

1.10 Calling Class Methods

Lesson Plan

Pacing: 2 class periods of 45 minutes

Learning objectives: 1.10.A

Essential knowledge: 1.10.A.1, 1.10.A.2

Materials

- Projector
- Whiteboard and markers
- Scrap paper or mini whiteboards for predictions
- IDE on every computer
- Calling on the Class slide deck
- Printed Calling on the Class puzzle worksheet and exit ticket

Agenda

Minutes	Activity
5	Day 1 warm-up — what comes before the dot? (slides 2–3)
12	Day 1 direct instruction — class methods and the keyword static (slides 4–7)
10	Day 1 direct instruction — calling with the class name and the dot operator (slides 8–10)
10	Day 1 guided practice — which one compiles? (slide 11), then write the call
5	Day 1 independent practice — puzzle worksheet questions 1–2
3	Day 1 closure — what static tells you
5	Day 2 warm-up — predict three Math calls
10	Day 2 direct instruction — arguments first, nested calls, results that are thrown away (slides 12–16)
10	Day 2 direct instruction — calls inside the defining class (slides 17–20)
10	Day 2 guided practice — trace TileFloor
3	Day 2 closure — when do you need the class name? (slides 21–22)
7	Day 2 exit ticket

Overview

Two periods on one learning objective: develop code to call class methods and determine the result of those calls (1.10.A). Day 1 covers what a class method is and how to spot one (1.10.A.1), and the usual way to call one, through the class name and the dot operator (1.10.A.2). Day 2 works out the value of calls, including nested calls, and then covers calls written inside the method's defining class, where the class name is optional (1.10.A.2).

The CED's suggested skills are 2.C (write code that uses procedural abstractions), 3.C (determine the result of that code), and 3.D (explain why code will not compile and fix it). Day 1 leans on writing calls; Day 2 leans on predicting results and fixing calls that do not compile.

Stay inside the topic boundary:

- The running example is `Math`. Use only `sqrt`, `abs`, and `pow`, and say what each does when it first appears: `sqrt` returns the square root as a `double`; `abs` returns the absolute value (the number without its minus sign) and has an `int` and a `double` version; `pow(a, b)` returns a raised to the power `b` as a `double`. The rest of the `Math` class is Topic 1.11.
- Students **read** class methods written in their own classes (`total`, `toSeconds`, `floorArea`, `printDivider`) and **call** them. They do not write method bodies of their own; that is Unit 3.
- The `String` header for `length` and the `Scanner` headers for `next`, `nextInt`, and `nextDouble` appear only as documentation to read, to show a header without `static`. Students have used the `Scanner` calls since Topic 1.4, but not as a lesson on objects. Calling methods on objects is Topics 1.12 to 1.14. Do not explain what an instance method is beyond "it belongs to each object, not to the class."
- Every header starts with `public`. Tell students once that `public` is explained in a later topic and can be read past. `static` is taught today.
- A label and a number still take a `print` followed by a `println`; joining text and numbers is Topic 1.15.

TEACHER NOTE

Every program below is in the topic's `examples/` folder. 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.

- **Day 1, calling with the class name:** `FirstCalls`, `TemperatureChange`, `SqrtOfText`, and `PowIntoInt`.
- **Day 1 guided practice:** `CallPractice` (the answers to "write the call").
- **Day 2 warm-up:** `WarmUpCalls`.
- **Day 2, working out calls:** `Hypotenuse`, `ArgumentFirst`, `IgnoredResult`, and `IgnoredResultFixed` (runnable versions of the ideas on slides 13–15).
- **Day 2, calls inside the defining class:** `BakeTimer`, `ScoreCard`, `NoClassName`, and `MissingStatic`.
- **Day 2 guided practice:** `TileFloor`.
- **Printed only (exit ticket):** `ExitCalls`. The puzzle worksheet and its key are separate Word files; their code is printed on the page.

The Calling on the Class deck covers both days: slides 1–11 on Day 1 and slides 12–22 on Day 2. Its speaker notes hold the answers to every Pause and Try slide. Before class, open `FirstCalls` and a scratch class in the IDE next to the deck. Compiler messages are quoted from a recent JDK; the wording can vary slightly between Java versions.

Day 1 warm-up: what comes before the dot?

Show slide 2: `console.nextInt()` and `Math.sqrt(25.0)`. Give pairs two minutes on "What comes right before the dot in each call?" Before the first dot is `console`, a variable made on the line above

with the `Scanner` pattern from Topic 1.4. Before the second dot is `Math`, and nobody ever made a `Math`. Leave the question "Where was `Math` ever made?" open, and read the goals on slide 3.

TEACHER NOTE

Keep this to five minutes. Do not explain `static` yet; if students guess at it ("it means it never changes," "it is a math thing"), collect the guesses and come back to them on slide 7. Tell students that `public` is explained in a later topic and can be read past.

Day 1 direct instruction: class methods and the keyword `static`

Start Part 1 (slide 4).

1. **Instances** (review of Topic 1.7). A class is a blueprint that can be used to make objects, also called **instances** of the class. Some jobs only make sense for one particular object: "read the next word typed" needs to know which `Scanner`, and "how long is this text?" needs to know which text. Ask, "What would `length` measure if there were no `String` to measure?"
2. **Class methods** (1.10.A.1, slide 5). A square root or an absolute value needs nothing except the number you hand it. No object is involved; the job belongs to the `Math` class as a whole. A **class method** belongs to the class itself, not to any instance of it, so it can be used without making any objects. Use the school-office picture: you ask "the office" for the date of the next holiday, because the answer belongs to the school; you ask a particular student for their locker number. Class methods are like the office. Point out where the picture stops working: a class is code that runs when called, not someone who answers.
3. **The keyword `static`** (1.10.A.1, slides 5–6). A class method has `static` in its header before the return type. Slide 5 puts the two `abs` headers, with `static`, next to the `Scanner` headers next and `nextInt`, without it. Take `public static int abs(int a)` apart word by word with slide 6: `public` (later topic), `static` (class method), `int` (return type), `abs` (name), `(int a)` (parameter list). Every method in `Math` has `static`. The `String` header `public int length()` does not: each `String` has its own `length`.
4. **You have been writing one all along.** Ask, "Which method in every program you have written has `static` in its header?" `public static void main(String[] args)`. `main` is a class method, which is how Java can start your program without first making any object.
5. **`static` does not mean "never changes"** (slide 7). `static` is not part of the return type, and the result is not fixed: `Math.abs(-3)` and `Math.abs(-40)` run the same method and return `3` and `40`. `static` says who the method belongs to. That also answers the warm-up: `Math` was never made because `sqrt` belongs to the `Math` class itself.

MISCONCEPTION `static` describes the value that comes back

Because `static` sits next to the return type, students read `static double` as a special kind of `double`, or think `static` means "the result never changes." Give two calls with different results (`Math.sqrt(49.0)` is `7.0`, `Math.sqrt(16.0)` is `4.0`), then have students point to the return type: it is always the single word just before the method name. Ask "Who does this method belong to?" whenever `static` comes up.

Day 1 direct instruction: calling with the class name and the dot operator

Start Part 2 (slide 8).

1. **The pattern** (1.10.A.2, slide 9). Ask, "Many classes could have a method named `sqrt`. How does Java know which one you mean?" You tell it: class name, a dot, the method name, and the arguments in parentheses. Write `ClassName.methodName(arguments)` on the board. The dot is the **dot operator**; read `Math.sqrt(49.0)` as "go into `Math` and run `sqrt` with `49.0`." Everything after the dot follows Topic 1.9's rules: the arguments must match the parameter list in number, order, and type.
2. **Predict: FirstCalls**. Slide 9's two calls print `9.0` and `6.5`; `FirstCalls` runs the same pattern in the IDE. `Math.sqrt(49.0)` returns `7.0`, which is stored in `side`; `Math.abs(-12.5)` returns `12.5`, which is used in `12.5 + 2.5`. Output: `7.0`, then `15.0`. Both methods are non-void, so the returned value has to be stored or used (Topic 1.9).
3. **Predict: the header decides the type** (slide 10). Students predict the three lines before you reveal `7`, `2.5`, `6.0`, then walk the slide's table: `abs` is overloaded (Topic 1.9), so the `int` argument runs `int abs(int a)` and the `double` argument runs `double abs(double a)`; `sqrt` has only a `double` version. Run `TemperatureChange` for the same idea with new numbers: `Math.abs(3 - 12)` runs the `int` version of the overloaded `abs` and returns the `int` `9`. `Math.sqrt(16)` has only one version, with a `double` parameter; the `int` `16` is accepted as `16.0`, and the result prints as `4.0`. The type that comes back is the return type in the header of the method that runs.
4. **Arguments still have to fit**. Show `SqrtOfText: Math.sqrt("25")` gives incompatible types: `String` cannot be converted to `double`. The class name and dot reach the right method; the argument must still match its parameter.

MISCONCEPTION The result has the same type as the argument

Students expect `Math.sqrt(16)` to print `4`, because `16` is an `int`. Send them back to the header every time: `sqrt` returns a `double`, so the answer is `4.0` whatever goes in. The contrast with `abs`, which has an `int` version, is the point of slide 10 and `TemperatureChange`: Java picks the version of `abs` whose parameter matches, and that version's return type decides the result.

MISCONCEPTION Any value that looks like a number can be passed

Students write `Math.sqrt("25")` because the quotes feel like a way to "type in" the number. Text in quotes is a `String`, and `sqrt` takes a `double`. Run `SqrtOfText` so students see the message, then remove the quotes.

Day 1 guided practice: which one compiles, then write the call

Start with slide 11: which of four lines stores `2` to the `5th`? Students vote, then debrief. `A` is right (`32.0` in a `double`); `B` does not compile (possible lossy conversion from `double` to `int`, run

`PowIntoInt` to show the message); C swaps the class and method names; D puts the method name inside the parentheses.

Then write the four tasks below on the board. Pairs write one statement for each task on mini whiteboards, then hold them up:

Task	Statement	Value
Store the square root of 2.25 in <code>side</code> .	<code>double side = Math.sqrt(2.25);</code>	1.5
Print the absolute value of -40.	<code>System.out.println(Math.abs(-40));</code>	40
Store 3 to the power 4 in <code>power</code> .	<code>double power = Math.pow(3, 4);</code>	81.0
With <code>int low = 18;</code> and <code>int high = 25;</code> , store the distance between them in <code>gap</code> .	<code>int gap = Math.abs(low - high);</code>	7

Debrief each one, then run `CallPractice` to check the values. Expect two errors and use them: `int power = Math.pow(3, 4);` (the same possible lossy conversion as slide 11's line B, because `pow` returns a `double`), and `double side = sqrt(2.25);` with no class name. Do not explain the second one fully yet; say that Java needs to be told which class `sqrt` is in, and that Day 2 shows the one place where the class name can be left out.

TEACHER NOTE

For the last task, both `Math.abs(low - high)` and `Math.abs(high - low)` are correct. Storing the result in a `double` also compiles; ask why `int` is the better fit (the `int` version of `abs` runs, so the result is an `int`).

Day 1 independent practice

Hand out the Calling on the Class puzzle worksheet. Students do Part A, questions 1–2 (the type a call returns, and sorting headers into class and instance methods), in class; whatever is not finished is homework. Circulate and check question 1: students who choose 13 are reading the argument's type instead of `sqrt`'s return type. The rest of the worksheet uses calls from another class, nested calls, and arguments worked out first, so it waits for Day 2.

Day 1 closure

Ask students to finish two sentences: "A method with `static` in its header belongs to ..." and "To call it from my program, I write ..." Listen for "the class, not an object" and "the class name, a dot, the method name, and the arguments."

Day 2 warm-up: predict three Math calls

Show `WarmUpCalls` and have students write the exact output: 15, 9.0, 2.5. The second line is the one to check: `pow` returns a `double`, so `Math.pow(3, 2)` prints 9.0, not 9. Ask one student to read each call aloud as "go into Math and run ...".

Day 2 direct instruction: arguments first, nested calls, results that are thrown away

Start Part 3 (slide 12).

1. **The rule** (1.10.A, determine the result). Build it from Topic 1.3 and Topic 1.9: (1) Java works out each argument completely, using the usual rules, including integer division; (2) the method runs with a copy of each value; (3) the whole call is replaced by the value it returns, and the expression continues. When calls are nested, the inner call is an argument of the outer one, so work from the inside out.
2. **Inside out** (slide 13). Students write the value of `d` before you reveal the table, then walk it one row at a time: `Math.sqrt(36.0 + Math.pow(8, 2))`, `Math.sqrt(36.0 + 64.0)`, `Math.sqrt(100.0)`, `10.0`. Each row replaces one call with its return value. `sqrt` cannot start until its whole argument is one value. Hypotenuse (legs 3 and 4, 5.0) is the same calculation in the IDE.
3. **Predict: one character apart** (slide 14). `Math.sqrt(25 / 4)` receives 6, not 6.25, because `25 / 4` is integer division and the argument is finished before `sqrt` runs; it prints about 2.449. `Math.sqrt(25.0 / 4)` receives 6.25 and prints 2.5. `ArgumentFirst` is the runnable version with `9 / 4`.
4. **A call is a question, not a command** (slide 15). `Math.abs(temp);` compiles, and `temp` still prints `-8.0`: the call asked "what is the absolute value of `temp`?" and threw the answer away; the method only received a copy of `temp` (call by value, Topic 1.9). `temp = Math.abs(temp);` stores the answer and prints `8.0`. `IgnoredResult` and `IgnoredResultFixed` show the same thing in the IDE.
5. **Pause and try** (slide 16). Students write one step per line for
`Math.abs(5 - 9) + Math.sqrt(Math.pow(3, 2))`: `Math.abs(-4)` returns `4`, `Math.pow(3, 2)` returns `9.0`, `Math.sqrt(9.0)` returns `3.0`, and `4 + 3.0` prints `7.0`. If someone says 7, ask what type `sqrt` returns.

MISCONCEPTION The method works on the expression as written

Students picture `Math.sqrt(25 / 4)` as "the square root of twenty-five fourths" and write 2.5. The method never sees `25 / 4`; it sees the value Java already computed, 6. Have students rewrite the argument as a single value before they think about the method at all. Slide 14's pair is the fix: same method, one character apart, different answers.

MISCONCEPTION `Math.abs(balance);` makes balance positive

The call reads like a command, so students expect it to change the variable. There is no error message, which makes this mistake hard to find. Ask of every non-void call: "Where does the returned value go?" If the answer is "nowhere," the call did nothing your program can see.

Day 2 direct instruction: calls inside the defining class

Start Part 4 (slide 17).

1. **Your own class methods.** In Topic 1.9 students read methods written in their own classes. If such a method's header includes `static`, it is a class method of that class. On slide 18, `total` is a class method of `Tickets`.
2. **The defining class** (1.10.A.2, slide 18). The class a method is written in is its **defining class**. When the call is inside the defining class, the class name is optional: `total(2, 3)` leaves it out and `Tickets.total(1, 1)` includes it. Both run the same method; the output is 48, then 20. `BakeTimer` shows the same thing in the IDE (`toSeconds(3)` and `BakeTimer.toSeconds(12)` print 180, then 720).
3. **Void class methods too.** Run `ScoreCard` from the IDE: `printDivider();` and `ScoreCard.printDivider();` both print the divider line. The class-name rule is the same for void and non-void class methods; a void call still stands alone as a statement.
4. **Predict: outside the defining class** (slide 19). After seeing `total(2, 3)` work, ask whether `sqrt(64.0)` will work inside `Garden`. It does not: `cannot find symbol, symbol: method sqrt(double), location: class Garden`. `sqrt` is defined in `Math`, so leaving out the class name makes Java look only in `Garden`. Every call to `Math` from your programs needs `Math.`. Run `NoClassName` to show the message live.
5. **Forgetting static.** Show `MissingStatic`: the header of `toMinutes` has no `static`. The compiler reports `non-static method toMinutes(int) cannot be referenced from a static context`. `main` is a class method, so no object exists while it runs, and a method without `static` belongs to an object. Adding `static` fixes it. Do not go further into objects; Topics 1.12 to 1.14 cover them.
6. **Pause and try** (slide 20). `Timer` has `toSeconds`; the three calls are written in `Race`. The answer is B, II only: I gives `cannot find symbol` because Java looks only in `Race`, and III writes `timer` with a lowercase `t`, which is not the class name.

MISCONCEPTION The class name can always be left out

After `total(2, 3)` works, students try the same shortcut with `Math`. Give them one question to ask of every call: "Which class is this method written in, and which class is this call in?" If they are the same class, the class name is optional. If they are different, it is required. The message `location: class Garden` on slide 19 shows exactly where Java looked.

MISCONCEPTION Leaving out static makes no difference

Students copy a header and drop `static` because it "looks extra." The message about a "static context" is hard to read, so translate it: `main` is a class method and has no object, and a method without `static` needs one. When students see that message in this unit, the first thing to check is the called method's header for `static`.

Day 2 guided practice: trace `TileFloor`

Pairs open `TileFloor` in the IDE (or read it on the projector). Before running it, they do two things:

1. Label each call on lines 10, 11, 14, and 15: is the class name **required**, **optional and used**, or **optional and left out**?
2. Trace `main` and write the exact output.

Line	Call	Class name	Value
10	<code>floorArea(3.5, 4.0)</code>	Optional, left out	kitchen is 14.0
11	<code>TileFloor.floorArea(1.5, 6.0)</code>	Optional, used	hall is 9.0
14	<code>Math.abs(hall - kitchen)</code>	Required	argument 9.0 - 14.0 is -5.0; returns 5.0
15	<code>Math.sqrt(kitchen + 2.0)</code>	Required	argument 16.0; returns 4.0

```
14.0
9.0
5.0
4.0
```

Debrief with two questions: Why can line 10 leave out the class name but line 15 cannot? (`floorArea` is defined in `TileFloor`, the class the call is in; `sqrt` is defined in `Math`.) What would line 14 print if it were `Math.abs(kitchen - hall)`? (5.0 as well; the absolute value is the same either way.)

Day 2 independent practice

Puzzle worksheet questions 3–10 are homework (about 25 minutes): code tiles for a program that reads input with `Scanner` (Topic 1.4) and calls its own class method, a snippet pool with a call to another class, a face-off between `toCups(8)` and `Recipe.toCups(8)`, a Who Am I?, deciding which calls to another class compile, a one-token fix for integer division in an argument, a result that is thrown away, and a reflection on when the class name can be left out.

Day 2 closure

Point to a card on slide 21 and have a student give an example from today. Then show slide 22: pairs explain why `Math.sqrt(49.0)` needs `Math.` in their program but `total(2, 3)` did not need `Tickets.` inside `Tickets`, using the sentence starter. Cold-call two pairs. Listen for "the class name can be left out only inside the defining class; everywhere else Java looks only in the class the call is in." If time allows, ask the class about `toSeconds(3)` written in a class other than `BakeTimer` (required; it would have to be `BakeTimer.toSeconds(3)`).

Day 2 exit ticket

Give the exit ticket (about seven minutes). Question 1 checks determining the result of nested class method calls (1.10.A); question 2 checks `static` and when the class name is required (1.10.A.1, 1.10.A.2). Re-teach the weaker one at the start of Topic 1.11, which uses `Math` calls throughout.