

1.5 Casting and Range of Variables

Lesson Plan

Pacing: 2 class periods of 45 minutes

Learning objectives: 1.5.A, 1.5.B, 1.5.C

Essential knowledge: 1.5.A.1, 1.5.A.2, 1.5.A.3, 1.5.A.4, 1.5.B.1, 1.5.B.2, 1.5.B.3, 1.5.C.1

Materials

- Projector
- Whiteboard and markers
- Scrap paper or mini whiteboards for predictions
- IDE on every computer
- Cast, Chop, and Wrap slide deck
- Printed Cast, Chop, and Wrap puzzle worksheet and exit ticket

Agenda

Minutes	Activity
5	Day 1 warm-up — numbers with limits (slide 2), then the average of 256 points over 3 quizzes
7	Day 1 direct instruction — widening, one operation at a time (slides 3–5)
8	Day 1 direct instruction — the (int) cast truncates toward zero (slide 6)
9	Day 1 direct instruction — the (double) cast and where it goes (slides 7–8)
7	Day 1 direct instruction — rounding to the nearest int (slides 9–13)
6	Day 1 guided practice — fix the quiz percentage in the IDE
3	Day 1 closure — chop, round, or wrap? (slide 22), then hand out the puzzle worksheet
5	Day 2 warm-up — can an int hold 8.1 billion?
8	Day 2 direct instruction — 4 bytes, Integer.MIN_VALUE and Integer.MAX_VALUE (slides 14–15)
9	Day 2 direct instruction — integer overflow (slides 16–17)
7	Day 2 direct instruction — round-off error and exact int values (slides 18–20)
10	Day 2 guided practice — overflow and round-off experiments in the IDE
6	Day 2 closure — words to know (slide 21), then the exit ticket

Overview

Day 1 is conversion between `int` and `double` (1.5.A). Students see Java widen an `int` to a `double` on its own (1.5.A.3), use the casting operators `(int)` and `(double)` on purpose (1.5.A.1), learn that `(int)` truncates (1.5.A.2), and round with `(int) (x + 0.5)` and `(int) (x - 0.5)` (1.5.A.4). Day 2 is the limits of each type: the `int` range and integer overflow (1.5.B), and round-off error in `double` values (1.5.C).

Students have declared `int`, `double`, and `boolean` variables (Topic 1.2), evaluated arithmetic with integer division and `%` (Topic 1.3), and stored values with assignment statements (Topic 1.4). In Topic 1.4 they met possible lossy conversion from `double` to `int` and were told to match the types; today explains why that rule is one-way. Stay inside the topic boundary:

- A label and a value still take a `print` followed by a `println`; joining text and numbers in one statement is Topic 1.15.
- Do not use `+=` or other compound assignment operators (Topic 1.6). Write `points = points + 1000;`.
- Do not use `Math` methods such as `Math.round` (Topics 1.10 and 1.11). Rounding in this topic is the cast formula only.
- `Integer.MAX_VALUE` and `Integer.MIN_VALUE` are read as whole names. Say once that the dot and the `Integer` class are explained in Topics 1.7 and 1.10.
- The CED excludes other decimal data types that avoid rounding errors. The fix taught here is to use `int` values, such as cents instead of dollars.

TEACHER NOTE

Files whose names end in `.snippet.java` hold only the statements that go inside `main`; paste them into the `main` of a scratch class to run them. All files are in the topic's `examples/` folder.

- **Day 1 demonstrations:** `AverageIntDivision`, `WidenAssign`, `WidenArithmetic`, `WidenPerOperation`, `NoAutoNarrow.java`, `WholeDoubleToInt.java`, `CastToInt`, `TruncateTowardZero`, `CastIsNotRounding`, `AverageCast`, `CastOtherOperand`, `CastTooLate`, `RoundPositive`, `RoundNegative`, `RoundNoParens.java`, and, for the guided practice, `PercentWrong`, `PercentTruncated`, and `PercentFixed` (the model fix; do not show it until students have tried).
- **Day 2 demonstrations:** `TooLargeLiteral.java`, `IntRange`, `OverflowMax`, `OverflowMin`, `MillisInMonth`, `AverageOverflow`, `AverageOverflowFixed`, `GamePoints`, `RoundOff`, `PriceDouble`, `PriceCents`, and `DollarsToCents`.
- **Printed only (exit ticket):** `ExitCasts`. The puzzle worksheet and its key are separate Word files; their code is printed on the page.

The Cast, Chop, and Wrap deck covers both days: slides 1–13 on Day 1, plus slide 22 as the Day 1 closure, and slides 14–21 on Day 2. Its speaker notes hold the answers to every Pause and Try slide. Before Day 1, open a scratch class in the IDE next to the deck so each snippet is one paste away. Compiler messages in this topic are quoted from a recent JDK; the wording can vary slightly between Java versions.

Day 1 warm-up: the average of three quizzes

Open with slide 2 (Numbers with limits): give pairs two minutes on the odometer at 999999 and the 89.6 grade. Collect "it rolls over" and "89 or 90", and say both are today's topic. Then show `AverageIntDivision`: three quiz scores total 256 points, and the program displays `total / count`. Ask students to write two things: what Java displays, and what the true average is. Most will get 85 from Topic 1.3's integer division, and some will work out 85.333... by hand. Leave the gap open:

"Today you will learn two ways to make Java keep that .333, and why one of them looks right but is not."

Day 1 direct instruction: widening

Read the goals on slide 3, then move to Part 1 (slide 4).

1. **Widening** (1.5.A.3, slide 5). Every whole number is also a decimal number: 5 is the same amount as 5.0. Turning an `int` into a `double` loses nothing, so Java does it on its own. This automatic conversion is called **widening**. It happens in two places students already know: storing an `int` value in a `double` variable (WidenAssign: `double price = wholePrice;` stores 5.0), and an arithmetic operation with one `int` and one `double` operand (WidenArithmetic: `laps * lapLength` widens 7 to 7.0). This is the "how" behind Topic 1.3's rule that an operation with a `double` gives a `double`.
2. **One operation at a time**. Students predict WidenPerOperation: `7 / 2 * 1.0` and `7 * 1.0 / 2`. Reveal 3.0 and 3.5. Java looks at each operator and its two operands in precedence order. In the first line, `7 / 2` is already 3 before any `double` appears.
3. **Never the other way**. Run NoAutoNarrow.java: `int miles = raceLength;` gives possible lossy conversion from `double` to `int`. The .2 has nowhere to go, so the compiler refuses to drop it silently. "Lossy" means something could be lost.

MISCONCEPTION A double anywhere in the line makes the whole line double

Students who see `* 1.0` or `2.0` somewhere in an expression assume every division in it keeps its decimal part. Widening happens one operation at a time: an `int` division that runs before the `double` is involved has already dropped the decimal part. Have students rewrite the expression one operation per line, labeling the type of each intermediate result, as on the board.

MISCONCEPTION Java will convert a double to an int if the value is whole

Students expect `int count = 12.0;` to compile because 12.0 has no decimal part. Run WholeDoubleToInt.java: same possible lossy conversion error as the "Without a cast" box on slide 6. The compiler checks the type, not the particular value.

Day 1 direct instruction: the (int) cast

Use slide 6.

1. **Casting** (1.5.A.1). A **cast** converts a value to another type on purpose. The **casting operator** is the type name in parentheses, written in front of the value: `(int) raceLength`. Show CastToInt: `(int) raceLength` is 26, and `raceLength` still holds 26.2, because the cast produces a new value and does not change the variable.
2. **Truncation** (1.5.A.2). `(int)` **truncates**: it drops every digit after the decimal point. It does not round. Students predict TruncateTowardZero (7.99, -7.99, 0.6, -0.6), then reveal 7, -7, 0, 0. For negative numbers truncation goes **up**, so the rule to remember is "toward zero."

MISCONCEPTION (int) rounds

Students expect `(int) 89.9` to be 90 because that is how people round grades. Run `CastIsNotRounding`: 89. The program runs, so this is a logic error, not a compiler error. Say the rule as "chop, don't round," and promise the rounding formula later in the period.

MISCONCEPTION Truncating a negative number goes down

Students apply "truncate means go down" and write -8 for `(int) -7.99`. Draw a number line: truncation erases the decimal digits, and -7 is closer to zero than -7.99. Ask: "Which digits did the cast remove?" The digits `.99`; what is left is `-7`.

Day 1 direct instruction: the (double) cast

1. **Casting an int to a double** (1.5.A.1). Back to the warm-up. Widening will not help `total / count` on its own, because neither operand is a `double`. `(double) total` produces `256.0`. Show `AverageCast` and write the steps on the board: the cast happens first, then `256.0 / count` widens 3, then a `double` division gives `85.33333333333333`. Line 5 displays 256: the variable is still an `int`. `CastOtherOperand` shows that casting `count` instead works just as well; one `double` operand is enough.
2. **A cast applies only to the value right after it** (slide 7, with 22 points over 5). Java does a cast before `*`, `/`, `%`, `+`, and `-`. To cast a whole expression, put the expression in parentheses.
3. **Predict: CastTooLate**. `(double) (total / count)` and `total / count` stored in a `double`. Both display `85.0`. The parentheses made the `int` division happen first, and casting 85 afterward cannot bring the `.333` back.
4. **Pause and try** (slide 8). Students predict the three lines for `hours = 7.75`, then reveal `7`, `7.0`, `7.75`: widening 7 back to a `double` does not return the `.75`, and the cast never changed `hours`.

MISCONCEPTION Casting after the division

`(double) (total / count)` reads like "make the average a `double`," and its result has a `.0`, so it looks right at a glance. It is still the whole-number answer. Give students a test: "Is the cast touching an operand of the division, or the result?" Cast an operand: `(double) total / count`.

Day 1 direct instruction: rounding to the nearest int

Start Part 2 (slide 9).

1. **The formula** (1.5.A.4, slide 10). To round a `double x` to the nearest `int`: `(int) (x + 0.5)` when `x` is zero or positive, and `(int) (x - 0.5)` when `x` is negative. Show why on the board with 3.5 up to 4.5: adding 0.5 moves every value that should round to 4 into 4.0 up to 5.0, and truncation does the rest. Run `RoundPositive` (`3.7 → 4`, `3.2 → 3`, `2.5 → 3`).
2. **Negative values** (slide 11). Run `RoundNegative`. `(int) (-2.7 + 0.5)` is `-2`, the wrong answer: adding 0.5 moved the value toward zero, and truncation moved it toward zero again. `(int) (-2.7 - 0.5)` is `-3`.

3. **The parentheses matter** (slide 12). Run `RoundNoParens.java`: `(int) distance + 0.5` casts only `distance`, then adds 0.5, so the result is the `double` 3.5, which cannot be stored in an `int`.
4. **Pause and try** (slide 13). Pairs pick `+ 0.5` or `- 0.5` for 12.5, 0.49, -2.5, and -0.6, then round each: 13, 0, -3, -1. Check that everyone used `- 0.5` for the negative values.

MISCONCEPTION Leaving out the parentheses in the rounding formula

Students write `(int) x + 0.5` because in their head they "cast and add." The cast applies only to `x`, and the sum is a `double`, so `int r = (int) x + 0.5;` does not compile. Connect it to the rule they just learned: a cast applies only to the value right after it.

MISCONCEPTION `+ 0.5` works for every value

Students memorize one formula and use it on negative values, which gives a logic error that is easy to miss. Have them check one negative value every time they round: -2.7 should become -3.

Day 1 guided practice: fix the quiz percentage

Students paste `PercentWrong` into the IDE. It should display the percentage of quiz questions answered correctly, rounded to the nearest whole percent: 7 out of 8 is 87.5%, which rounds to 88. It displays 0. Pairs find and fix the error, run their fix, and compare.

Debrief in two steps. `correct / total` is `int` division, so `7 / 8` is 0, and `0 * 100` is 0. Casting `correct` gives 87 (`PercentTruncated`), still not right, because `(int)` truncates 87.5. Adding 0.5 inside the cast gives 88 (`PercentFixed`).

TEACHER NOTE

If pairs are stuck after two minutes, ask: "What is `7 / 8` in Java?" Pairs that finish early can change `correct` to 5 and check that the program displays 63 (62.5% rounded).

Day 1 independent practice

Puzzle worksheet questions 1–4 (Part A and the first two Part B puzzles: truncation, cast placement, and rounding) are homework.

Day 1 closure

Jump to slide 22 (Chop, round, or wrap?). Pairs explain why `(int) 4.9` and `(int) (4.9 + 0.5)` differ, using the sentence starter; cold-call two pairs. Then ask: "A teacher's grade program has 89.9. When is 89 the right answer, and when is 90?" Look for "`(int)` truncates, which is right for counting complete things (whole miles, full hours); rounding with `+ 0.5` is right for a nearest value

such as a grade." Hand out the Cast, Chop, and Wrap puzzle worksheet. Day 2 picks up the deck at slide 14.

Day 2 warm-up: can an int hold 8.1 billion?

Show `TooLargeLiteral.java`, which stores 8,100,000,000 in an `int` (more than 8 billion people live on Earth). Ask: "Does it compile? If it runs, what does it display?" Take a vote, then compile it: `integer number too large`. Ask: "Why would a number be too large? A number is a number." Leave the question open until slide 15.

Day 2 direct instruction: the range of int values

Start Part 3 (slide 14).

1. **Fixed memory** (1.5.B.2). Every variable is a storage location, and the computer sets aside a fixed amount of memory for each value based on its type. An `int` always gets **4 bytes**. A **bit** is a single 0 or 1; a **byte** is 8 bits; so an `int` has 32 bits and $2 \times 2 \times \dots \times 2$ (32 times) = 4,294,967,296 possible patterns. That is how many `int` values exist. Go back to the odometer from the Day 1 warm-up (slide 2): a fixed number of digit wheels can show only so many readings.
2. **The limits** (1.5.B.1, slide 15). Half of the patterns are negative, and zero takes one of the nonnegative half. Run `IntRange`: `Integer.MAX_VALUE` is 2,147,483,647 and `Integer.MIN_VALUE` is -2,147,483,648. Each is a **constant**, a named value that never changes while the program runs. Read each name as a whole for now; the dot and the `Integer` class are explained in Topics 1.7 and 1.10.
3. **Inclusive**. Every `int` must be between the two limits, inclusive: both limits are themselves valid values. A literal outside the range, such as the warm-up's 8100000000, is caught by the compiler. Slide 15's 10-digit phone number is too big for the same reason.

MISCONCEPTION The two limits are mirror images

Students expect the smallest `int` to be -2,147,483,647. Point to the split: the negative side has 2,147,483,648 values and the nonnegative side has the same number, but one of those is zero. A memory hook: `MIN_VALUE` ends in 8, `MAX_VALUE` ends in 7.

Day 2 direct instruction: integer overflow

1. **Predict: OverflowMax**. `Integer.MAX_VALUE + 1`. Most students predict an error or 2147483648. Run it: -2147483648. If an `int` expression would produce a value outside the range, an **integer overflow** occurs (1.5.B.3): the result is an `int` inside the range, but not the expected one. Past the top, it wraps to the bottom. `OverflowMin` and slide 17 show the other direction: one less than `Integer.MIN_VALUE` is `Integer.MAX_VALUE`.

2. **No warning.** The program compiles and runs to the end. The compiler checks only literals written in the code; a sum or product is computed while the program runs. Run `GamePoints` (2,147,483,000 points plus 1,000 displays a large negative number).
3. **A real calculation.** Walk through slide 16 (seventy years in seconds): 70×365 is fine, but the full product, 2,207,520,000, is past the limit, so the program displays a negative number of seconds. Then show `MillisInMonth` and fill in a step table on the board. The last multiplication, $2,592,000 \times 1000$, is past 2,147,483,647, so the program displays a negative number of milliseconds. Counting in seconds instead keeps every step in range. Students do not need to predict the exact wrong value; they need to recognize when a step goes past the limit.

MISCONCEPTION Overflow is an error or an exception

Students expect Java to stop the program or refuse to compile it, as it did for `8100000000` and `/ by zero`. Overflow does neither: the program keeps running with a wrong value, which makes it a logic error. Contrast the warm-up and slide 17's phone number (literals, caught by the compiler) with `GamePoints` (a sum, never reported).

MISCONCEPTION The answer fits, so the calculation is safe

Run `AverageOverflow`: the average of two values of 2,000,000,000 is 2,000,000,000, which fits, but the sum 4,000,000,000 does not, and the program displays `-147483648`. Every step must stay in range, not only the final answer. `AverageOverflowFixed` halves each value first. (That fix is exact only because both values are even; the point is to notice which step overflows.)

Day 2 direct instruction: round-off error

Start Part 4 (slide 18).

1. **Predict: RoundOff.** Ask for the output of `0.1 + 0.2` and `1.0 - 0.9`. Reveal `0.30000000000000004` and `0.09999999999999998`.
2. **Why** (1.5.C.1, slide 19). A `double` is also stored in a fixed amount of memory, so it can keep only a limited number of digits: its **precision** is limited. In binary, 0.1 and 0.2 repeat forever, the way $1/3$ does in decimal, so Java stores the closest value it can. When a result is more precise than a `double` can store, it is rounded to the nearest value a `double` can hold. The small difference is a **round-off error**.
3. **Use int values when results must be exact.** Show `PriceDouble` (three items at \$1.10 displays `3.3000000000000003`) next to `PriceCents` ($110 \text{ cents} \times 3$ is exactly 330). Every whole number in the `int` range is stored exactly. This is why programs that handle money often count in cents; slide 19 shows the same idea with `1.1 + 2.2`.
4. **Pause and try** (slide 20). Pairs choose `int` or `double` for laps, a body temperature, a store total exact to the cent, and a 10-digit phone number. The phone number is too big for an `int`; since nobody does arithmetic with it, a `String` fits better.

MISCONCEPTION Round-off error is a bug in Java

Students conclude the computer "can't do math." Use the $1/3$ comparison: on paper you also have to stop writing 0.333... somewhere. A `double` stops sooner than students expect, and in binary even 0.1 never ends. It is a property of fixed memory, the same fact behind the `int` range.

MISCONCEPTION A product that should be whole will truncate correctly

Run `DollarsToCents: 0.29 * 100` is 28.999999999999996, and `(int)` truncates it to 28, losing a cent. Students expect 29 because the math on paper is exact. The fix combines two Day 1 ideas: round instead of truncating, `(int) (price * 100 + 0.5)`.

Day 2 guided practice: overflow and round-off experiments

Students run each program in the IDE. For each one, they predict first, run, record the output, and then make the change:

1. `GamePoints`: change 1000 to 600. Does it still overflow? (No: 2,147,483,600 is under 2,147,483,647.)
2. `MillisInMonth`: change 30 days to 24 days. Does the milliseconds line still overflow? (No: $24 \times 24 \times 60 \times 60 \times 1000$ is 2,073,600,000.)
3. `DollarsToCents`: change the price to 0.57. What do the three lines display, and which one is right? (56.99999999999999, 56, 57; the rounded line.)
4. `PriceCents`: change the price to 1999 cents and change the 3 to 4. What are the dollars and cents? (79 dollars, 96 cents.)

Debrief: in 1 and 2, the same code overflows or not depending on the values, which is why testing needs large inputs. In 3, rounding protects against round-off error before truncating. In 4, the `int` result is exact.

TEACHER NOTE

Check the numbers in step 3 on your own JDK before class; `0.57 * 100` displays 56.99999999999999 on current Java versions. If computers are not available, run the four changes on the projector and have students vote before each run.

Day 2 independent practice

Puzzle worksheet questions 5–10 (the face-off, be-the-compiler, two truths and a lie, one-token fix, who am I, and the cents reflection: the `int` range, integer overflow, round-off error, and a review of rounding) are homework. Collect the whole worksheet the next day and check it with the key.

Day 2 closure and exit ticket

Spend one minute on slide 21 (Words to know): point to each card and ask what Java does (a compiler error, a wrong value with no warning, a tiny difference, or dropped decimals) and how to avoid it. Then give the exit ticket (about five minutes). Question 1 checks casting, widening, and rounding (1.5.A), question 2 integer overflow (1.5.B), and question 3 round-off error (1.5.C). Re-teach the weakest one at the start of Topic 1.6.