

1.1 Introduction to Algorithms, Programming, and Compilers

Lesson Plan

Pacing: 2 class periods of 45 minutes

Learning objectives: 1.1.A, 1.1.B, 1.1.C

Essential knowledge: 1.1.A.1, 1.1.A.2, 1.1.B.1, 1.1.B.2, 1.1.C.1, 1.1.C.2, 1.1.C.3, 1.1.C.4

Materials

- Projector and the Plans, Programs, and Bugs slide deck
- Whiteboard and markers
- Blank paper for the Day 1 warm-up
- IDE on every computer (Day 2)
- Printed Plans, Programs, and Bugs puzzle worksheet (answer key for you) and exit ticket

Agenda

Minutes	Activity
8	Day 1 warm-up — exact-instructions challenge
12	Day 1 direct instruction — algorithms, precision, words and flowcharts (slides 3–6)
14	Day 1 guided practice — sequencing and the swap test (slides 7–9)
8	Day 1 independent practice — puzzle worksheet questions 1–2
3	Day 1 closure — why precision and order matter
5	Day 2 warm-up — predict the output of a Java program
12	Day 2 direct instruction — IDE, compile then run, syntax errors (slides 10–15)
8	Day 2 direct instruction — logic errors, testing, run-time errors (slides 16–18)
13	Day 2 guided practice — break a program on purpose in the IDE
7	Day 2 closure — name that error (slides 19–21), then the exit ticket

Overview

This is the first lesson of the course. Day 1 is unplugged: students write, draw, and reorder everyday algorithms (1.1.A). Day 2 moves to Java: students see where code is written, what the compiler does, and the three kinds of errors (1.1.B, 1.1.C). Students do not write Java of their own yet. Every program they see contains only `System.out.println` statements, plus a division by zero on Day 2 to cause a run-time error.

Keep one idea running through both days: a computer is a **literal follower**. It does exactly what is written, in the order written, and fills in nothing. The study guide uses the same phrase, so

students who read it will recognize it. It is the guide's name for the idea, not College Board vocabulary.

TEACHER NOTE

The slides are the Plans, Programs, and Bugs deck (21 slides, with speaker notes); slide numbers below refer to it. Day 1 uses slides 1–9 and Day 2 uses slides 10–21. The Plans, Programs, and Bugs puzzle worksheet (10 questions, about 30 minutes) covers both days: questions 1–2 need only Day 1 material, and questions 3–10 need Day 2.

Before Day 2, open the IDE on the projector and create `GreetingReordered.java`, `Greeting.java`, `MissingSemicolon.java`, and `Snack.java` from the topic's examples, so each demonstration is one click. Check that students can create and run a file in the IDE; if accounts or installs are not ready, run the Day 2 guided practice as a whole-class demo instead.

Day 1 warm-up: exact-instructions challenge

Ask each student to write, on blank paper, the steps for drawing a smiley face on the board. Give them three minutes. Collect three or four sets and follow each one at the board **exactly** as written, with a straight face. "Draw a circle" gets a tiny circle in a corner; "draw two eyes" gets two eyes anywhere, maybe outside the circle; a missing "pick up the marker" means you cannot start.

Then ask: "What went wrong, and whose fault was it: mine or the instructions'?" Steer students to two ideas you will name in a moment: the steps were not **precise**, and sometimes they were in the wrong **order**.

TEACHER NOTE

Pick at least one set that fails because of vagueness and one that fails because of order or a missing step. Keep the tone playful; the goal is for students to feel what a literal follower does, not to embarrass anyone. Stop at eight minutes even if more students want a turn. If the room or time does not suit the board activity, slide 2 (a robot making a bowl of cereal) is a quicker partner version of the same warm-up.

Day 1 direct instruction: algorithms, precision, words and flowcharts

1. **Goals** (slide 3). Read the five goals aloud; the first three are today's, the last two are Day 2's.
2. **Define algorithm** (1.1.A.1, slides 4–5): a step-by-step process that completes a task or solves a problem. Ask for everyday examples: a recipe, directions to a house, getting ready for school, logging in to a website.
3. **Precision** (slide 5). Show only the vague smoothie steps first ("Add some fruit. Blend it. Serve."). Ask, "What would a literal follower ask about each step?" Collect questions such as "How much?" and "For how long?", then reveal the questions and precise rewrites in the table.

4. **Two representations** (1.1.A.1, slide 6). Show the login algorithm as a numbered list and as a flowchart side by side. Match them box by box. Point out that the ovals mark Start and Stop and are not steps, and that in a flowchart **the arrows are the order**.
5. **Same algorithm or not?** Ask: "If I swapped the username and password boxes in the flowchart, would it still be the same algorithm as the list?" No: what defines an algorithm is its steps and their order, not whether it is written in sentences or boxes.

MISCONCEPTION The computer will know what I mean

Students leave out steps that feel obvious ("open the toothpaste", "turn on the water") because a human reader would fill them in. A computer fills in nothing. Address it by having students read their own algorithm aloud as a literal follower, acting out only what is written. The warm-up gives you a shared example to point back to.

MISCONCEPTION A diagram and a list are always different algorithms

Students think that redrawing a list as a flowchart creates a new algorithm. Have them compare the two step by step and check the order. If the steps and order match, it is the same algorithm in a different representation. Then show the opposite case: same boxes, different arrows, different algorithm.

Day 1 guided practice: sequencing

Define **sequencing** (1.1.A.2, slides 7–8): the order in which steps are completed. Steps are completed **one at a time**, and each step happens in the situation the earlier steps left behind.

1. Show the cake algorithm on slide 8 in its original order (bake, cool, ice). Ask pairs to predict what happens if steps 2 and 3 swap before you reveal the right column: the icing melts on the hot cake. Same steps, worse result.
2. **The swap test** (slide 9). Ask pairs which two steps of the tea algorithm (cup, tea bag, bag in cup, hot water) can swap safely and which swap breaks it. Getting the cup and getting the tea bag can swap; putting the bag in the cup cannot come before getting the cup. Draw out the rule: **order matters when a later step depends on what an earlier step did**.
3. Have each pair write a four-step algorithm for a school routine (for example, checking out a library book) as a numbered list, then draw it as a flowchart. Call on two pairs to share and have the class check the order by asking of each step, "Is everything this step needs already done?"

MISCONCEPTION If all the right steps are there, the algorithm is correct

Students check only that nothing is missing. A list of the right steps in the wrong order is still a wrong algorithm. Give them the dependency question to ask of every step: "Was everything this step needs done by an earlier step?"

Day 1 independent practice

Hand out the puzzle worksheet. Students work Part A questions 1–2: question 1 sorts steps into precise and vague, and question 2 matches a plant-watering flowchart to the written list that is the same algorithm. Circulate and look especially at the second half of question 2: students should say that choice A has the right steps in the wrong order, so it is a different algorithm.

TEACHER NOTE

Questions 1–2 need only Day 1 material. Students keep the worksheet; the rest of it (questions 3–10) is Day 2 homework. Any of questions 1–2 not finished in class are homework too.

Day 1 closure

Ask each student to finish this sentence on a sticky note or in their notes: "A computer needs precise, ordered steps because ..." Look for two ideas: it is a literal follower (precision), and it does steps one at a time in order (sequencing).

Day 2 warm-up: predict the output

Show `GreetingReordered.java` without running it. Ask students to write what it displays. Most will guess correctly from Day 1's idea of sequencing; tell them that is exactly the point: Java statements run one at a time, top to bottom, in the order written. Run it to confirm.

TEACHER NOTE

Do not explain `public class` or `main` in detail. Say that every program in the course has this frame and that the lines inside `main` are the steps. The words `public` and `static` are explained in a later topic; tell students to read past them for now.

Day 2 direct instruction: IDE, compile then run, syntax errors

1. **From algorithm to program** (slides 10–11). A **programming language** (Java) has strict rules so every instruction means exactly one thing. An algorithm written in it is a **program**; the text is **source code**. Each line inside `main` is a **statement**; `Welcome.java` on slide 11 has two, and they run one at a time, in order.
2. **Where code is written** (1.1.B.1, slide 11). Code is plain text, so any text editor can hold it. An **IDE** puts the tools to write, compile, and run code in one application. Switch to the IDE and show its editor, its error highlighting, and its run button.
3. **Compile, then run** (1.1.B.2, slide 12). The **compiler** checks the code for some errors, then translates it so the computer can run it. Running carries out the statements one at a time. Errors the compiler finds must be fixed before anything runs.
4. **Live demo** (slide 15 afterward). Run `MissingSemicolon.java`. Before you click run, ask: "Line 5 is missing a semicolon. Line 6 is fine. What will we see?" Many students predict "Good morning!" and then an error. Run it: only the compiler message appears. Slide 15 shows the same result with three class periods: one missing semicolon on the last line, and no line prints.

5. **Syntax errors** (1.1.C.1, slide 14): code that breaks the language's rules; the compiler detects them. Use `Poster.java` on slide 14 to read a compiler message in three parts: file and line, what is wrong, and where the caret points. List the common syntax errors on the board (missing semicolon, missing closing quote, wrong capitalization, misspelled word) and stress that Java is **case-sensitive**.

MISCONCEPTION The first line is fine, so it will print before the error

This is the most common misunderstanding about compiling. Students picture the computer running line by line until it hits the bad line. Compiling happens before running and covers the whole program, so one compiler error means nothing runs. The `MissingSemicolon.java` demo and slide 15 are the fix: let students commit to a prediction before you run it.

MISCONCEPTION The program reads like English, so the computer understands English

Code contains English words such as `print`, so students assume the computer understands sentences. Show that `System.out.println` works and `system.out.println` does not. The computer accepts only code that follows Java's rules exactly.

MISCONCEPTION Fixing only the first error

When the compiler lists two errors, students fix one and expect the program to run. It will not run until every compiler error is fixed. Teach them to work down the list from the top and recompile after each fix, because one mistake can cause several messages.

Day 2 direct instruction: logic errors, testing, run-time errors

1. **Logic errors** (1.1.C.2, slide 16). Show the scrambled-eggs program. It compiles and runs to the end, but it cooks the eggs before whisking them. Ask: "Did the compiler complain? Is the program right?" A logic error makes the program behave incorrectly or unexpectedly; the compiler cannot see it.
2. **Testing** (slide 16). Logic errors are found by testing: decide the **expected outcome** first, run the program with specific data, and compare. Point to the expected and actual columns on the slide. Then give a doubling example on the board: an algorithm that "adds 2" passes a test with 2 ($2 + 2 = 2 \times 2$) but fails with 5 (expected 10, actual 7). One lucky test can hide a logic error.
3. **Run-time errors and exceptions** (1.1.C.3, 1.1.C.4, slide 17). Show `Snack.java` in the IDE and say that `/` means divide, as on a calculator (arithmetic is taught in Topic 1.3). Ask students to predict the output, then run it. The first line appears, then `ArithmeticException: / by zero`, and the last line never runs. Slide 17 shows the same pattern in a movie-night program. A **run-time error** happens while the program runs and usually makes it **terminate abnormally**. An **exception** is a run-time error the compiler did not catch; it interrupts the normal flow of the program.
4. **Compare all three** using the table on slide 18: does it compile, does it start running, how does it end, and who or what finds it.

MISCONCEPTION It compiled, so it works

Students stop as soon as the compiler is happy. No compiler error only means there are no errors the compiler can detect. Make "What is the expected outcome?" a habit: ask it before every run for the rest of the unit.

MISCONCEPTION An exception is just a syntax error that shows up later

They are different. Code that throws an exception follows every rule, so it compiles and starts running. Give students two questions: "Did it compile?" and "Does the message say `error:` (the compiler) or start with `Exception` (the running program)?" Warn them not to rely on whether output appeared first; an exception on the first statement shows no output at all.

Day 2 guided practice: break a program on purpose

Students type `Greeting.java` into the IDE and run it. Then, one at a time, they make each change below, predict what will happen, run it, record the result, and undo the change:

1. Delete the semicolon at the end of line 5.
2. Change `System` to `system` on line 6.
3. Swap lines 5 and 6.
4. Add `System.out.println(10 / 0);` between lines 5 and 6.

For each, students record: Did it compile? What was displayed? Which kind of error is it (syntax, logic, or run-time)? Debrief with the class: changes 1 and 2 are syntax errors (nothing displayed), change 3 is a logic error if the expected outcome was "Good morning!" first, and change 4 is a run-time error (one line, then the exception).

TEACHER NOTE

Some IDEs flag syntax errors as you type and may offer to fix them. Ask students to ignore the suggestions and run anyway, so they see the compiler's own message. This follows the CED's suggestion to verify errors with an IDE that does not autocorrect.

Day 2 independent practice

Puzzle worksheet questions 3–10 (be the compiler, code tiles, be the JVM, the match-up, the face-off, who am I, and two catch-the-mistake questions) are homework, about 25 minutes. Students who finish the IDE activity early can start them in class. Question 4 ends with a Think First prediction that question 6 settles, so ask students to work the questions in order.

Day 2 closure and exit ticket

Spend two minutes on slide 19 (Name that error): the class calls out syntax, run-time, or logic for each case and says how they can tell. Slide 20 lists the vocabulary, and slide 21's sentence starter

can be a quick pair-share if time allows. Then give the exit ticket (about five minutes). Score each question separately: question 1 checks algorithms and sequencing, question 2 the compile-then-run process, and question 3 the error types. Re-teach the weakest one at the start of Topic 1.2.