

1.5 Casting and Range of Variables

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Predict when Java turns an `int` into a `double` on its own, and explain why it never turns a `double` into an `int` on its own.
- Use the `(int)` and `(double)` casts in expressions and predict the exact value and type each one produces.
- Round a `double` to the nearest whole number, for positive and for negative values.
- State the smallest and largest values an `int` can hold, and explain what happens when an `int` calculation goes past them.
- Explain why some `double` results come out slightly wrong, and when `int` values avoid the problem.

What to remember from earlier

- An `int` holds a whole number and a `double` can have a decimal part. ([Topic 1.2: Variables and Data Types, 1.2.3 The Three Primitive Types: int, double, and boolean](#))
- Dividing two `int` values keeps only the whole-number part, so `7 / 2` is 3. ([Topic 1.3: Expressions and Output, 1.3.6 Integer Division](#))
- Multiplication, division, and remainder happen before addition and subtraction, and parentheses go first. ([Topic 1.3: Expressions and Output, 1.3.8 Operator Precedence](#))
- An assignment works out the right side and stores it in the variable on the left. ([Topic 1.4: Assignment Statements and Input, 1.4.2 The Right Side Is Evaluated First](#))
- The value assigned must be of a type the variable can hold. ([Topic 1.2: Variables and Data Types, 1.2.6 A Variable Holds Only Values of Its Type](#))
- A logic error lets the program run but gives a wrong result. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.7 Logic Errors and Testing](#))

1.5.1 Java Widens an `int` to a `double`

Real programs mix whole numbers and decimals all the time. A runner counts laps as whole numbers, but each lap is 0.25 miles. A shop stores a price of 5 dollars, but wants it in a variable that could later hold 4.99. So far you have kept `int` values and `double` values apart. This chapter is about moving a value from one type to the other, and about the limits of each type.

Start with the easy direction. Every whole number is also a number that could have a decimal part: 5 is the same amount as 5.0. Turning 5 into 5.0 loses nothing. Because nothing can go wrong, Java does this conversion for you, without being asked. This automatic conversion from `int` to `double` is called **widening**: the value moves into a "wider" type that can hold everything the narrower one can, and more.

Widening happens in two places you already know.

1. **Assignment.** When you store an `int` value in a `double` variable, Java widens the value first. This is what makes an `int` value compatible with a `double` variable, in the sense of Topic 1.4's rule that a stored value must have a compatible type.
2. **Arithmetic with one of each.** Topic 1.3 said that an operation with at least one `double` gives a `double`. Now you can see how: Java widens the `int` operand to a `double`, then does the operation on two `double` values.

KEY IDEA

Java automatically widens an `int` to a `double` whenever a `double` is needed. The value stays the same; only the type changes, so 5 becomes 5.0.

WORKED EXAMPLE Storing a whole-number price in a double

Watch the type of each printed value.

```
1 int wholePrice = 5;
2 double price = wholePrice;
3 System.out.println(wholePrice);
4 System.out.println(price);
```

```
5
5.0
```

- Line 1 stores the `int` 5 in `wholePrice`.
- Line 2 reads the value of `wholePrice`, which is the `int` 5. The variable on the left is a `double`, so Java widens the value to 5.0 and stores that in `price`. `wholePrice` itself is not changed; it is still an `int` holding 5.
- Lines 3 and 4 print each variable. The `int` prints without a decimal point and the `double` prints with one.

Now change the situation: instead of an assignment, the `int` is one operand of an arithmetic operation.

```
1 int laps = 7;
2 double lapLength = 0.25;
3 double distance = laps * lapLength;
4 System.out.println(distance);
5 System.out.println(laps / 2);
6 System.out.println(laps / 2.0);
```

```
1.75
3
3.5
```

Line	Expression	What Java does	Result
3	<code>laps * lapLength</code>	One operand is a <code>double</code> , so 7 is widened to 7.0; 7.0×0.25	1.75
5	<code>laps / 2</code>	Both operands are <code>int</code> , so nothing is widened; whole part only	3
6	<code>laps / 2.0</code>	2.0 is a <code>double</code> , so 7 is widened to 7.0; $7.0 \div 2.0$	3.5

Lines 5 and 6 differ by one character, but that character decides whether widening happens, and widening decides which kind of division Java does.

Widening happens **one operation at a time**, not for a whole line at once. Java looks at each operator and its two operands, in the order Topic 1.3 described.

```
1 System.out.println(7 / 2 * 1.0);
2 System.out.println(7 * 1.0 / 2);
```

```
3.0
3.5
```

In the first line, `7 / 2` is done first, and both operands are `int`, so the result is `3`. Only then does `3 * 1.0` widen `3` to `3.0`. The decimal part was already gone before any `double` appeared. In the second line, `7 * 1.0` is done first and gives `7.0`, so the division `7.0 / 2` keeps its decimal part: `3.5`.

WATCH OUT

Expecting Java to convert a `double` to an `int` on its own. Widening only goes one way. Here a student stores a race length in an `int`:

NoAutoNarrow.java

```
1 public class NoAutoNarrow
2 {
3     public static void main(String[] args)
4     {
5         double raceLength = 26.2;
6         int miles = raceLength;
7         System.out.println(miles);
8     }
9 }
```

```
NoAutoNarrow.java:6: error: incompatible types: possible lossy conversion from
double to int
```

An `int` has no room for the `.2`. If Java converted silently, part of the value would disappear without anyone noticing, so the compiler refuses. "Lossy" means "something could be lost." (The exact wording of compiler messages can vary slightly between Java versions.) To store the race length as it is, make `miles` a `double`. If you really do want only the whole miles, you have to say so on purpose, with a cast, which is the next subsection.

PAUSE AND TRY

Question 1 Multiple choice

Each statement appears on its own in a program. Which one does **not** compile?

- A. `double average = 12;`
- B. `double total = 4 * 2.5;`
- C. `int count = 12.0;`

D. `int score = 7 / 2;`

Question 2 Tracing

What do these two lines print?

```
1 System.out.println(9 / 4 * 2.0);
2 System.out.println(9 * 2.0 / 4);
```

1.5.2 Casting a double to an int

Sometimes you really do want only the whole-number part of a `double`. A race of 26.2 miles counts as 26 complete miles. A trip of 2.25 hours includes 2 full hours. As the last Watch Out showed, Java will not drop the decimal part on its own, because that might be an accident. You need a way to tell the compiler, "I know part of the value will be lost, and I want that."

That instruction is a **cast**. A cast converts a value to a different type. You write the type you want in parentheses directly in front of the value:

```
1 (int) raceLength
```

- `(int)` is the **casting operator**. The parentheses and the type name together form the operator; the type in the middle says what to convert to.
- `raceLength` is the value being converted. The cast produces a new `int` value. It does **not** change the variable `raceLength`, which is still a `double`.

When the cast turns a `double` into an `int`, Java **truncates** the value: it chops off every digit after the decimal point and keeps the whole-number part. It does not round. 26.2 becomes 26, and 26.9 also becomes 26.

KEY IDEA

`(int)` turns a `double` into an `int` by truncating: everything after the decimal point is dropped, with no rounding.

WORKED EXAMPLE Whole miles in a marathon

```
1 double raceLength = 26.2;
2 int miles = (int) raceLength;
3 System.out.println(raceLength);
4 System.out.println(miles);
```

```
26.2
26
```

- Line 1 stores 26.2 in the `double` variable `raceLength`.
- Line 2 reads 26.2, and the cast `(int)` truncates it to the `int` 26. That `int` is stored in `miles`. The compiler is satisfied because you asked for the conversion.

- Line 3 prints 26.2: the cast made a new value and left `raceLength` alone.
- Line 4 prints 26.

Now vary the values, including negative ones. Truncating means "drop the decimal part," so the result always moves **toward zero**.

```
1 System.out.println((int) 7.99);
2 System.out.println((int) -7.99);
3 System.out.println((int) 0.6);
4 System.out.println((int) -0.6);
```

```
7
-7
0
0
```

Value	Digits after the point	What is left	Moved toward zero?
7.99	.99	7	Yes, down to 7
-7.99	.99	-7	Yes, up to -7
0.6	.6	0	Yes
-0.6	.6	0	Yes

Look closely at `-7.99`. On a number line, the whole number just below `-7.99` is `-8`, but truncation does not go "down"; it just erases the decimal digits, leaving `-7`. A good way to remember it: for positive numbers truncation goes down, for negative numbers it goes up, and either way it goes toward zero.

WATCH OUT

Expecting (int) to round. A student wants a grade of 89.9 to count as 90:

```
1 double average = 89.9;
2 int grade = (int) average;
3 System.out.println(grade);
```

```
89
```

The program runs, but the result is a logic error for this purpose. Students expect rounding because that is what people usually do with 89.9 in everyday life. The cast only truncates, so `.9` is thrown away. To round to the nearest whole number, you need one more step, taught in 1.5.4.

PAUSE AND TRY

Question 3 Tracing

What does each line print?

```
1 System.out.println((int) 12.5);
2 System.out.println((int) -3.2);
3 System.out.println((int) 0.999);
```

Question 4 Debugging

A student wants the number of complete hours in 135 minutes. The code does not compile. Explain why, and give a corrected version of line 6 that stores 2 in `wholeHours`.

HoursWrong.java

```
1 public class HoursWrong
2 {
3     public static void main(String[] args)
4     {
5         int minutes = 135;
6         int wholeHours = minutes / 60.0;
7         System.out.println(wholeHours);
8     }
9 }
```

1.5.3 Casting an int to a double

A teacher stores three quiz scores whose total is 256 points. Both the total and the count are whole numbers, so they are naturally `int` values. But the average is not a whole number, and Topic 1.3's rule gets in the way:

```
1 int total = 256;
2 int count = 3;
3 System.out.println(total / count);
```

85

Both operands are `int`, so `/` keeps only the whole-number part and the `.333...` is lost. Widening will not help by itself, because widening only happens when a `double` is already part of the operation. You need to turn one of the `int` values into a `double` **before** the division happens.

The cast `(double)` does exactly that. It works like `(int)`, but in the other direction: `(double) total` produces the `double` version of `total`'s value, `256.0`. Nothing is lost, so the result is the same number with a decimal type. No `double` is around to cause widening, so you ask for the conversion yourself.

To use a cast correctly, you need one more rule: **a cast applies only to the value immediately after it**. Java does a cast before `*`, `/`, `%`, `+`, and `-`. To cast the result of a whole expression, put the expression in parentheses.

KEY IDEA

`(double)` turns an `int` into a `double` with no loss. A cast applies only to the value right after it, so cast an operand before the division to keep the decimal part.

WORKED EXAMPLE A quiz average with the decimal part

```
1 int total = 256;
2 int count = 3;
3 double average = (double) total / count;
4 System.out.println(average);
5 System.out.println(total);
```

```
85.33333333333333
256
```

Line 3 is evaluated in this order:

Step	What Java does	Value
1	Cast first: <code>(double) total</code> turns 256 into 256.0	256.0 / count
2	256.0 / count: one operand is a double, so 3 is widened to 3.0	256.0 / 3.0
3	Divide two double values; the decimal part is kept	85.33333333333333

Line 5 prints 256: the cast made a `double` copy of the value and left `total` as an `int`. Casting the other operand instead, `total / (double) count`, works just as well, because one `double` operand is all it takes:

```
1 int total = 256;
2 int count = 3;
3 System.out.println(total / (double) count);
```

```
85.33333333333333
```

Now change one thing: move the cast so it is outside parentheses around the division.

```
1 int total = 256;
2 int count = 3;
3 double tooLate = (double) (total / count);
4 double noCast = total / count;
5 System.out.println(tooLate);
6 System.out.println(noCast);
```

```
85.0
85.0
```

In `(double) (total / count)`, the parentheses make Java divide first. `256 / 3` divides two `int` values and gives 85. Then the cast turns 85 into 85.0. The decimal part was thrown away by the division, and casting afterward cannot bring it back. The second line, with no cast at all, does the same thing: `total / count` is 85, and widening during the assignment turns it into 85.0.

WATCH OUT

Casting after the division. Students write `(double) (total / count)` because it reads like "make the average a double," and the result does have a `.0`, so it looks right at a glance. But it is still the whole-number answer: `85.0`, not `85.333...`. A cast cannot recover digits that are already gone. Cast an operand, not the result: `(double) total / count`.

PAUSE AND TRY**Question 5** Multiple choice

Given `int a = 5;` and `int b = 2;`, which expression has the value `2.5`?

- A. `(double) (a / b)`
- B. `(double) a / b`
- C. `(int) (a / 2.0)`
- D. `a / b * 1.0`

Question 6 Short answer

A classmate says, "`(double) (total / count)` and `(double) total / count` are the same, because both have a `(double)` cast." Using `total = 256` and `count = 3`, explain why they give different values.

1.5.4 Rounding to the Nearest Integer

Back to the grade of `89.9` from 1.5.2. Truncating gives `89`, but `89.9` is much closer to `90`. Rounding to the nearest whole number is something programs need constantly: scores, distances, temperatures. Java's `(int)` only truncates, so the trick is to adjust the value **before** truncating so that truncation lands on the nearest whole number.

Think about which values should round up to `4`: everything from `3.5` up to (but not including) `4.5`. If you add `0.5` to each of those, you get values from `4.0` up to `5.0`, and truncating any of them gives `4`. Values from `3.0` up to `3.5` become `3.5` up to `4.0` after adding `0.5`, and truncate to `3`. Adding `0.5` shifts the "halfway point" onto a whole number, so truncation does the rest.

For a nonnegative `double x` (zero or positive), the nearest whole number is:

```
1 (int) (x + 0.5)
```

The parentheses matter: they make Java add `0.5` first and cast the sum. Without them, the cast would apply to `x` alone (the rule from 1.5.3).

For a negative `x`, the shift goes the other way. Truncating moves negative numbers **up** toward zero, so you subtract `0.5` first:

```
1 (int) (x - 0.5)
```

KEY IDEA

To round a `double` to the nearest `int`, use `(int) (x + 0.5)` when `x` is zero or positive, and `(int) (x - 0.5)` when `x` is negative.

WORKED EXAMPLE Rounding distances

```
1 double distance = 3.7;
2 System.out.println((int) (distance + 0.5));
3 distance = 3.2;
4 System.out.println((int) (distance + 0.5));
5 distance = 2.5;
6 System.out.println((int) (distance + 0.5));
```

```
4
3
3
```

distance	distance + 0.5	Truncated
3.7	4.2	4
3.2	3.7	3
2.5	3.0	3

A value exactly halfway, like 2.5, rounds up, the way you round in math class. Notice that lines 3 and 5 reuse the variable `distance` by assigning it a new value, as in Topic 1.4.

Now change the sign. A stock price falls by 2.7 dollars, written as `-2.7`, and you want the nearest whole number of dollars, which is `-3`.

```
1 double change = -2.7;
2 System.out.println((int) (change + 0.5));
3 System.out.println((int) (change - 0.5));
4 change = -2.2;
5 System.out.println((int) (change - 0.5));
```

```
-2
-3
-2
```

Line	Expression	Before the cast	Truncated toward zero
2	<code>(int) (-2.7 + 0.5)</code>	-2.2	-2 (wrong)
3	<code>(int) (-2.7 - 0.5)</code>	-3.2	-3 (correct)
5	<code>(int) (-2.2 - 0.5)</code>	-2.7	-2 (correct)

Adding 0.5 to a negative number moves it toward zero, and then truncating moves it toward zero again, so `-2.7` ends up at `-2`. Subtracting 0.5 moves it away from zero first, so truncation lands on `-3`. Line 5 checks that a value closer to `-2`, such as `-2.2`, still rounds to `-2`.

WATCH OUT

Leaving out the parentheses. A student writes the rounding formula without the parentheses around the sum:

RoundNoParens.java

```
1 public class RoundNoParens
2 {
3     public static void main(String[] args)
4     {
5         double distance = 3.7;
6         int rounded = (int) distance + 0.5;
7         System.out.println(rounded);
8     }
9 }
```

```
RoundNoParens.java:6: error: incompatible types: possible lossy conversion from
double to int
```

The cast applies only to `distance`, so `(int) distance` is 3, and then `3 + 0.5` is the double 3.5. Java will not store a double in an `int` variable, so the code does not compile. Students drop the parentheses because the formula "looks" like cast-then-add in their head. Write `(int) (distance + 0.5)` so the addition happens inside the cast.

PAUSE AND TRY**Question 7** Tracing

What does each line print?

```
1 System.out.println((int) (6.49 + 0.5));
2 System.out.println((int) (6.5 + 0.5));
3 System.out.println((int) (-1.5 - 0.5));
```

Question 8 Debugging

A weather program should round the low temperature, -4.6 degrees, to the nearest whole degree, -5. It prints -4 instead. Find the error and correct line 2.

```
1 double lowTemp = -4.6;
2 int rounded = (int) (lowTemp + 0.5);
3 System.out.println(rounded);
```

1.5.5 The Range of int Values

Could an `int` variable hold the number of people on Earth, about 8.1 billion? It seems like a number is a number. But every variable is a storage location in memory (Topic 1.2), and a storage location has a fixed size. The computer sets aside a specific amount of memory for each value based on its data type, and an `int` always gets the same amount: **4 bytes**.

Memory is measured in bits and bytes. A **bit** is a single 0 or 1, and a **byte** is 8 bits. So an `int` has $4 \times 8 = 32$ bits. Each bit has two possible settings, so 32 bits can be set in $2 \times 2 \times \dots \times 2$ (32 times) = 4,294,967,296 different patterns. That is how many different `int` values exist, and no more. Think of a car's trip counter with a fixed number of digit wheels: it can show only so many different readings.

Java splits those patterns between negative and nonnegative numbers: 2,147,483,648 of them for the negatives, and 2,147,483,648 for zero and the positives. Because zero takes one of the nonnegative slots, the largest positive `int` is one less in size than the most negative one. Those two limits are so important that Java gives them names:

- `Integer.MAX_VALUE` is the largest possible `int`, 2,147,483,647.
- `Integer.MIN_VALUE` is the smallest possible `int`, -2,147,483,648.

Each is a **constant**: a named value that never changes while the program runs. `Integer` is a class that comes with Java, and the dot means "the `MAX_VALUE` that belongs to `Integer`"; you will learn how classes and the values they contain work in Topics 1.7 and 1.10, so for now read each name as a whole. Constants are traditionally written in capital letters with underscores.

Every `int` value must be between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, **inclusive**, meaning both limits are themselves allowed.

KEY IDEA

An `int` is stored in 4 bytes of memory, so only a limited range of whole numbers fits: from `Integer.MIN_VALUE` (-2,147,483,648) to `Integer.MAX_VALUE` (2,147,483,647), including both ends.

WORKED EXAMPLE Printing the limits

The program prints a label with `print`, then the value with `println`, so each pair shares a line.

```
1 System.out.print("Largest int: ");
2 System.out.println(Integer.MAX_VALUE);
3 System.out.print("Smallest int: ");
4 System.out.println(Integer.MIN_VALUE);
```

```
Largest int: 2147483647
Smallest int: -2147483648
```

Java prints the values without commas. The largest is a little over 2.1 billion, and the smallest is -2,147,483,648, which ends in 8 rather than 7 because of the extra slot on the negative side.

Now try a value that is outside the range from the start. Java checks a number written directly in the code (a literal) when it compiles:

TooLargeLiteral.java

```
1 public class TooLargeLiteral
2 {
3     public static void main(String[] args)
4     {
5         int worldPopulation = 8100000000;
6         System.out.println(worldPopulation);
7     }
8 }
```

```
TooLargeLiteral.java:5: error: integer number too large
```

8,100,000,000 is bigger than 2,147,483,647, so it cannot be an `int` at all. The compiler catches this one because the number is sitting right there in the code. As the next subsection shows, the compiler cannot catch a value that only becomes too large while the program runs.

WATCH OUT

Assuming the two limits are mirror images. Many students expect the smallest `int` to be `-2,147,483,647`, the negative of the largest. It is `-2,147,483,648`: the negative side has one more value because zero uses one of the nonnegative patterns. When a question asks for an exact limit, use the constant names, or remember that `Integer.MIN_VALUE` ends in 8 and `Integer.MAX_VALUE` ends in 7.

PAUSE AND TRY**Question 9** Multiple choice

Which value can be stored in an `int` variable?

- A. 2147483648
- B. -2147483648
- C. 3000000000
- D. 2147483647.5

1.5.6 Integer Overflow

The compiler caught 8,100,000,000 because it was written in the code. But most numbers in a program are calculated while it runs. What happens when a calculation's true answer is bigger than `Integer.MAX_VALUE`?

Return to the trip counter. If it shows 999999 and you drive one more mile, it does not stop or complain; it rolls over to 000000. An `int` behaves similarly. If an expression would produce an `int` value outside the allowed range, an **integer overflow** occurs. Java does not stop the program and does not report an error. It produces an `int` that is inside the range, but it is usually not the value you expected. Past the top of the range, the result wraps around to the bottom.

The analogy has a limit: the trip counter rolls over to zero, but an `int` that goes one past the top wraps to the **most negative** value, `Integer.MIN_VALUE`. That is because the negative values sit "after" the largest positive one in the way the 32 bits are arranged.

KEY IDEA

When an `int` expression would give a value outside the `int` range, integer overflow happens: Java quietly produces an `int` in the range, usually not the one you expected. No error or exception is reported.

WORKED EXAMPLE One past the largest int

```
1 int biggest = Integer.MAX_VALUE;
2 int oneMore = biggest + 1;
3 System.out.println(biggest);
4 System.out.println(oneMore);
```

```
2147483647
-2147483648
```

- Line 1 stores `Integer.MAX_VALUE`, 2,147,483,647, in `biggest`.
- Line 2 adds 1. Both operands are `int`, so the result must be an `int`. The true sum, 2,147,483,648, does not fit, so the value wraps around to the bottom of the range, -2,147,483,648, which is `Integer.MIN_VALUE`.
- The program runs to the end. Nothing warns you that `oneMore` is wrong.

Going below the bottom wraps the other way, to the top:

```
1 int smallest = Integer.MIN_VALUE;
2 int oneLess = smallest - 1;
3 System.out.println(oneLess);
```

```
2147483647
```

Overflow does not need anything as obvious as `Integer.MAX_VALUE`. Here a program computes how many milliseconds (thousandths of a second) are in a 30-day month, and compares it with the number of seconds.

```
1 int millisInMonth = 30 * 24 * 60 * 60 * 1000;
2 System.out.println(millisInMonth);
3 int secondsInMonth = 30 * 24 * 60 * 60;
4 System.out.println(secondsInMonth);
```

```
-1702967296
2592000
```

The multiplications happen left to right:

Step	True result	Fits in an int?
------	-------------	-----------------

30 * 24	720	Yes
720 * 60	43,200	Yes
43200 * 60	2,592,000	Yes
2592000 * 1000	2,592,000,000	No: larger than 2,147,483,647

The last step overflows, and the program prints a negative number of milliseconds, which is impossible. This is a logic error (Topic 1.1): the code compiles and runs, but the result is wrong. The second calculation stops before the `* 1000`, and 2,592,000 fits comfortably. Choosing a unit that keeps values in range (seconds instead of milliseconds) is one way to avoid overflow. You do not need to predict the exact wrong value an overflow produces in a case like this; what matters is recognizing when a calculation goes past the limits.

WATCH OUT

Thinking an answer is safe because the final result fits. A program averages two large values. The average, 2,000,000,000, is less than `Integer.MAX_VALUE`, so the student expects it to work:

```
1 int first = 2000000000;
2 int second = 2000000000;
3 int average = (first + second) / 2;
4 System.out.println(average);
```

-147483648

The parentheses make Java add first, and the sum, 4,000,000,000, is out of range. It overflows to a negative number, and dividing that by 2 gives a negative "average." Every step of an `int` calculation must stay in range, not just the final answer. Here, dividing each value by 2 first keeps every step small:

```
1 int first = 2000000000;
2 int second = 2000000000;
3 int average = first / 2 + second / 2;
4 System.out.println(average);
```

2000000000

(This particular fix is exact because both values are even; with odd values, each `int` division would drop a half. The point is to notice which step overflows.)

PAUSE AND TRY**Question 10** Tracing

What does this code print?

```
1 int low = Integer.MIN_VALUE;
2 int below = low - 1;
3 int stepBack = below + 1;
4 System.out.println(below);
5 System.out.println(stepBack);
```

Question 11 Multiple choice

Each statement declares an `int`. In which one does an integer overflow happen?

- A. `int seconds = 60 * 60 * 24 * 365;`
- B. `int big = 2000000000 + 1000000000;`
- C. `int total = 1000000 * 3000;`
- D. `int small = 0 - 2000000000;`

1.5.7 Round-Off Error with double Values

Add 0.1 and 0.2 on a calculator and you get 0.3. Ask Java:

```
1 System.out.println(0.1 + 0.2);
2 System.out.println(1.0 - 0.9);
```

```
0.30000000000000004
0.09999999999999998
```

Both answers are off by a tiny amount. This is not a bug in Java; it comes from the same fact as the last two subsections. A `double` is stored in a fixed amount of memory (8 bytes), so it can hold only a limited number of digits. Just as an `int` has a largest value, a `double` has a limit on how **precise** it can be: how many digits it can keep.

You have seen this problem with fractions. One third written as a decimal is 0.3333... forever, so on paper you must stop somewhere and round. A `double` stores its digits in binary (0s and 1s), and in binary, simple decimals like 0.1 and 0.2 repeat forever, the way 1/3 does in decimal. So Java stores the closest value it can, which is very slightly off. When a calculation's exact answer has more digits than a `double` can hold, the result is rounded to the nearest value a `double` can store. The small difference is called a **round-off error**.

`int` values do not have this problem. Every whole number in the `int` range is stored exactly, and adding, subtracting, or multiplying them gives exact results (as long as nothing overflows). So when exact results matter, store the values as `int`.

KEY IDEA

A `double` can keep only a limited number of digits, so some results are rounded to the nearest value it can store, causing a small round-off error. Use `int` values when results must be exact.

WORKED EXAMPLE Three items at \$1.10, as dollars and as cents

First with `double` dollars:

```
1 double price = 1.10;
2 double total = price * 3;
3 System.out.println(total);
```

```
3.3000000000000003
```

1.10 is stored as a value very close to 1.1, and multiplying by 3 makes the tiny difference show up in the printed digits. Now store the price as a whole number of cents, an `int`:

```
1 int priceInCents = 110;
2 int totalCents = priceInCents * 3;
3 System.out.println(totalCents);
4 System.out.print("Dollars: ");
5 System.out.println(totalCents / 100);
6 System.out.print("Cents: ");
7 System.out.println(totalCents % 100);
```

```
330
Dollars: 3
Cents: 30
```

- Line 1 stores the price as 110 cents, which is exact.
- Line 2 multiplies two `int` values: 330 cents, exact.
- Lines 5 and 7 split the total for display with Topic 1.3's operators: $330 / 100$ is 3 whole dollars, and $330 \% 100$ is the 30 cents left over.

This is why programs that handle money often count in cents.

WATCH OUT

Truncating a value that has round-off error. A student converts a price of \$0.29 to cents by multiplying by 100 and casting:

```
1 double price = 0.29;
2 System.out.println(price * 100);
3 int truncated = (int) (price * 100);
4 System.out.println(truncated);
5 int rounded = (int) (price * 100 + 0.5);
6 System.out.println(rounded);
```

```
28.999999999999996
28
29
```

$0.29 * 100$ comes out just below 29 because of round-off. Truncating then chops the result down to 28, a whole cent lost. Students expect 29 because the math on paper is exact. The

fix combines two ideas from this chapter: round instead of truncating, with `(int) (price * 100 + 0.5)`. Adding 0.5 gives about 29.4999..., which truncates to 29.

PAUSE AND TRY

Question 12 Short answer

Explain why `System.out.println(0.1 + 0.2);` prints `0.30000000000000004` instead of `0.3`.

Question 13 Multiple choice

A store's program must add up thousands of prices and get the total exactly right, to the cent. Which choice avoids round-off error?

- A. Store each price in dollars as a `double`, such as 4.99.
- B. Store each price in cents as an `int`, such as 499.
- C. Store each price in dollars as a `double`, and cast the total with `(int)`.
- D. Store each price in whole dollars as an `int`, such as 4.

Putting it together

A science class records the low temperature, as a whole number of degrees, on three winter days: -3, -3, and -2. The program below reports the average four ways. Before reading the output, predict each line.

TemperatureAverage.java

```
1 public class TemperatureAverage
2 {
3     public static void main(String[] args)
4     {
5         int monday = -3;
6         int tuesday = -3;
7         int wednesday = -2;
8         int sum = monday + tuesday + wednesday;
9         System.out.print("int division: ");
10        System.out.println(sum / 3);
11        double average = (double) sum / 3;
12        System.out.print("Exact average: ");
13        System.out.println(average);
14        System.out.print("Truncated: ");
15        System.out.println((int) average);
16        System.out.print("Rounded: ");
17        System.out.println((int) (average + 0.5));
18    }
19 }
```

```
int division: -2
Exact average: -2.6666666666666665
Truncated: -2
Rounded: -3
```

Line	Code	What happens	Value
8	<code>monday + tuesday + wednesday</code>	<code>-3 + -3 + -2</code> , all <code>int</code>	<code>-8</code>
10	<code>sum / 3</code>	Two <code>int</code> values: keep only the whole part of <code>-2.666...</code> , toward zero	<code>-2</code>
11	<code>(double) sum / 3</code>	Cast first: <code>-8.0</code> ; then <code>3</code> is widened; <code>double</code> division	<code>-2.6666666666666665</code>
15	<code>(int) average</code>	Truncate toward zero	<code>-2</code>
17	<code>(int) (average - 0.5)</code>	<code>-2.666... - 0.5 = -3.166...</code> , then truncate toward zero	<code>-3</code>

What each idea contributed:

- **Casting an operand (1.5.3).** Without `(double)` on line 11, the division would be `int` division and the average would be `-2.0`. The cast goes on `sum`, before the division, not around `sum / 3`.
- **Truncation toward zero (1.5.2).** `int` division and the `(int)` cast both give `-2`, because both drop the decimal part. On a number line, `-2.666...` is closer to `-3`, so neither one rounds.
- **Rounding (1.5.4).** The average is negative, so the program uses `(int) (average - 0.5)`. Using `+ 0.5` would give `(int) (-2.1666...)`, which is `-2`, the wrong answer.
- **Round-off (1.5.7).** The exact average is $-2\frac{2}{3}$, whose decimal digits go on forever. A `double` cannot hold them all, so the last printed digit is `5`, the stored value rounded to the nearest one a `double` can hold.
- **Range (1.5.5 and 1.5.6).** Temperatures are small numbers, so no step comes anywhere near `Integer.MAX_VALUE` or `Integer.MIN_VALUE`; overflow is not a concern here. It would be if the program summed millions of large values.

Vocabulary

Term	Definition	Example
widening	Java's automatic conversion of an <code>int</code> value to a <code>double</code> , with no change in amount	<code>double price = 5</code> ; stores <code>5.0</code>
cast	An instruction that converts a value to another type	<code>(int) 26.2</code> is <code>26</code>
casting operator	A type name in parentheses, written in front of the value to convert	<code>(int)</code> , <code>(double)</code>
truncate	Drop every digit after the decimal point, which moves the value toward zero	<code>(int) -7.99</code> is <code>-7</code>
bit / byte	A single 0 or 1 / a group of 8 bits	An <code>int</code> uses 4 bytes, 32 bits
constant	A named value that never changes while a program runs	<code>Integer.MAX_VALUE</code>
<code>Integer.MAX_VALUE</code>	The largest possible <code>int</code>	<code>2147483647</code>
<code>Integer.MIN_VALUE</code>	The smallest possible <code>int</code>	<code>-2147483648</code>
inclusive	Including the endpoints of a range	Both limits are valid <code>int</code> values

integer overflow	What happens when an <code>int</code> expression would go outside the range: the result wraps to an unexpected <code>int</code>	<code>Integer.MAX_VALUE + 1</code> is <code>Integer.MIN_VALUE</code>
precise / precision	How many digits a value can keep	A <code>double</code> keeps about 16 digits
round-off error	A small error caused by rounding a <code>double</code> result to a value that fits in its memory	<code>0.1 + 0.2</code> prints <code>0.30000000000000004</code>

Chapter summary

- Java automatically widens an `int` to a `double` when a `double` is needed, in an assignment or in an operation with a `double` operand. It never turns a `double` into an `int` on its own, because the decimal part could be lost.
- A cast converts a value on purpose. `(int)` truncates a `double` toward zero, and `(double)` turns an `int` into a `double`. A cast applies only to the value right after it, so cast an operand before dividing to keep the decimal part.
- Round a `double` to the nearest `int` with `(int) (x + 0.5)` when `x` is zero or positive, and `(int) (x - 0.5)` when `x` is negative.
- An `int` is stored in 4 bytes, so its values run from `Integer.MIN_VALUE` (`-2,147,483,648`) to `Integer.MAX_VALUE` (`2,147,483,647`), inclusive. An `int` expression that would go outside that range overflows and quietly gives an unexpected value.
- A `double` has limited precision, so some results carry a small round-off error. Use `int` values, such as cents instead of dollars, when results must be exact.

Review questions

Question 14 Multiple choice

What is stored in `result`?

```
1 double result = 9 / 2 + 1.5;
2 System.out.println(result);
```

- A. 6.0
- B. 5.5
- C. 5
- D. Nothing; the code does not compile

Question 15 Tracing

A runner does 17 laps in 5 days. Trace the code and give the exact output.

```
1 int laps = 17;
2 double perDay = laps / 5;
3 double exact = (double) laps / 5;
4 int whole = (int) exact;
5 int nearest = (int) (exact + 0.5);
6 System.out.println(perDay);
7 System.out.println(exact);
8 System.out.println(whole);
9 System.out.println(nearest);
```

Question 16 Debugging

A quiz app should show the percentage of questions answered correctly, rounded to the nearest whole percent. With 7 correct out of 8, it should show 88 ($7 \div 8$ is 87.5%). The code prints 0. Explain the error and give a corrected line 3.

```
1 int correct = 7;
2 int total = 8;
3 int percent = correct / total * 100;
4 System.out.println(percent);
```

Question 17 Multiple choice

A game stores a player's points in an `int`. The player has 2,147,483,000 points and earns 1,000 more:

```
1 int points = 2147483000;
2 points = points + 1000;
3 System.out.println(points);
```

What happens?

- A. The program does not compile, because the sum is too large for an `int`.
- B. The program stops with an exception when it reaches line 2.
- C. The program runs, and `points` ends up holding a large negative number.
- D. The program runs, and `points` stays at `Integer.MAX_VALUE`.

Question 18 Short answer

A bank program stores account balances as a whole number of cents in an `int`, so that 1999 means \$19.99. Give one advantage of this choice over storing dollars in a `double`, and one limit it creates. Use numbers in your answer.

Question 19 Multiple choice

What is printed?

```
1 double ratio = (double) (int) 7.8 / 2;
2 System.out.println(ratio);
```

- A. 3.9
- B. 3.5
- C. 3.0
- D. 4.0

Reflection

- ☐ I can explain when Java widens an int to a double and why it never narrows a double to an int on its own. Reread: 1.5.1 Java Widens an int to a double
- ☐ I can predict the result of (int) on positive and negative double values. Reread: 1.5.2 Casting a double to an int
- ☐ I can place a (double) cast so that a division keeps its decimal part. Reread: 1.5.3 Casting an int to a double
- ☐ I can round a positive or negative double to the nearest int. Reread: 1.5.4 Rounding to the Nearest Integer
- ☐ I can state the smallest and largest int values and explain why the range is limited. Reread: 1.5.5 The Range of int Values
- ☐ I can recognize when an int calculation overflows and describe what happens. Reread: 1.5.6 Integer Overflow
- ☐ I can explain round-off error and when to use int values instead of double. Reread: 1.5.7 Round-Off Error with double Values

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

C. `12.0` is a `double` literal, even though its value is a whole number. Java never converts a `double` to an `int` on its own, so the compiler reports possible lossy conversion from `double` to `int`. The compiler goes by the type, not by whether the decimal part happens to be zero.

- **A** compiles: the `int` `12` is widened to `12.0` and stored.
- **B** compiles: `4` is widened to `4.0`, and the `double` result `10.0` is stored in a `double`.
- **D** compiles: `7 / 2` divides two `int` values, giving the `int` `3`, which fits in an `int` variable.

Choices A, B, and D together:

```
1 double average = 12;
2 double total = 4 * 2.5;
3 int score = 7 / 2;
4 System.out.println(average);
5 System.out.println(total);
6 System.out.println(score);
```

```
12.0
10.0
3
```

Choice C on its own:

`WholeDoubleToInt.java`

```
1 public class WholeDoubleToInt
2 {
3     public static void main(String[] args)
4     {
5         int count = 12.0;
6         System.out.println(count);
7     }
8 }
```

Reread 1.5.1 *Java Widens an int to a double* if you chose differently.

Question 2

```
4.0
4.5
```

- Line 1: `9 / 4` comes first. Both operands are `int`, so the result is the whole part, `2`. Then `2 * 2.0` widens `2` to `2.0` and gives `4.0`.
- Line 2: `9 * 2.0` comes first. `9` is widened to `9.0`, giving `18.0`. Then `18.0 / 4` widens `4` to `4.0` and gives `4.5`.

Both lines contain a `double`, but in line 1 the `int` division happened before the `double` was involved. If you wrote `4.5` for line 1, you widened the whole line at once instead of one operation at a time. Reread *1.5.1 Java Widens an int to a double*.

Question 3

```
12
-3
0
```

Each cast drops the digits after the decimal point and keeps the whole-number part. `12.5` becomes `12` (no rounding up, even at exactly `.5`). `-3.2` becomes `-3`, which is toward zero; it is not `-4`. `0.999` becomes `0`, however close it is to 1.

Reread *1.5.2 Casting a double to an int* if any of your answers differed.

Question 4

Why: `minutes / 60.0` has a `double` operand, so `minutes` is widened and the result is the `double` `2.25`. Storing a `double` in an `int` variable could lose the `.25`, so Java will not do it on its own. The compiler reports incompatible types: possible lossy conversion from `double` to `int` at line 6.

Corrected line: cast the whole quotient, using parentheses so the cast applies to the result of the division:

```
1 int minutes = 135;
2 int wholeHours = (int) (minutes / 60.0);
3 System.out.println(wholeHours);
```

```
2
```

`135 / 60.0` is `2.25`, and `(int)` truncates it to `2`. (Writing `minutes / 60` would also store `2`, because `int` division already keeps only the whole part.)

Reread *1.5.2 Casting a double to an int* if your fix differed.

Question 5

B. The cast applies only to `a`, turning it into `5.0` before the division. Then `5.0 / b` widens `2` to `2.0`, and the result is `2.5`.

```
1 int a = 5;
2 int b = 2;
3 System.out.println((double) (a / b));
4 System.out.println((double) a / b);
5 System.out.println((int) (a / 2.0));
6 System.out.println(a / b * 1.0);
```

```
2.0
2.5
2
2.0
```

- **A** is wrong: the parentheses make `a / b` happen first, as `int` division, giving 2. The cast then turns 2 into 2.0.
- **C** is wrong: `a / 2.0` is 2.5, but the `(int)` cast truncates it to 2.
- **D** is wrong: `a / b` is done first and gives 2; multiplying by 1.0 only widens it to 2.0.

Reread 1.5.3 *Casting an int to a double* if you chose differently.

Question 6

Model answer: A cast applies only to the value right after it. In `(double) (total / count)`, the value after the cast is the parenthesized division, so the division happens first, as `int` division: `256 / 3` is 85, and the cast turns that into 85.0. In `(double) total / count`, the cast applies to `total` alone, so 256 becomes 256.0 first, the division has a `double` operand, and the result is 85.33333333333333.

A complete answer must:

- say that the cast applies only to the value immediately after it,
- say that in the first expression the `int` division happens before the cast, so the decimal part is already lost, and
- give both results, 85.0 and 85.33333333333333.

Reread 1.5.3 *Casting an int to a double* if your explanation differed.

Question 7

6
7
-2

Value	After adjusting	Truncated
6.49	6.99	6
6.5	7.0	7
-1.5	-2.0	-2

6.49 is just under halfway, so it rounds down to 6. 6.5 is exactly halfway and rounds up to 7. For the negative value, subtracting 0.5 gives -2.0, which truncates to -2 (a halfway negative value rounds away from zero).

Reread 1.5.4 *Rounding to the Nearest Integer* if any line differed.

Question 8

What is wrong: the formula `(int) (x + 0.5)` works only for values that are zero or positive. `-4.6 + 0.5` is -4.1, and truncating moves it toward zero, to -4. Java compiles and runs the code, so this is a logic error.

Corrected code: subtract 0.5 for a negative value.

```
1 double lowTemp = -4.6;
2 int rounded = (int) (lowTemp - 0.5);
3 System.out.println(rounded);
```

-5

$-4.6 - 0.5$ is -5.1 , which truncates toward zero to -5 .

Reread *1.5.4 Rounding to the Nearest Integer* if you did not spot the sign problem.

Question 9

B. $-2,147,483,648$ is `Integer.MIN_VALUE`. The range is inclusive, so the smallest value itself is allowed.

```
1 System.out.print("Largest int: ");
2 System.out.println(Integer.MAX_VALUE);
3 System.out.print("Smallest int: ");
4 System.out.println(Integer.MIN_VALUE);
```

- **A** is wrong: it is one more than `Integer.MAX_VALUE` ($2,147,483,647$), so it is out of range. It is the right size only on the negative side.
- **C** is wrong: three billion is larger than `Integer.MAX_VALUE`, so writing it in an `int` declaration gives the compile error `integer number too large`.
- **D** is wrong: it has a decimal part, so it is not an `int` value at all, whatever its size.

Reread *1.5.5 The Range of int Values* if you chose differently.

Question 10

```
2147483647
-2147483648
```

- Line 1: `low` is `Integer.MIN_VALUE`, $-2,147,483,648$.
- Line 2: one less than the smallest `int` is out of range, so it wraps to the top: $2,147,483,647$, which is `Integer.MAX_VALUE`.
- Line 3: one more than the largest `int` is out of range, so it wraps to the bottom again: $-2,147,483,648$.

Overflow works in both directions, and wrapping one way then the other returns to the starting value. Reread *1.5.6 Integer Overflow* if your trace differed.

Question 11

C. $1,000,000 \times 3,000$ is $3,000,000,000$, which is larger than `Integer.MAX_VALUE` ($2,147,483,647$). Both operands are `int`, so the product overflows. The code compiles, and `total` ends up holding a wrong value. Printing all four variables shows it:

```
1 int seconds = 60 * 60 * 24 * 365;
2 int big = 2000000000 + 100000000;
3 int total = 1000000 * 3000;
4 int small = 0 - 2000000000;
5 System.out.println(seconds);
6 System.out.println(big);
7 System.out.println(total);
8 System.out.println(small);
```

```
31536000
2100000000
-1294967296
-2000000000
```

- **A** is wrong: the number of seconds in a year is 31,536,000, well within range.
- **B** is wrong: 2,100,000,000 is just under 2,147,483,647, so it fits.
- **D** is wrong: -2,000,000,000 is greater than `Integer.MIN_VALUE` (-2,147,483,648), so it fits.

Reread *1.5.6 Integer Overflow* if you chose differently.

Question 12

Model answer: A `double` is stored in a fixed amount of memory, so it can hold only a limited number of digits. In binary, 0.1 and 0.2 cannot be written exactly (they repeat forever, as 1/3 does in decimal), so Java stores the closest values it can. Adding those slightly-off values gives a result that is rounded to the nearest value a `double` can hold, which prints as `0.30000000000000004`. The tiny difference is a round-off error.

A complete answer must:

- say that a `double` has limited memory and therefore limited precision, and
- say that the stored values or result are rounded to the nearest value a `double` can hold.

Reread *1.5.7 Round-Off Error with double Values* if your explanation differed.

Question 13

B. Whole numbers in the `int` range are stored exactly, and adding them gives exact results. Counting in cents keeps every price a whole number without losing any value.

- **A** is wrong: decimal prices like 4.99 cannot always be stored exactly in a `double`, so round-off errors can build up over thousands of additions.
- **C** is wrong: the `double` total may already contain round-off error, and `(int)` then truncates away all the cents.
- **D** is wrong: an `int` is exact, but whole dollars throw away the 99 cents before any arithmetic happens.

Reread *1.5.7 Round-Off Error with double Values* if you chose differently.

Question 14

B. Division comes before addition. `9 / 2` divides two `int` values, so it gives `4`. Then `4 + 1.5` widens `4` to `4.0` and gives `5.5`, a `double`, which is stored in the `double` variable.

```
5.5
```

- **A** is wrong: it treats `9 / 2` as `4.5`, as if the `1.5` later in the line made the division a `double` division. Widening happens one operation at a time.
- **C** is wrong: once `1.5` is involved, the result is a `double`, and a `double` variable holds a `double`.
- **D** is wrong: the right side is a `double`, which is exactly what a `double` variable can hold.

Reread *1.5.1 Java Widens an int to a double* if you chose differently.

Question 15

3.0
3.4
3
3

Line	Variable	How it is computed	Value
2	perDay	17 / 5 is int division, 3; widened when stored	3.0
3	exact	(double) laps is 17.0; 17.0 / 5 keeps the decimal part	3.4
4	whole	(int) 3.4 truncates	3
5	nearest	3.4 + 0.5 is 3.9, which truncates to 3	3

whole and nearest are equal here only because 3.4 is closer to 3 than to 4. With 3.6 they would differ (3 and 4). If you wrote 3.4 for perDay, review where the cast goes in [1.5.3 Casting an int to a double](#); if nearest gave you trouble, reread [1.5.4 Rounding to the Nearest Integer](#).

Question 16

What is wrong: correct / total divides two int values, so 7 / 8 keeps only the whole part, 0. Then 0 * 100 is 0. The code compiles and runs, so this is a logic error.

Corrected line:

```
1 int correct = 7;
2 int total = 8;
3 int percent = (int) ((double) correct / total * 100 + 0.5);
4 System.out.println(percent);
```

88

- (double) correct turns 7 into 7.0 before the division, so 7.0 / 8 is 0.875.
- 0.875 * 100 is 87.5.
- The percentage is never negative, so add 0.5 to round: 88.0, which (int) truncates to 88.

A version that casts without rounding compiles but gives 87, because truncation drops the .5:

```
1 int correct = 7;
2 int total = 8;
3 int percent = (int) ((double) correct / total * 100);
4 System.out.println(percent);
```

87

Reread [1.5.3 Casting an int to a double](#) and [1.5.4 Rounding to the Nearest Integer](#) if your fix differed.

Question 17

C. The true sum, 2,147,484,000, is larger than Integer.MAX_VALUE (2,147,483,647), so the addition overflows and wraps around past the bottom of the range. The program prints -2147483296 and never reports a problem.

- **A** is wrong: the compiler checks only literals written in the code, such as `8100000000`. Each literal here fits; the sum is only calculated while the program runs.
- **B** is wrong: integer overflow does not cause an exception. The program keeps running with a wrong value, which is what makes it a logic error.
- **D** is wrong: an `int` does not stop at its limit; it wraps around.

Reread *1.5.6 Integer Overflow* if you chose differently.

Question 18

Model answer: The advantage is exactness. Every `int` is stored exactly, so adding cents never produces a round-off error, while a `double` such as 1.10×3 can come out as `3.3000000000000003`. The limit is range. The largest `int` is `2,147,483,647`, so the largest balance this program can hold is `2,147,483,647` cents, about \$21.47 million. A deposit that pushed a balance past that would overflow and give a wrong (likely negative) balance with no error message.

A complete answer must:

- name exactness (no round-off error) as the advantage of `int` cents,
- name the `int` range as the limit, using `Integer.MAX_VALUE` or about \$21 million, and
- say that going past it causes overflow rather than an error.

Reread *1.5.7 Round-Off Error with double Values* and *1.5.5 The Range of int Values* if your answer differed.

Question 19

B. Two casts in a row are applied right to left, each to the value right after it. First `(int) 7.8` truncates to `7`. Then `(double)` turns `7` into `7.0`. Both casts happen before the division, so the expression becomes `7.0 / 2`, which widens `2` and gives `3.5`.

- **A** is wrong: it skips the `(int)` cast and divides `7.8` by `2`.
- **C** is wrong: it treats the division as `int` division, `7 / 2`, and then converts. The `(double)` cast happens before the division, so the division has a `double` operand.
- **D** is wrong: it rounds `7.8` up to `8` instead of truncating it to `7`.

Reread *1.5.2 Casting a double to an int* and *1.5.3 Casting an int to a double* if you chose differently.