

1.9 Method Signatures

Lesson Plan

Pacing: 2 class periods of 45 minutes

Learning objectives: 1.9.A, 1.9.B

Essential knowledge: 1.9.A.1, 1.9.A.2, 1.9.B.1, 1.9.B.2, 1.9.B.3, 1.9.B.4, 1.9.B.5

Materials

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

Agenda

Minutes	Activity
5	Day 1 warm-up — one name, many kinds of values (slides 2–3)
10	Day 1 direct instruction — methods, blocks, calls, and the flow of control (slides 4–6)
6	Day 1 direct instruction — procedural abstraction and reading documentation (slide 7)
12	Day 1 direct instruction — headers, parameters, arguments, and signatures (slides 8–11)
8	Day 1 guided practice — which calls compile? (slide 12)
4	Day 1 closure — parameter, argument, signature, then hand out the puzzle worksheet
5	Day 2 warm-up — signatures from memory
10	Day 2 direct instruction — calls, return values, void and non-void methods (slides 13–15)
7	Day 2 direct instruction — call by value (slide 16)
7	Day 2 direct instruction — overloaded methods (slides 17–19)
8	Day 2 guided practice — trace QuizReport
3	Day 2 closure — words to know and read this header out loud (slides 20–21)
5	Day 2 exit ticket

Overview

Two periods on two learning objectives: identify the correct method to call from documentation and method signatures (1.9.A), and describe how to call methods (1.9.B). Day 1 covers what a method is, the flow of control, procedural abstraction, and how a call's arguments must match a signature (1.9.A.1, 1.9.A.2, 1.9.B.3, 1.9.B.5). Day 2 covers return values, void and non-void methods, call by value, and overloading (1.9.B.1–1.9.B.4), then puts everything together in one trace.

The CED's suggested skill is 3.C, determining the result or output of code that contains procedural abstractions. Most of class time is prediction: students commit to an output or a "compiles / does not compile" before you run anything.

Stay inside the topic boundary:

- Students **read** methods; they do not write their own methods until Unit 3. Every method in the deck and the puzzle worksheet is already written; students read headers, trace calls, and decide whether calls compile.
- Every method header starts with `public static`. Tell students once that those words are explained in a later topic and can be read past for now.
- `Math.sqrt`, `Math.pow`, `Math.abs`, and `Math.random` are called through the class name. Say what each does where it is used; Topic 1.10 explains calls through a class name fully, and Topic 1.11 covers the whole `Math` class.
- The `String` headers for `length`, `indexOf`, and `substring` appear only as documentation to read. Students do not call `String` methods until Topic 1.14, and `substring` is taught in Topic 1.15.
- 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, methods and flow of control:** `NeverCalled`, `ReminderTwice` (the program on slide 5), `Announcements`, `NestedCalls`, `DefinedFirst`, and `MissingParens`.
- **Day 1, procedural abstraction:** `SquareRoot`, `Ladder`, and `PowGuess`.
- **Day 1, parameters and signatures:** `TicketTotal`, `TicketVariables`, `ParameterOutside`, `PowOrder`, `SqrtInt`, `SqrtTwoArgs`, `SqrtString`, and `TicketSwapped`.
- **Day 2, return values:** `ReturnValues`, `MinutesBack`, `VoidInExpression`, `PrintVoid`, `IgnoredReturn`, and `IgnoredReturnFixed`.
- **Day 2, call by value:** `BonusCopy`, `BonusPoints`, and `SameName`.
- **Day 2, overloading:** `AbsOverload`, `Receipt`, `OrderOverload`, and `SameSignature`.
- **Day 2 guided practice:** `QuizReport`.
- **Printed only (exit ticket):** `ExitRefill`. The puzzle worksheet and its key are separate Word files; their code is printed on the page.

The Read the Header deck covers both days: slides 1–12 on Day 1 and slides 13–21 on Day 2. Its speaker notes hold the answers to every Pause and Try slide. Before class, open `NeverCalled`, `TicketTotal`, 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: one name, many kinds of values

Show slide 2. All five lines call `println` with an `int`, a `double`, a `String`, and nothing at all. Give pairs two minutes on "How can one method accept all of these?" Collect guesses such as "it

checks the type" and "there are several `println`," and leave the question open: the answer is overloading, on Day 2 (slide 18). Read the goals on slide 3.

TEACHER NOTE

Keep this to five minutes. Tell students `public static` is explained in a later topic and can be read past, and that `void` is explained on Day 2.

Day 1 direct instruction: methods, blocks, calls, and the flow of control

Start Part 1 (slide 4).

1. **Predict: `NeverCalled`.** Show it and ask students to write its exact output. Many will write three lines, including "Bring your library book." Run it: only two lines appear. Ask, "Line 11 is a perfectly good statement. Why did it not run?"
2. **Block and method** (1.9.A.1, slide 5). Any section of code enclosed in braces is a **block**. A **method** is a named block of code that runs only when it is **called**. In `NeverCalled`, lines 10–12 are the block that belongs to `printReminder`, and nothing calls it. Students have been calling methods since Topic 1.1: `System.out.println` is one, and `main` is the method Java calls for them when the program starts.
3. **A call.** Slide 5 shows the same method with two calls; `ReminderTwice` is the runnable version. A call is the method's name, parentheses, and a semicolon. The reminder appears twice, and its wording lives in one place. Because the method is in the same class as `main`, the call is just its name; Topic 1.10 explains the full rules for calling methods.
4. **Flow of control** (1.9.B.5). The order statements actually run in is the **flow of control**. A call interrupts the top-to-bottom order: Java runs every statement in the method, then returns to the point immediately after the call. Run `Announcements`: the two lunch lines appear between `main`'s lines, exactly where the call was. Use the recipe picture: "make the sauce (see page 40)", do page 40, come back to the next step.
5. **Calls inside calls.** Run `NestedCalls`. When `printLunchMenu` finishes, control returns to `printSchedule` (whoever called it), not to `main`. Each method returns to its caller.
6. **Predict: `DefinedFirst`.** The method is written above `main`. Students predict, then run: `First period, The bell rings., Second period.` The program always starts in `main`, and where a method is written in the file does not matter.
7. **The parentheses are the call** (slide 6). `printReminder;` does not compile: `not a statement`. Run `MissingParens` to show the message live.

MISCONCEPTION Methods run in the order they are written in the file

Students read a file from the top line to the bottom line, so they expect a method written above `main` to run first and a method that is never called to run anyway. `NeverCalled` and `DefinedFirst` are the fix: let students commit to the wrong prediction, then run the program. Give them one rule for every trace: start at the first line of `main`, and follow the calls.

MISCONCEPTION The method's name alone runs it

Students write `printReminder;` and expect the reminder. Slide 6 and `MissingParens` show that Java reports not a statement. Without parentheses, the name is just a name; the parentheses make it a call.

Day 1 direct instruction: procedural abstraction

1. **What, not how** (1.9.A.1). Ask, "How does `System.out.println` get letters onto the screen?" Nobody knows, and everyone has used it since Topic 1.1. **Procedural abstraction** lets a programmer use a method by knowing what it does, even without knowing how it was written. A driver turns the steering wheel without knowing how the steering system is built.
2. **Documentation is the whole view** (slide 7). The slide shows the documentation comment and header for `Math.random`: it needs nothing and gives back a `double` from 0.0 up to, but not including, 1.0. Nobody in the room knows how it picks the number, and nobody needs to. Then say the documentation for `Math.sqrt` aloud: "Returns the positive square root of a," with the precondition that a is not negative (Topic 1.8). Run `SquareRoot (12.0, then 1.4142135623730951)`. The code that finds square roots is nowhere in sight.
3. **Reuse**. Run `Ladder`: the same method, a new problem (a ladder 3 meters up a wall and 4 meters out is 5.0 meters long). One well-documented method serves many programs.
4. **Predict: PowGuess**. `Math.pow(a, b)` gives a raised to the power b. Students predict `Math.pow(2, 3)`. Most say 8; it prints 8.0, because the documentation's header, `public static double pow(double a, double b)`, says the method gives back a `double`. Procedural abstraction lets you skip the *how*, never the *what*.

MISCONCEPTION Guessing what a method does from its name

Students treat abstraction as permission not to read anything. They predict 8 for `Math.pow(2, 3)`, or pass a negative number to `Math.sqrt`. Make "What goes in? What comes out? Any precondition?" the three questions students ask of every method's documentation before they call it.

Day 1 direct instruction: headers, parameters, arguments, and signatures

Start Part 2 (slide 8).

1. **Method header** (1.9.A.2, slide 9). Take apart `public static void printTotal(double price, int qty): public static` (later topic), the **return type** `void` (Day 2), the name, and the **parameter list** in parentheses. A **parameter** is a variable declared in the header that the method's body can use; it is how values get into the method.
2. **Method signature** (1.9.A.2, slide 9). The signature is the method's name and the ordered list of its parameter types: `printTotal(double, int)`. A method with no parameters has an empty list: `random()`. Walk the slide's three rows, header to signature. Ask, "Why are the return type and

the parameter names left out?" Because a call shows neither; the signature holds exactly what Java can check against a call.

3. **Arguments** (1.9.B.3, slide 10). An **argument** is a value passed into a method when it is called. On the slide, `printTotal(4.5, 3)` starts `price` as 4.5 and `qty` as 3 and prints 13.5. The first parameter gets the first argument, the second gets the second. Run `TicketTotal` for the same idea with two calls: each call starts the method fresh with new values.
4. **Arguments are values**. Run `TicketVariables`: the arguments are a variable and the expression `groupSize + 1`, and their names have nothing to do with the parameter names. Java evaluates each argument and passes the value (38.0, then 47.5). Matching is by position, never by name.
5. **Parameters stay in their method**. Show `ParameterOutside`: `main` tries to print `tickets` and gets `cannot find symbol`.
6. **Matching a call** (1.9.B.3, slide 11). Arguments must be compatible with the signature in **number, order, and type**. An `int` argument may go into a `double` parameter (widening, Topic 1.5): `SqrtInt` prints 7.0. A `double` may not go into an `int` parameter. On the slide, `printTotal(3, 4.5)` does not compile: 4.5 would go into `int qty`, and Java reports `incompatible types: possible lossy conversion from double to int`. `TicketSwapped` is the runnable version. Show `PowOrder`: `pow(2.0, 5.0)` is 32.0 but `pow(5.0, 2.0)` is 25.0. Run `SqrtTwoArgs` and `SqrtString` for the wrong-number and wrong-type messages.

MISCONCEPTION A parameter can be used anywhere in the program

Students expect `tickets` to exist in `main` after the call. A parameter belongs to its own method's block. Point at `ParameterOutside`'s `cannot find symbol`, and ask: "Where was `tickets` declared?" If `main` needs the value, `main` keeps it in its own variable.

MISCONCEPTION Java matches arguments to parameters by name or by meaning

Students remember which values a method needs but not their order, and expect Java to sort them out. Java matches by position only. When the types differ, a swap is a compile error (slide 11, `TicketSwapped`); when both parameters have the same type, a swap compiles and gives a wrong answer (`PowOrder`). Have students write the signature next to every call they check.

Day 1 guided practice: which calls compile?

Show slide 12: the header `public static void printTotal(double price, int qty)` and four calls. Pairs write the signature, then mark each call "compiles" or "does not compile" with a reason that names number, order, or type, and write the output of each call that compiles. Debrief one call at a time:

Call	Compiles?	Why
<code>printTotal(2.0, 5);</code>	Yes	A <code>double</code> , then an <code>int</code> . It prints 10.0.
<code>printTotal(5, 2.0);</code>	No	Order: 2.0 would go into the <code>int</code> parameter <code>qty</code> .
<code>printTotal(2, 5);</code>	Yes	The <code>int</code> 2 widens to the <code>double</code> 2.0. It prints 10.0.

<code>printTotal(2.0);</code>	No	Number: one argument, two parameters (argument lists differ in length).
-------------------------------	----	---

TEACHER NOTE

Call C is the one to linger on: many students mark it wrong because 2 "is not a double." The rule is "compatible with" the parameter type, not "exactly the same as." If computers are available, pairs can paste the `printTotal` method from slide 10 into a scratch class and try all four calls in `main`.

Day 1 independent practice

Puzzle worksheet question 1 (Pencil Check: return types and signatures from headers) is homework. The rest of the worksheet uses return values, void methods, call by value, and overloading, so it waits for Day 2.

Day 1 closure

Ask students to finish three sentences in their notes: "A parameter is ...", "An argument is ...", and "A method signature is ...". Call on three students, one for each. Listen for "declared in the header", "the value passed in a call", and "the name and the ordered parameter types". Hand out the Read the Header puzzle worksheet. Day 2 picks up the deck at slide 13.

Day 2 warm-up: signatures from memory

Students write the signatures for `public static int abs(int a)` and `public static void printTotal(double price, int qty)`, then decide which of `printTotal(3, 4.5);` and `printTotal(4.5, 3);` compiles. Answers: `abs(int)`, `printTotal(double, int)`, and only the second call. Re-teach the order rule with slide 11 if more than a few students miss the last part. Check puzzle worksheet question 1 quickly against the same rule.

Day 2 direct instruction: void and non-void methods, return values

Start Part 3 (slide 13).

1. **Jump in, run, come back** (1.9.B.5, slide 14). Trace `toInches` on the board. The second line of `main` needs `toInches(3)` before it can add 4, so control jumps into the method, which prints `converting`. The **return statement** sends 36 back and ends the method. Control returns to the middle of the same line, where the call is replaced by 36, and `36 + 4` is 40. Output: `measuring, converting, 40`. `MinutesBack` is the same idea in the IDE (`Converting`, then 135).
2. **Two kinds of methods** (1.9.B.1, 1.9.B.2, slide 15). The return type says which kind. A **void method** gives back nothing (a doorbell): `printTotal`, `System.out.println`. Its call is used only

as a statement on its own, never as part of an expression. A **non-void method** gives back a **return value** of exactly its return type (a calculator): `toInches` returns an `int`, `sqrt` and `pow` return a `double`.

3. **Using a return value.** The caller must **store it in a variable** or **use it in an expression**. Show `ReturnValues` and replace each call with its value on the board: `double side = 9.0;`, then `System.out.println(15 + 5);`. It prints 9.0 and 20. Explain absolute value once: a number's distance from zero.
4. **Predict: a void call used as a value.** Slide 15's `double t = printTotal(4.5, 3);` does not compile: incompatible types: void cannot be converted to double. Run `VoidInExpression` for the same message, and `PrintVoid`, which puts the call inside `println` and gets `'void' type not allowed here.`
5. **Predict: a return value thrown away.** Slide 15's `toInches(5);` compiles, but its 60 is thrown away. Show `IgnoredReturn`. It compiles and prints -8: `Math.abs(temperature)` returned 8, but nothing stored or used it. `IgnoredReturnFixed` stores the result and prints 8.

MISCONCEPTION A void method's call can be stored or printed

Students see `printTotal(4.5, 3)` print 13.5 and assume the call "is" 13.5. Printing is not returning. A void method has no value to store, so its call must stand alone. Point to the return type before every call: `void` means a statement on its own line.

MISCONCEPTION Calling a non-void method is enough

`Math.abs(temperature);` compiles, so students believe it worked. The value came back and was thrown away. There is no error message, which makes this mistake harder to find than the void one. Ask of every non-void call: "Where does the returned value go?"

Day 2 direct instruction: call by value

1. **Predict: BonusCopy.** `main` passes `myScore` (80) to `addBonus`, which adds 10 to its parameter and prints it. Students write both lines of output. Many predict 90 and 90. Run it: 90, then 80.
2. **Call by value** (1.9.B.3, slide 16). Java passes arguments using **call by value**: each parameter is initialized with a **copy** of its argument's value. The parameter is a separate variable, so changing it changes only the copy. Walk the slide's table with one column for `count` and one for `n`: `addOne` prints 6, and `main` still prints 5. The photocopy picture works: scribbling on the copy leaves the original untouched.
3. **Same name, still two variables.** Run `SameName` (the guide's version, where `main`'s variable is also called `score`): the output is the same as `BonusPoints`, Inside `addBonus`: 90 and Back in `main`: 80. Each method sees only its own variable; the call copies the value from one to the other, and that is the only connection.

MISCONCEPTION The method changed its parameter, so my variable changed too

This is the most common mistake with methods. Students picture the variable itself traveling into the method. Use the two-column trace from slide 16 every time a variable is passed

today, with the parameter's column marked "gone" after the method finishes. Then ask: "If a method cannot change my variable, how could it get a new value back to me?" Answer: it returns the value, and the caller stores it (slide 14).

Day 2 direct instruction: overloaded methods

Start Part 4 (slide 17).

1. **Same name, different signatures** (1.9.B.4, slide 18). Two methods named `area` have different parameter lists, so different signatures: `area(int)` and `area(int, int)`. `area(4)` prints 16 and `area(4, 5)` prints 20. Methods are **overloaded** when several share a name but have different signatures. Java compares the call's arguments with each signature and runs the one that matches. Back to the warm-up: `println` is overloaded the same way, with `println(int)`, `println(double)`, `println(String)`, `println()`, and more.
2. **Predict: AbsOverload**. `Math.abs` has two versions, `abs(int)` and `abs(double)`. `-7` is an `int` literal and `-7.0` is a `double` literal, so they pick different versions: 7, 7.5, 7.0.
3. **Different number or order**. The `String` documentation lists `substring(int)` and `substring(int, int)`. If time allows, run `Receipt (4.25, then 8.5)` for overloads that differ in number, and `OrderOverload (Version 2)` for overloads that differ only in the order of the types.
4. **Not overloading**. Slide 18's last line: adding `double area(int length)` fails, because its signature is `area(int)` again. `SameSignature` shows the same thing in the IDE: two methods named `half` that differ only in return type and parameter name, and Java reports method `half(int)` is already defined. A call shows neither the return type nor the parameter names, so neither can tell two methods apart.
5. **Pause and try** (slide 19). Students decide which of three `show` methods each call runs. A runs `show(int)`, B runs `show(double)`, C runs `show(int, double)` because 4 widens to 4.0, and D does not compile: 3.0 cannot go into `int n`.

MISCONCEPTION Changing the return type or a parameter name makes a new method

Students think any change to a header creates an overload. Have them write the signatures of both headers side by side. If the signatures match, Java cannot tell the methods apart. Only the number, types, or order of the parameters can make two signatures different.

Day 2 guided practice: trace QuizReport

Pairs open `QuizReport` in the IDE (or read it on the projector). First they make a small table on paper giving each header's signature and saying whether it is void or non-void. Then they trace `main` with columns for `quiz` and `curved` and write the exact output before running it:

```
Score: 78
Score: 83
Points from goal: 7
```

Debrief with four questions: Where does control go at each call? (Line 6 jumps into `addCurve`, whose lines 16–17 return 83 to the middle of line 6.) Why does `quiz` still print 78 after `score += points`? (Call by value: `score` is a copy.) Why does line 6 store a result but lines 7 and 8 do not? (`addCurve` is non-void; `printScore` is void.) Which `abs` runs on line 9? (`abs(int)`, because `curved - 90` is an `int`.) The study guide's "Putting it together" section has the full trace table.

Day 2 independent practice

Puzzle worksheet questions 2–9 are homework (about 25 minutes): tracing a call into a method and back, deciding which calls compile, a snippet pool, a void versus non-void face-off, overloaded `abs` headers, call by value, and a reflection on parameters and arguments.

Day 2 closure and exit ticket

Point to a card on slide 20 and have a student give an example of each word from today. Then show slide 21: pairs read `public static String initials(String first, String last)` aloud with the sentence starter, and you cold-call two pairs. Listen for "returns a `String`," "needs two `String`s in that order," and "signature `initials(String, String)`." Then give the exit ticket (about five minutes). Question 1 checks flow of control, call by value, and return values (1.9.B.2, 1.9.B.3, 1.9.B.5); question 2 checks signatures and matching calls (1.9.A.2, 1.9.B.3). Re-teach the weaker one at the start of Topic 1.10.