorter form:

```
1 score += 10;
```

Read `+=` as "add to." The line means "add 10 to `score`." It does exactly what `score = score + 10;` does: Java reads the current value of `score`, adds 10, and stores the result back in `score`.

The `+=` symbol is a **compound assignment operator**: an operator that does an arithmetic operation between a variable's current value and the value on its right, then stores the result back in that same variable. "Compound" means it combines two jobs, arithmetic and assignment, in one symbol.

Look at each piece of `score += 10;`:

Piece	What it is
<code>score</code>	The variable to update. Its current value is used, then replaced.
<code>+=</code>	Add the right side to the variable, then store the result in it.
<code>10</code>	The amount to add. It can be a literal, a variable, or expression.
<code>;</code>	Ends the statement, as always.

The left side must be a variable. A compound operator stores a result, and the only place Java can store a value is in a variable, the same reason `=` needs a variable on its left in Topic 1.4. The order of the two symbols matters: plus first, then equals.

KEY IDEA

`score += 10;` means the same as `score = score + 10;`: add to the variable's current value, then store the result back in that variable.

WORKED EXAMPLE Adding to a score twice

Watch what `score` holds after each line. Each `+=` starts from whatever `score` holds at that moment, not from its starting value.

```
1 int score = 40;
2 score += 10;
3 System.out.println(score);
4 score += 25;
5 System.out.println(score);
```

Line	What Java does	score	Printed
1	Declares <code>score</code> and stores 40.	40	
2	Reads 40, adds 10, stores 50.	50	
3	Prints the current value.	50	50
4	Reads 50 (not 40), adds 25, stores 75.	75	
5	Prints the current value.	75	75

Output:

```
50
75
```

Line 4 is the one to study. The `40` from line 1 is gone; line 2 replaced it. So the second update builds on `50`.

Now vary two things: the variable is a `double`, and the amount added is another variable instead of a literal.

WORKED EXAMPLE A running total with a variable on the right

A cart adds the price of two snacks at `3.5` each and a drink at `1.25`.

```
1 double total = 0.0;
2 double price = 3.5;
3 total += price;
4 total += price;
5 total += 1.25;
6 System.out.print("Total: ");
7 System.out.println(total);
```

Line	What Java does	total	price
1	Stores <code>0.0</code> .	0.0	
2	Stores <code>3.5</code> .	0.0	3.5
3	<code>0.0 + 3.5</code> is <code>3.5</code> ; store it.	3.5	3.5
4	<code>3.5 + 3.5</code> is <code>7.0</code> ; store it.	7.0	3.5
5	<code>7.0 + 1.25</code> is <code>8.25</code> ; store it.	8.25	3.5

Lines 6 and 7 print the label with `print`, which stays on the same line, then the number with `println`:

```
Total: 8.25
```

Notice that `price` never changes. Only the variable on the **left** of `+=` is updated; the right side is only read.

WATCH OUT

Writing `+=` instead of `++`. The two look almost the same, and both compile, so this slip is easy to miss:

```
1 int score = 40;
2 score += 10;
3 System.out.println(score);
```

This prints `10`, not `50`. Java reads `+= 10` as a plain `=` followed by the number `+10`, a positive ten, the same way `-10` is a negative ten. So line 2 means `score = +10;`, which throws away the `40` and stores `10`. No error message warns you, because assigning `+10` is legal Java.

The fix is to put the arithmetic symbol first: `score ++ 10;`. A memory hook: the operator does arithmetic **then** assignment, so the arithmetic symbol comes first.

PAUSE AND TRY**Question 1** Tracing

What does this code print? Trace the value of `coins` after each line.

```
1 int coins = 12;
2 coins += 8;
3 coins += coins;
4 System.out.println(coins);
```

Question 2 Multiple choice

The variable `level` holds `6`. Which statement has the same effect as `level = level + 3`?

- A. `level += 3`;
- B. `level =+ 3`;
- C. `level = 3`;
- D. `3 += level`;

1.6.2 Subtract, Multiply, Divide, and Remainder

Adding is not the only update programs need. A balance goes down when you pay for something. A prize doubles. A bill is split among friends. Java has a compound operator for each of the five arithmetic operators you learned in Topic 1.3, and every one works the same way as `+=`: do the arithmetic between the variable's current value and the right side, then store the result back in the variable.

Operator	Means	x starts at 20: statement	x afterward
<code>+=</code>	<code>x = x + value</code>	<code>x += 6;</code>	26
<code>-=</code>	<code>x = x - value</code>	<code>x -= 6;</code>	14
<code>*=</code>	<code>x = x * value</code>	<code>x *= 6;</code>	120
<code>/=</code>	<code>x = x / value</code>	<code>x /= 6;</code>	3
<code>%=</code>	<code>x = x % value</code>	<code>x %= 6;</code>	2

Here is that table as code, with each operator starting from a fresh variable that holds `20`:

```

1 int plus = 20;
2 plus += 6;
3 System.out.println(plus);
4 int minus = 20;
5 minus -= 6;
6 System.out.println(minus);
7 int times = 20;
8 times *= 6;
9 System.out.println(times);
10 int divide = 20;
11 divide /= 6;
12 System.out.println(divide);
13 int remainder = 20;
14 remainder %= 6;
15 System.out.println(remainder);

```

It prints 26, 14, 120, 3, and 2, one per line. Look closely at the last two. Because `divide` and `6` are both `int` values, `/=` follows the integer division rule from Topic 1.3: `20 / 6` is `3.333...`, and Java keeps only the whole-number part, `3`. And `%=` stores the remainder: `6` goes into `20` three times (that is 18), leaving `2`.

KEY IDEA

Each compound operator (`+=`, `-=`, `*=`, `/=`, `%=`) applies its arithmetic to the variable's current value and the right side, then stores the result back in the variable. The usual `int` and `double` rules from Topic 1.3 still apply.

WORKED EXAMPLE Several updates in a row

In a game, a player starts with 50 gems, spends 8, doubles the rest with a bonus, trades them five-for-one, and then keeps only the remainder after groups of five. Each line updates the same variable, so each one starts from the result of the line before.

```

1 int gems = 50;
2 gems -= 8;
3 gems *= 2;
4 gems /= 5;
5 gems %= 5;
6 System.out.print("Gems: ");
7 System.out.println(gems);

```

Line	Arithmetic	gems
1	Store 50	50
2	<code>50 - 8</code>	42
3	<code>42 * 2</code>	84
4	<code>84 / 5 → 16</code>	16
5	<code>16 % 5</code>	1

Line 4 is integer division: $84 / 5$ is 16.8 on a calculator, but both values are `int`, so Java keeps 16. Line 5 asks how much is left after taking out as many fives as possible from 16: three fives make 15, leaving 1. The output is:

```
Gems: 1
```

Now change one thing: the variable is a `double`. Watch what that does to `/=`.

WORKED EXAMPLE The same kinds of updates with a double

A savings balance starts at 80.0. A purchase of 12.5 comes out, a matching gift doubles it, and the result is split four ways.

```
1 double balance = 80.0;
2 balance -= 12.5;
3 balance *= 2;
4 balance /= 4;
5 System.out.print("Balance: ");
6 System.out.println(balance);
```

Line	Arithmetic	balance
1	Store 80.0	80.0
2	$80.0 - 12.5$	67.5
3	$67.5 * 2$	135.0
4	$135.0 / 4$	33.75

```
Balance: 33.75
```

On line 3 the right side is the `int` 2, but `balance` is a `double`, so the multiplication uses at least one `double` and gives a `double`, as in Topic 1.3. On line 4 the same is true for `/=`, so nothing is thrown away: 33.75 keeps its decimal part. Compare this with line 4 of the gem example, where both values were `int` and the decimal part was dropped. The operator is the same; the types decide the result.

WATCH OUT

Expecting `int` arithmetic to keep a decimal. Suppose a price stored as an `int` goes up by half:

```
1 int price = 15;
2 price *= 1.5;
3 System.out.println(price);
```

You might expect 22.5, but this prints 22. Here is what happens. $15 * 1.5$ is worked out as a `double`, 22.5. But `price` is an `int` and cannot hold 22.5. A compound operator quietly converts its result to the variable's type, just like an `(int)` cast from Topic 1.5, so the decimal part is chopped off and 22 is stored.

That hidden conversion is the one place where `x *= value` and `x = x * value` behave differently. Written the long way, the same update does not even compile:

```
1 int price = 15;
2 price = price * 1.5;
3 System.out.println(price);
```

error: incompatible types: possible lossy conversion from double to int

With plain `=`, Java refuses to squeeze a `double` into an `int` without a cast you write yourself. With `*=`, it does the conversion for you without a warning, which makes the lost `.5` easy to miss. (Exact error wording can vary between Java versions.)

The fix is to choose the right type in the first place. If a value can have a decimal part, store it in a `double`:

```
1 double price = 15.0;
2 price *= 1.5;
3 System.out.println(price);
```

This prints `22.5`.

PAUSE AND TRY

Question 3 Tracing

What does this code print? Show the value of `pages` after each line.

```
1 int pages = 45;
2 pages -= 9;
3 pages /= 5;
4 pages %= 4;
5 System.out.println(pages);
```

Question 4 Multiple choice

After these statements run, what does `slices` hold?

```
1 int slices = 7;
2 slices /= 2;
3 System.out.println(slices);
```

- A. 3
- B. 3.5
- C. 1
- D. 4

1.6.3 The Right Side Is Evaluated First

So far the right side of every compound operator has been a single number or variable. It can also be a whole expression. Suppose a store has 10 boxes of a product and each one is replaced by a pack of $2 + 3$ boxes:

```
1 stock *= 2 + 3;
```

Which happens: $stock * 2$, then $+ 3$? Or $stock * (2 + 3)$? Topic 1.3 taught that $*$ happens before $+$, which might suggest the first reading. But that is not how compound operators work.

Java first works out the **entire** right side, down to a single value. Only then does it combine that value with the variable. So `stock *= 2 + 3;` means `stock = stock * (2 + 3);`, as if the right side were wrapped in parentheses. This is the same order as a plain assignment in Topic 1.4: the right side is finished before anything is stored. The operator is "multiply the variable by whatever the right side turns out to be."

KEY IDEA

In `variable op= expression`, Java evaluates the whole expression first and then combines it with the variable. It means `variable = variable op (expression)`, with parentheses around the right side.

WORKED EXAMPLE Multiplying by a sum

```
1 int stock = 10;
2 stock *= 2 + 3;
3 System.out.println(stock);
```

Step	What Java does	stock
1	Line 1 stores 10.	10
2	Line 2: evaluate the right side, $2 + 3$, to get 5.	10
3	Combine: $10 * 5$ is 50; store it.	50

It prints 50. If Java had done $stock * 2 + 3$, the answer would have been 23. It is not.

The rule matters most with `-=` and `/=`, because subtraction and division give different answers when the grouping changes.

WORKED EXAMPLE Subtracting and dividing by an expression

A theater has 30 seats; a group of 4 and a group of 6 arrive. Separately, 100 points are shared equally among $2 * 5$ players.

```
1 int seats = 30;
2 seats -= 4 + 6;
3 System.out.println(seats);
4 int points = 100;
5 points /= 2 * 5;
6 System.out.println(points);
```

Line	Right side first	Then combine	Result
------	------------------	--------------	--------

2	4 + 6 is 10	30 - 10	20
5	2 * 5 is 10	100 / 10	10

20
10

Both groups are subtracted: 30 - 10 is 20. On line 5, Java does not do 100 / 2 and then * 5 (that would give 250); it divides by the whole right side, 10.

WATCH OUT

Expanding a compound operator without parentheses. When students rewrite `seats -= 4 + 6`; the long way, they often drop the parentheses:

```
1 int seatsA = 30;
2 seatsA = seatsA - 4 + 6;
3 System.out.println(seatsA);
4 int seatsB = 30;
5 seatsB = seatsB - (4 + 6);
6 System.out.println(seatsB);
```

32
20

The first version, `seatsA - 4 + 6`, runs left to right: 30 - 4 is 26, then 26 + 6 is 32. The 6 is added instead of subtracted. That is a different calculation from `seats -= 4 + 6`. The second version keeps the right side together with parentheses and gives the correct 20. When you translate a compound operator into a plain assignment, always put parentheses around the old right side.

PAUSE AND TRY

Question 5 Tracing

What does this code print? Show both steps for each update.

```
1 int width = 20;
2 width -= 2 + 3;
3 width /= 1 + 2;
4 System.out.println(width);
```

Question 6 Short answer

The variable `cost` holds 10. One student writes `cost *= 1 + 2`; and another writes `cost = cost * 1 + 2`;, thinking they mean the same thing. What does `cost` hold after each statement? Explain why they differ.

1.6.4 Adding or Subtracting One

One update is more common than all the others: changing a variable by exactly one. A program counts laps, visitors, or correct answers one at a time; a game takes away one life. You could write `laps += 1;`, and that works. Counting by one is so common, though, that Java has an even shorter form.

The **post-increment operator** `++` adds 1 to the value stored in a variable. The **post-decrement operator** `--` subtracts 1. "Increment" means increase by a step and "decrement" means decrease by a step; "post" means the operator is written **after** the variable name. These three statements all do the same thing:

Statement	Effect on <code>laps</code>
<code>laps = laps + 1;</code>	adds 1
<code>laps += 1;</code>	adds 1
<code>laps++;</code>	adds 1

and in the same way, `lives--;` does what `lives -= 1;` and `lives = lives - 1;` do. Like the other compound operators, `++` and `--` read the variable's current value, change it, and store the new value back in the same variable. There is no right side, because the amount is always 1.

KEY IDEA

`count++;` adds 1 to `count` and stores the result in `count`. `count--;` subtracts 1. Write them after the variable, as a statement on their own line.

WORKED EXAMPLE Counting laps and losing a life

```

1 int laps = 0;
2 laps++;
3 laps++;
4 laps++;
5 System.out.print("Laps: ");
6 System.out.println(laps);
7 int lives = 3;
8 lives--;
9 System.out.print("Lives: ");
10 System.out.println(lives);

```

Line	What Java does	laps	lives
1	Stores 0.	0	
2	0 + 1; store it.	1	
3	1 + 1; store it.	2	
4	2 + 1; store it.	3	
7	Stores 3.	3	3
8	3 - 1; store it.	3	2

Lines 5–6 and 9–10 print a label with `print` and the value with `println`:

```
Laps: 3  
Lives: 2
```

Starting a counter at 0 and adding one each time something happens is a pattern you will use constantly.

Now vary the type. `++` and `--` work on any number variable, including a `double`. They still change it by exactly one; they do not move to the next whole number.

```
1 double temperature = 71.5;  
2 temperature++;  
3 System.out.println(temperature);  
4 temperature--;  
5 temperature--;  
6 System.out.println(temperature);
```

```
72.5  
70.5
```

71.5 plus one is 72.5. Then two decrements take it down to 70.5. The decimal part is untouched.

One form only. Java also lets you write the plus signs in front of the variable (`++laps`), and it lets you put `laps++` inside a bigger expression, such as in the middle of another calculation or a print statement. Both of those uses follow extra rules that are easy to misread, and neither is part of this course or the AP exam. In this course, `++` and `--` always come after the variable, in a statement by themselves, like `laps++`. The same goes for every compound operator: use each one as its own statement.

WATCH OUT

Writing an expression that never stores anything. A student who wants to add one sometimes writes `laps + 1`; thinking it updates `laps`:

```
1 int laps = 0;  
2 laps + 1;  
3 System.out.println(laps);
```

```
error: not a statement
```

`laps + 1` only works out a value, 1; it says nothing about where to put it. Java will not compile a line that calculates a number and then throws it away, so it reports that this is not a statement. The fix is any form that stores: `laps++`, `laps += 1`, or `laps = laps + 1`.

WATCH OUT

Updating a variable that has no value yet. Every compound operator, including `++`, starts by reading the variable's current value. Topic 1.4 said a variable must be given a value before it can be used, and that applies here too:

```
1 int laps;  
2 laps++;  
3 System.out.println(laps);
```

error: variable laps might not have been initialized

Line 1 declares `laps` but never stores a value in it, so there is nothing for line 2 to add 1 to. Fix it by giving the counter a starting value when you declare it: `int laps = 0;`.

PAUSE AND TRY

Question 7 Tracing

What does this code print? Trace `lives` line by line.

```
1 int lives = 3;  
2 lives--;  
3 lives--;  
4 lives++;  
5 lives -= 1;  
6 System.out.println(lives);
```

Question 8 Debugging

This code should use one stamp and report how many are left, 4. Instead it prints Stamps left: -1. Find the error, explain why it gives -1, and correct the line.

```
1 int stamps = 5;  
2 stamps -= 1;  
3 System.out.print("Stamps left: ");  
4 System.out.println(stamps);
```

Putting it together

This program follows one round of an arcade game and uses every operator from the chapter. Before reading the trace, try to predict the four lines of output yourself.

WORKED EXAMPLE One round of an arcade game

ArcadeRound.java

```

1 public class ArcadeRound
2 {
3     public static void main(String[] args)
4     {
5         int score = 0;
6         int lives = 3;
7         int tokens = 20;
8         score += 150;
9         score *= 2;
10        lives--;
11        tokens -= 4;
12        score -= 25 + 15;
13        lives++;
14        int prizes = tokens;
15        prizes /= 3;
16        tokens %= 3;
17        System.out.print("Score: ");
18        System.out.println(score);
19        System.out.print("Lives: ");
20        System.out.println(lives);
21        System.out.print("Prizes: ");
22        System.out.println(prizes);
23        System.out.print("Tokens left: ");
24        System.out.println(tokens);
25    }
26 }

```

Lines 5–7 set the starting values: score is 0, lives is 3, and tokens is 20. Then each update changes one variable, starting from that variable's latest value:

Line	Statement	Work	score	lives	tokens	prizes
8	score += 150;	0 + 150	150	3	20	
9	score *= 2;	150 * 2	300	3	20	
10	lives--;	3 - 1	300	2	20	
11	tokens -= 4;	20 - 4	300	2	16	
12	score -= 25 + 15;	right side 40, then 300 - 40	260	2	16	
13	lives++;	2 + 1	260	3	16	
14	int prizes = tokens;	copy 16	260	3	16	16
15	prizes /= 3;	16 / 3 → 5 (integer division)	260	3	16	5
16	tokens %= 3;	16 % 3	260	3	1	5

Lines 17–24 print each label with `print` and each value with `println`:

```

Score: 260
Lives: 3
Prizes: 5
Tokens left: 1

```

Three details are worth a second look. Line 12 subtracts the whole right side, 40, not 25 with 15 added back. Lines 15 and 16 together split 16 tokens into 5 prizes of 3 tokens each, with 1 token left over: /= keeps the whole-number part and %= keeps the remainder. And line 14 copies the value of tokens into prizes, so changing prizes on line 15 does not change tokens.

Vocabulary

Term	Definition	Example
compound assignment operator	An operator that does arithmetic between a variable's current value and the right side, then stores the result back in that variable: +=, -=, *=, /=, %=	total += 5;
post-increment operator	++ written after a variable; adds 1 to the value stored in it	laps++;
post-decrement operator	-- written after a variable; subtracts 1 from the value stored in it	lives--;

Chapter summary

- A compound assignment operator updates a variable from its own current value in one step: `x += 5;` means `x = x + 5;`. There is one for each arithmetic operator: +=, -=, *=, /=, and %=. The left side must be a variable.
- Each update starts from the value the variable holds at that moment, so a series of updates must be traced line by line.
- The whole right side is evaluated first, then combined with the variable: `x -= a + b;` means `x = x - (a + b);`.
- The usual type rules apply: with two `int` values, /= drops the decimal part and %= keeps the remainder. If the result is a `double` but the variable is an `int`, the decimal part is silently chopped off.
- `x++;` adds 1 and `x--;` subtracts 1. Write them after the variable, as a statement by themselves.
- `=+` and `=-` are not compound operators; they assign a positive or negative value and compile without any warning.

Review questions

Question 9 Multiple choice

What does this code print?

```
1 double average = 7;  
2 average /= 2;  
3 System.out.println(average);
```

- A. 3
- B. 3.5

- C. 3.0
- D. Nothing; `double average = 7;` does not compile.

Question 10 Tracing

This code updates two variables, and some right sides use the other variable. Write the exact output.

```
1 int apples = 9;
2 int baskets = 2;
3 apples += baskets * 3;
4 baskets++;
5 apples %= baskets + 1;
6 baskets *= apples;
7 apples--;
8 System.out.print("Apples: ");
9 System.out.println(apples);
10 System.out.print("Baskets: ");
11 System.out.println(baskets);
```

Question 11 Debugging

This code should turn 200 seconds into whole minutes and leftover seconds, printing `Minutes: 3` and `Seconds: 20`. Instead it prints `Minutes: 20` and `Seconds: 3`. Find the error and correct it.

```
1 int seconds = 200;
2 int minutes = seconds;
3 minutes %= 60;
4 seconds /= 60;
5 System.out.print("Minutes: ");
6 System.out.println(minutes);
7 System.out.print("Seconds: ");
8 System.out.println(seconds);
```

Question 12 Multiple choice

A student expects this code to print 7, since halving a number and then doubling it should give the number back. What does it actually print?

```
1 int score = 7;
2 score /= 2;
3 score *= 2;
4 System.out.println(score);
```

- A. 7
- B. 6
- C. 7.0
- D. 8

Question 13 Short answer

`distance` is an `int` that holds 7. The statement `distance *= 2.5;` compiles, but `distance = distance * 2.5;` does not. Explain why, and say what `distance` holds after `distance *= 2.5;`.

Reflection

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

- ☐ I can explain what `+=` does and why `=+` is a different thing. Reread: 1.6.1 Adding to a Variable in One Step
- ☐ I can trace a series of `-=`, `*=`, `/=`, and `%=` updates, including integer division. Reread: 1.6.2 Subtract, Multiply, Divide, and Remainder
- ☐ I can work out the whole right side before combining it with the variable. Reread: 1.6.3 The Right Side Is Evaluated First
- ☐ I can use `++` and `--` to add or subtract one, as a statement on its own. Reread: 1.6.4 Adding or Subtracting One

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

It prints 40.

Line	What Java does	coins
1	Stores 12.	12
2	12 + 8 is 20; store it.	20
3	The right side is coins, which is now 20. 20 + 20 is 40; store it.	40

Line 3 uses the same variable on both sides. That is fine: Java reads the current value (20) for both, adds, and only then stores. It is the same as `coins = coins + coins;`, which doubles the value. If you got 24, you used the starting value 12 instead of the updated 20.

Reread 1.6.1 *Adding to a Variable in One Step* if your answer differed.

Question 2

A. `level += 3;` adds 3 to the current value of `level` and stores the result, so `level` becomes 9, exactly as `level = level + 3;` does.

- **B** is wrong: `=+ 3` is a plain assignment of positive three, so `level` becomes 3. The symbols are in the wrong order.
- **C** is wrong: it replaces 6 with 3 and never uses the old value.
- **D** is wrong: it does not compile. The left side of a compound operator must be a variable, because the result has to be stored somewhere, and 3 is a value, not a variable. Java reports `unexpected type (required: variable, found: value)`.

Reread 1.6.1 *Adding to a Variable in One Step* if you chose differently.

Question 3

It prints 3.

Line	Arithmetic	pages
1	Store 45	45
2	45 - 9	36
3	36 / 5 → 7	7
4	7 % 4	3

Line 3 is integer division: `36 / 5` is 7.2, and Java keeps 7. Line 4: one 4 fits into 7, leaving 3. If you got 0 at the end, you probably used `36` on line 4 (`36 % 4` is 0); each line must start from the value the line before it stored.

Reread 1.6.2 *Subtract, Multiply, Divide, and Remainder* if your answer differed.

Question 4

A. `slices /= 2;` means `slices = slices / 2;`. Both 7 and 2 are `int`, so this is integer division, which keeps only the whole-number part of 3.5: 3.

- **B** is wrong: `3.5` is the calculator answer, but integer division drops the decimal part, and an `int` variable cannot hold `3.5` anyway.
- **C** is wrong: `1` is the remainder, `7 % 2`. That would be `slices %= 2;`
- **D** is wrong: integer division does not round up; it drops the decimal part.

Reread 1.6.2 *Subtract, Multiply, Divide, and Remainder* if you chose differently.

Question 5

It prints `5`.

Line	Right side first	Then combine	width
1		Store <code>20</code>	<code>20</code>
2	<code>2 + 3</code> is <code>5</code>	<code>20 - 5</code>	<code>15</code>
3	<code>1 + 2</code> is <code>3</code>	<code>15 / 3</code>	<code>5</code>

If you got `21` after line 2, you computed `20 - 2 + 3`, which subtracts 2 and then adds 3 back. Carrying that mistake into line 3 gives `21 / 1 + 2`, which is `23`. A compound operator always finishes its right side first: subtract 5, then divide by 3.

Reread 1.6.3 *The Right Side Is Evaluated First* if your answer differed.

Question 6

Model answer: After `cost *= 1 + 2;`, `cost` holds `30`. Java evaluates the whole right side first, `1 + 2` is `3`, and then multiplies: `10 * 3`. After `cost = cost * 1 + 2;`, `cost` holds `12`, because in a normal expression `*` happens before `+`: `10 * 1` is `10`, then `10 + 2` is `12`. They differ because a compound operator acts like it has parentheses around its right side, `cost = cost * (1 + 2)`, and the second student left the parentheses out.

A complete answer must:

- give both values, `30` and `12`,
- say that the compound operator evaluates its whole right side before multiplying, and
- say that the long form needs parentheses around `1 + 2` to match.

Reread 1.6.3 *The Right Side Is Evaluated First* if your answer differed.

Question 7

It prints `1`.

Line	Update	<code>lives</code>
1	Store <code>3</code>	<code>3</code>
2	<code>3 - 1</code>	<code>2</code>
3	<code>2 - 1</code>	<code>1</code>
4	<code>1 + 1</code>	<code>2</code>
5	<code>2 - 1</code>	<code>1</code>

`lives -= 1;` on line 5 has the same effect as `lives--;`. Each statement changes the value by exactly one.

Reread 1.6.4 *Adding or Subtracting One* if your answer differed.

Question 8

The error is on line 2. `--` is not a compound operator. Java reads `stamps -= 1;` as `stamps = -1;`: a plain assignment of negative one. The `5` is thrown away and `-1` is stored, which is why the output is `-1`. It compiles because assigning `-1` is legal, so only the wrong output reveals the bug. It is the subtraction version of the `+=` slip.

Corrected code, using the post-decrement operator (`stamps -= 1;` would also work):

```
1 int stamps = 5;
2 stamps--;
3 System.out.print("Stamps left: ");
4 System.out.println(stamps);
```

This prints `Stamps left: 4`.

Reread [1.6.4 Adding or Subtracting One](#) and the Watch Out in [1.6.1 Adding to a Variable in One Step](#) if you missed it.

Question 9

B. The variable is a `double`. On line 1, Java widens the `int 7` to the `double 7.0` automatically (Topic 1.5). On line 2, `average /= 2;` divides `7.0` by `2`; with at least one `double`, the division keeps the decimal part, so `3.5` is stored and printed.

- **A** is wrong: that would be integer division, which happens only when both values are `int`. `average` holds a `double`.
- **C** is wrong: it treats the division as integer division (`3`) and then shows it as a `double`. The decimal part is never dropped here.
- **D** is wrong: an `int` value fits in a `double` variable without a cast; Java widens it.

Reread [1.6.2 Subtract, Multiply, Divide, and Remainder](#) if you chose differently.

Question 10

```
Apples: 2
Baskets: 9
```

Line	Right side first	Then combine	apples	baskets
1-2			9	2
3	baskets * 3 is 6	9 + 6	15	2
4		2 + 1	15	3
5	baskets + 1 is 4	15 % 4	3	3
6	apples is 3	3 * 3	3	9
7		3 - 1	2	9

Line 5 uses the updated `baskets` (`3`, after line 4), and the whole right side `3 + 1` is worked out before the remainder: `15 % 4` is `3` because three fours make `12`. Line 6 changes `baskets` using the current `apples`; `apples` itself is only read there.

Reread [1.6.3 The Right Side Is Evaluated First](#) and [1.6.4 Adding or Subtracting One](#) if your output differed.

Question 11

The operators on lines 3 and 4 are swapped. Whole minutes come from integer division by 60, and leftover seconds come from the remainder after dividing by 60. The code does the opposite: line 3 stores `200 % 60`, which is 20, in `minutes`, and line 4 stores `200 / 60`, which is 3, in `seconds`. Both lines compile, so only the output shows the mistake.

Corrected code:

```
1 int seconds = 200;
2 int minutes = seconds;
3 minutes /= 60;
4 seconds %= 60;
5 System.out.print("Minutes: ");
6 System.out.println(minutes);
7 System.out.print("Seconds: ");
8 System.out.println(seconds);
```

Now line 3 stores `200 / 60`, which is 3 (integer division drops .333...), and line 4 stores `200 % 60`, which is 20 (three minutes use 180 seconds, leaving 20). It prints:

```
Minutes: 3
Seconds: 20
```

Line 2 copies the value 200 into `minutes` before either update, so each variable starts from 200.

Reread 1.6.2 *Subtract, Multiply, Divide, and Remainder* if you missed it.

Question 12

B. Line 2 is integer division because `score` and 2 are both `int`: `7 / 2` is 3, and the .5 is gone for good. Line 3 doubles what is left: `3 * 2` is 6.

- **A** is wrong: halving then doubling gives the original only if nothing is lost. Integer division lost the .5, and `*=` cannot bring it back.
- **C** is wrong: every value here is an `int`, so nothing prints with .0.
- **D** is wrong: integer division never rounds up; `7 / 2` is 3, not 4.

Reread 1.6.2 *Subtract, Multiply, Divide, and Remainder* if you chose differently.

Question 13

Model answer: In both versions, `7 * 2.5` is worked out as a `double`, 17.5, because one of the values is a `double`. With plain `=`, Java will not put a `double` into an `int` variable without a cast, since the decimal part would be lost, so it reports a possible lossy conversion and does not compile. A compound operator converts its result to the variable's type automatically, like an (`int`) cast, so `distance *= 2.5`; compiles and truncates 17.5 to 17. After it runs, `distance` holds 17.

A complete answer must:

- say that `7 * 2.5` is the `double` 17.5,
- explain that plain assignment needs an explicit cast to store a `double` in an `int`, but a compound operator converts automatically, and
- give the stored value, 17, with the decimal part dropped, not rounded.

Reread the Watch Out in 1.6.2 *Subtract, Multiply, Divide, and Remainder* if your answer differed.