

1.8 Documentation with Comments

Lesson Plan

Pacing: 1 class period of 45 minutes

Learning objectives: 1.8.A

Essential knowledge: 1.8.A.1, 1.8.A.2, 1.8.A.3

Materials

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

Agenda

Minutes	Activity
5	Warm-up — what is this program for? (slide 2)
9	Direct instruction — comments are ignored, commenting out, and <code>//</code> comments (slides 3–8)
9	Direct instruction — <code>/* */</code> block comments and <code>/** */</code> Javadoc comments (slides 9–15)
8	Direct instruction — preconditions and postconditions (slides 16–20)
7	Guided practice — label, run, and break PizzaShare
3	Closure — words to know and before and after (slides 21–22)
4	Exit ticket

Overview

One period on a single learning objective (1.8.A): describe the functionality and use of code through comments. It has three parts:

- **Comments** (1.8.A.1): notes for the original programmer and other programmers, ignored by the compiler and never run. Java has three kinds: `//` (one line), `/* */` (a block), and `/** */` (Javadoc, used to create API documentation).
- **Preconditions** (1.8.A.2): what must be true just before a method or code segment runs. The code is not expected to check them.
- **Postconditions** (1.8.A.3): what is always true afterward, described as the value returned or the current values of an object's attributes.

The CED's suggested skill is 4.B, describing the initial conditions that must be met for a code segment to work as intended, so give preconditions real time even though comments come first.

Stay inside the topic boundary:

- Students do not write methods until Unit 3, and method headers are Topic 1.9. Javadoc comments for library methods appear as comment text only, with no header line. Programs show Javadoc comments above the class and above `main`. If students ask about `public` and `static`, they are explained in a later topic and can be read past for now.
- `Math.sqrt` and `Math.pow` appear as in Topic 1.7; say what each does where it is used. Calls through a class name are taught fully in Topic 1.10.
- The `BankAccount` and `Playlist` classes are invented, to show postconditions about attributes. Students only read their documentation.
- A label and a value 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.

- **Direct instruction, comments:** `ScoreReport` with `ScoreReportCommented`, `StaleComment`, `SmoothieOrder`, `CommentedOut`, `CommentSwallowsCode` with `CommentSwallowsCodeFixed`, `GardenOrder`, `GardenSkipped`, `CommentMix`, `UnclosedComment` with `UnclosedCommentFixed`, and `TipCalculator`.
- **Direct instruction, conditions:** `RoundTemperature`, `RoundNegative`, `AveragePoints`, `AverageNoGames`, `SqrtCall`, `SplitMinutes`, and `SplitMinutesSmall`.
- **Guided practice:** `PizzaShare`, and `PizzaShareZero` for step 3 (do not show it until students have predicted).
- **Printed only (exit ticket):** `ExitStickers`, and `ExitPages` with `ExitPagesZero` for the key. The puzzle worksheet and its key are separate Word files; their code is printed on the page.

The Notes for People deck runs in order through the period; its speaker notes hold the answers to every Pause and Try slide. Before class, open `ScoreReport`, `PizzaShare`, 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.

Warm-up: what is this program for?

Show slide 2 and ask three questions students must answer in writing: "What is 45? Why times 3? Would you know a month from now?" Give two minutes. Every student can say it displays `135`, but guesses about why will disagree (minutes? dollars? points?). That is the point: the code says exactly **what** the computer does, but not **why**.

TEACHER NOTE

Keep the warm-up to five minutes. Do not define comments yet; let students notice that the notes did not change the output, and name the idea in the next section.

Direct instruction: comments, single-line comments, and commenting out

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

1. **Comment** (1.8.A.1, slide 5). A comment is text in a program written for people to read: the original programmer (you, two months from now) and other programmers. The compiler ignores comments, so they are never translated and never run. Slide 5 is the warm-up program with comments added: the 45 and the 3 now make sense, and the output is still 135. Run `ScoreReport` and `ScoreReportCommented` for a second pair: three notes added, and both display `Final quiz score: 20`.
2. **The code wins**. Run `StaleComment`. The comment says "Add 10 bonus points," but the code says `score += 5;`, so it displays 85, not 90. The compiler cannot warn you about a wrong comment, because it never reads comments. Update a comment in the same moment you change its code.
3. **Single-line comments** (Part 2, slides 7–8). `//` starts a comment that runs to the end of that line. It can fill a whole line or follow code on the same line (an end-of-line comment). Slide 8 shows ticket prices with end-of-line notes, including `4 times 2.5`, which only looks like arithmetic; it displays 10.0. `SmoothieOrder` is the same idea in the IDE (`3 times 4.5`, displays 13.5).
4. **Commenting out** (slide 6). Slide 6 traces `laps` with a `println` commented out: 5, then 7. Run `CommentedOut`. Line 2 is a valid `println` statement, but it sits after `//`, so it never runs, and the program displays only 6. Programmers switch lines off this way without deleting them. The rule is about **where** text is, not what it looks like.
5. **Predict: code after //**. Show `CommentSwallowsCode` and ask what happens. Everything after `//` on line 5 is comment, including `int pages = 320;`, so `pages` is never declared and line 6 gives `cannot find symbol`. `CommentSwallowsCodeFixed` puts the code first and displays 320. A `//` comment is always the last thing on its line.

MISCONCEPTION A comment that disagrees with the code tells you what the program does

Students read the English before the Java and trust it. In `StaleComment`, many predict 90. The compiler never checks comments, so an out-of-date note is never caught. Make "trace the code, not the comment" the rule for every prediction today, and point out that a wrong comment is a trap the programmer left for the next reader.

MISCONCEPTION Code after `//` still runs if it is valid Java

Students who type the note first and the code after it on the same line expect the code to run. The compiler stops reading Java at `//` and starts again on the next line. When the swallowed code is a declaration, the compiler reports `cannot find symbol` on a later line, which sends students hunting in the wrong place; when it is an update such as `+=`, the program compiles and gives a wrong answer. Ask: "Where on this line does `//` start?"

Direct instruction: block comments and Javadoc comments

1. **Block comments** (1.8.A.1, slide 9). A block comment starts with `/*` and ends with `*/`, and it can cover any number of lines. The end of a line does **not** end it; only `*/` does. Show `GardenOrder`: lines 1–5 are one block comment describing the program, and the `*` at the start of lines 2–4 is decoration. It displays 24.
2. **Switching off several lines.** Run `GardenSkipped`: lines 3 and 4 are inside `/*` and `*/`, so `rows` stays 6 and `Added two rows` never appears. It still displays 24. One pair of markers replaces four `//` markers.
3. **Predict: CommentMix.** Students trace `cups` line by line. Reveal: 8, 12, then `/*` opens after `cups += 4;` on line 3, `cups *= 2;` is inside the comment, `*/` closes it partway through line 4, and `cups -= 2;` runs. It displays 10. A block comment ends exactly at `*/`, even in the middle of a line.
4. **Forgetting `*/`** (slide 10). The left side of slide 10 is the `//` case from `CommentSwallowsCode`; the right side is a missing `*/`. Run `UnclosedComment`: the comment opened on line 3 never closes, so it swallows `main` and the closing braces. The compiler reports `unclosed comment` (usually followed by `reached end of file while parsing`). `UnclosedCommentFixed` adds `*/` and displays `Bring a calculator!` Habit: type `*/` as soon as you type `/*`, then fill in the note.
5. **Pause and try** (slide 11). Students mark each line "runs" or "comment" and write the output: 5. The block comment closes partway through line 3, so `score -= 3;` runs. A wrong answer of 8 means the whole of line 3 was treated as comment; 33 means the block ran; 0 means line 4 ran.
6. **Javadoc comments** (1.8.A.1, Part 3, slides 12–13). A Javadoc comment starts with `/**` (two stars) and ends with `*/`. It sits directly above a class or a method and describes it. The compiler ignores it like any comment, but a tool that comes with Java, **javadoc**, collects every `/** */` comment and builds the API documentation pages from them. The `String` and `Math` pages students read in Topic 1.7 were made this way; slide 13 shows a simplified `String` source, where the comment above the class becomes the page's description and the `//` note inside `length()` stays off the page. Show `TipCalculator`: one Javadoc comment above the class, one above `main`; it displays 9.0 as if they were not there.
7. **Tags** (slide 14). Show the simplified Javadoc comment for `Math.pow`. A **tag** is a word starting with `@` that the javadoc tool treats specially: `@param` describes a value the method must be given (`a`, the base; `b`, the exponent), and `@return` describes the value it gives back. That is how students knew in Topic 1.7 that the first value of `pow` is the base: `Math.pow(5.0, 2.0)` is 25.0.
8. **Pause and try** (slide 15). Which of the three comments reaches the documentation? Only C, the `/** */` comment.

MISCONCEPTION The end of the line ends a block comment

Students carry the `//` rule over to `/*`. In `CommentMix` and slide 11 they treat the closing line as code, or treat the rest of that line as comment. Have them put a finger on `/*` and slide it forward until they reach `*/`: everything the finger passes is comment, however many lines that is, and the code right after `*/` runs.

MISCONCEPTION `/*` and `/**` do the same job

Both compile, because the compiler ignores both, so students cannot see a difference until the documentation is built. The javadoc tool collects only comments that start with `/**`. A description written in `/* */` above a method will be missing from the documentation pages. Rule: `/** */` for what users of a class need to know; `//` or `/* */` for notes about how the code works inside.

Direct instruction: preconditions and postconditions

Start Part 4 (slide 16).

1. **Precondition** (1.8.A.2, slide 17). A precondition is a condition that must be true just before a method or code segment runs, so that it does what it is supposed to do. It is usually stated in a comment. The code is **not** expected to check it: making it true is the job of whoever uses the code. Connect to Topic 1.5: `(int) (x + 0.5)` rounds correctly only when `x` is not negative.
2. **Met, then broken** (slide 18). Slide 18 rounds 6.8 to 7, then -6.8 to -6, though the nearest whole number is -7. Run `RoundTemperature` (2.7 displays 3), then ask students to predict `RoundNegative` (-2.7). It displays -2, but the nearest whole number is -3. Nothing stopped or warned; the code ran every step and gave a wrong answer.
3. **Sometimes it crashes** (slide 19). Run `AveragePoints` (Computing average... then 14), then `AverageNoGames`: with games 0, the first line displays, then line 5 throws `ArithmeticException: / by zero` (Topic 1.1) and the average never appears. The precondition comment was the only warning. Slide 19 shows the same crash with `total / count` and `count 0`: it compiles, because the compiler never reads the precondition.
4. **Library methods have them too**. Write a simplified documentation comment for `Math.sqrt` on the board: "Precondition: `a` is not negative." Run `SqrtCall`: 49.0 is not negative, so it gives back 7.0. Before you call a method, check the values you are about to give it.
5. **Postcondition** (1.8.A.3, slide 17). A postcondition is a condition that is always true after the method or code segment runs, as long as the precondition was met. It is the code's promise. It describes the outcome in one of two ways: the value **returned** ("returns the square root of `a`"), or the current value of an object's **attributes** (Topic 1.7). For the second, use the invented `BankAccount` class: "Postcondition: the balance has gone up by amount."
6. **Check a postcondition**. Show `SplitMinutes`. Students trace it (135 gives 2 and 15) and check the promise: 15 is between 0 and 59. Then change `totalMinutes` to 59 (`SplitMinutesSmall`: 0 h 59 min). The postcondition holds for every value that meets the precondition, not just typical ones. Then give pairs slide 20 (Check the promise): 437 cents gives `dollars` 4 and `left` 37, so the postcondition holds; with -437 the values are -4 and -37, and the code is not broken, because the precondition was not met.
7. **Tell them apart**. Two questions sort any condition: **When** is it true, before or after? **Who** makes it true, the caller or the code?

MISCONCEPTION If the input were bad, the code would tell me

Students assume code checks its own inputs and complains. A precondition means the opposite: the code assumes the condition is true and goes ahead. `RoundNegative` printed `-2` with no warning at all. Ask students, before every run today: "What is the precondition, and do these values meet it?"

MISCONCEPTION Preconditions and postconditions are the same kind of rule

Students label a requirement such as "amount must be greater than 0" as a postcondition because it appears in the same comment. Use the two questions: a requirement the caller must meet beforehand is a precondition; an outcome the code guarantees afterward is a postcondition. A statement about a variable that does not have its value until the code runs cannot be a precondition.

Guided practice: label, run, and break `PizzaShare`

Pairs open `PizzaShare` in the IDE (or read it on the projector).

1. **Label every comment.** Pairs write the line numbers and kind of each comment: lines 1–3 and 6–8 are Javadoc (above the class and above `main`); lines 11–15 are a block comment stating the precondition and postcondition; lines 16 and 19 have end-of-line comments; line 24 is a `println` that has been commented out.
2. **Predict, run, and check the promise.** Pairs write the exact output, then run it:
Each friend gets 3 and Slices left over: 1 (16 / 5 is 3; 16 % 5 is 1). The precondition held (5 is greater than 0), so the postcondition must be true, and it is: both values were printed, and 1 is less than 5.
3. **Break the precondition.** Pairs change line 17 to `int friends = 0`; and predict before running. Expected (`PizzaShareZero`): nothing is displayed, then `ArithmeticException: / by zero`. The division on line 19 comes before any `print`, so the program stops before its first line of output. Nothing in the program checked; the comment on line 12 was the only warning.

Debrief: ask which comment would appear in the API documentation (the two Javadoc comments), and whose job it was to keep `friends` above 0 (the programmer who sets `friends`, not the division).

TEACHER NOTE

If computers are not available, make the change on the projector after pairs predict. Pairs that finish early can write a postcondition for lines 11–20 that mentions `perFriend` by name, then trade with another pair and check whether it is a promise about the result.

Independent practice

The Notes for People puzzle worksheet (nine questions, about 30 minutes) is homework. Part A (questions 1–4) traces code with `//` and `/* */` comments, decides which snippets compile, fills in a precondition-and-postcondition face-off, and reads a Javadoc comment with `@param` and `@return`; Part B (questions 5–7) orders code tiles under a precondition and postcondition, places comment markers from a pool, and names the ideas; Part C (questions 8–9) checks a claim about preconditions and fixes an unclosed block comment, with review of integer division and `%` (Topic 1.3) and compound assignment (Topic 1.6).

Closure and exit ticket

Closure takes about three minutes. Show slide 21 (Words to know) and point to the three comment cards: ask the class where each kind starts, where it ends, and who reads it. Then show slide 22 (Before and after): pairs finish the sentence starter, "A precondition must be true ... and ... makes it true. A postcondition is guaranteed ..."; cold-call two pairs. Hand out the puzzle worksheet. Give the exit ticket (about four minutes). Question 1 checks reading code that contains `//` and `/* */` comments (1.8.A.1), and question 2 stating a precondition and telling it apart from a postcondition (1.8.A.2, 1.8.A.3). Re-teach the weaker one at the start of Topic 1.9.