

1.7 Application Program Interface (API) and Libraries

Lesson Plan

Pacing: 2 class periods of 45 minutes

Learning objectives: 1.7.A

Essential knowledge: 1.7.A.1, 1.7.A.2

Materials

- Projector
- Whiteboard and markers
- IDE on every computer
- Check Out the Library slide deck
- Printed Check Out the Library puzzle worksheet and exit ticket

Agenda

Minutes	Activity
6	Day 1 warm-up — a square garden with only + - * / %, then who wrote this code? (slide 2)
14	Day 1 direct instruction — classes, libraries, and the library classes students already use (slides 3–6)
12	Day 1 direct instruction — packages, full names, and import (slides 7–9)
9	Day 1 guided practice — break and fix the import line in the IDE
4	Day 1 closure — how many import lines? (slide 10)
5	Day 2 warm-up — packages and import, from memory
10	Day 2 direct instruction — API documentation and finding the method you need (slides 11–15)
9	Day 2 direct instruction — attributes and behaviors (slides 16–18)
7	Day 2 direct instruction — classes are types that make objects (slides 19–20)
7	Day 2 guided practice — take apart NameLength
2	Day 2 closure — look it up, don't guess (slides 21–22)
5	Day 2 exit ticket

Overview

Topic 1.7 has one learning objective (1.7.A): identify the attributes and behaviors of a class found in the libraries contained in an API. The CED's suggested skill is 4.A, describing the behavior of a code segment, so students spend most of both days reading code and documentation and saying what each part is and does, not writing new programs.

Students have been using library classes since Topic 1.3 without knowing it. This topic names what they have been doing:

- **Day 1 (1.7.A.1):** libraries are collections of classes, and classes are grouped into packages. `String`, `Math`, `System`, `Integer`, and `Scanner` all come from Java's library; only `Scanner` needs an import line.
- **Day 2 (1.7.A.1, 1.7.A.2):** the API specification tells programmers how to use a class without reading its code; a class's attributes are data stored in variables and its behaviors are defined by methods; a class defines a reference type and is used to create objects.

Stay inside the topic boundary:

- The only methods students meet are the ones the study guide uses: `Math.sqrt`, `Math.abs`, `Math.pow`, `length()` on a `String`, and the `Scanner` reads from Topic 1.4. Explain each one when it appears. The full `Math` list waits for Topic 1.11 and the `String` list for Topic 1.15.
- Calls such as `Math.sqrt(144.0)` and `word.length()` may be shown and traced. Say once that the rules for calls through a class name are taught in Topic 1.10 and calls on an object in Topic 1.14. Method headers and what goes in the parentheses are Topic 1.9.
- Documentation lines show the word `static` and the program frame shows `public` and `static`. Tell students they are explained in a later topic and can be read past for now.
- No comments in code (Topic 1.8). A label and a value still take a `print` followed by a `println`; joining text and numbers is Topic 1.15.
- `new` appears only as the `Scanner` pattern from Topic 1.4. Say that it creates an object and that Topic 1.13 explains it; do not teach constructors.

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 direct instruction:** `GardenPlan`, `ScoreBoard`, `LowercaseClass`, `TicketCount`, `FullName`, `MissingImport`, and `WrongPackage` with `WrongPackageFixed` for the answer.
- **Day 1 guided practice:** students type `TicketCount` and change it; `MissingImport`, `WrongPackage`, and `FullName` show the results they should get. `ImportLocalDate` shows the `java.time.LocalDate` import from the closure question compiling.
- **Day 2 direct instruction:** `TemperatureSwing`, `PowOrder`, `PowTry`, `GuessedName`, `IntegerLimits`, `CityLengths`, `AttributeParens`, `TripInfo`, and `TextInInt`.
- **Day 2 guided practice:** `NameLength`. `WordGoal` is the study guide's version of the same program, for students who finish early.
- **Printed only:** none. The exit ticket has no code, and the puzzle worksheet and its key are separate Word files with their code printed on the page.

The Check Out the Library deck covers both days: slides 1–10 on Day 1 and slides 11–22 on Day 2. Its speaker notes hold the answers to every Pause and Try slide. Open `GardenPlan`, `TicketCount`, `TripInfo`, and `NameLength` in the IDE next to the deck before each day, and keep a scratch class open for the snippets. Compiler messages are quoted from a recent JDK; the wording can vary slightly between Java versions.

Day 1 warm-up: a square garden

Put this on the board: "A square garden has an area of 144 square feet. Write Java that stores the length of one side. You may use only `+`, `-`, `*`, `/`, and `%`." Give two minutes.

Most students write `double side = 12.0;` or `144 / 12`. Ask: "How did you know 12? What if the area were 150?" There is no operator that does it. Someone would have to write an algorithm that guesses, checks, and improves the guess over and over. Then ask: "Would you trust your version in every program you ever write?"

Show `GardenPlan` and run it: Side length in feet: 12.0. Line 6, `Math.sqrt(144.0)`, hands 144.0 to square-root code that someone else wrote and tested. Then show slide 2 (Who wrote this code?): pairs pick out the capitalized names in `Bakery` and say who wrote each. Only `Bakery` is the programmer's; `Math.sqrt(225.0)` gives back 15.0 from code nobody in the room wrote. Today is about where that code comes from.

TEACHER NOTE

Keep the warm-up to six minutes. Do not explain the call syntax yet; read `Math.sqrt(144.0)` as "use the `Math` class's square-root code on 144.0" and say that Topic 1.10 explains calls like this.

Day 1 direct instruction: classes and libraries

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

1. **Class.** Point at `public class GardenPlan`. Every program students have written is wrapped in a **class**: a named unit of Java code that groups related data with the actions that work on that data. Read past `public` for now.
2. **Library** (1.7.A.1, slide 5). A **library** is a collection of classes written to be used by other programs. Java comes with a large standard library. Ask the class to list capitalized names they have typed this year that they did not write. Collect `System` (Topic 1.3), `String` (Topic 1.2), `Scanner` (Topic 1.4), and `Integer` (Topic 1.5), then add `Math`, and reveal the table on slide 5. None of them is part of the Java language the way `int` and `double` are; they are classes that come with every Java installation.
3. **Spot the library classes** (slide 6). Walk through `Bakery` line by line with the slide's table: `Bakery` is the programmer's class; `String`, `Math`, and `System` are library classes. `GardenPlan` has the same shape. The program has six statements and relies on three classes it never defined. Then show `ScoreBoard`: different calculation (`Math.abs(-12)` gives the distance of -12 from zero, 12), same lesson. One class is yours; every other capitalized class name came from the library.
4. **Why use them?** Ask for two reasons a programmer would use `Math.sqrt` instead of writing their own. Expected: it saves time, it is already tested, and the programmer can focus on their own problem.
5. **Spelling counts.** Run `LowercaseClass: math.sqrt(144.0)` gives `cannot find symbol with symbol: variable math`. There is no class named `math`, so Java guesses it is a variable, and there is no such variable either.

MISCONCEPTION Library classes are part of the Java language

Students treat `String` and `Math` as keywords, like `int`. Point to the capital letter: primitive type names and keywords are lowercase; class names start with a capital. `String` is a class someone wrote and shipped with Java, which is why it has documentation like any other class. This sets up Day 2, where a class turns out to be a type.

MISCONCEPTION math and Math are the same

Students who type fast write `math`, `string`, or `scanner`. Java is case-sensitive (Topic 1.1), and a library class has exactly one spelling. Teach students to read the `symbol:` line: variable `math` or class `String` means the name is spelled differently from the class.

Day 1 direct instruction: packages, full names, and import

Start Part 2 (slide 7).

1. **Package** (1.7.A.1, slide 8). Thousands of classes in one pile would be hard to search, and two classes could not share a short name. A **package** is a named group of related classes, like a labeled shelf. Package names are lowercase words joined by dots. Two matter in this course: `java.lang` (`String`, `Math`, `System`, `Integer`) and `java.util` (`Scanner`).
2. **Full name**. A class's **full name** is its package, a dot, and its class name: `java.util.Scanner`, `java.lang.String`. Everything before the last dot is the package.
3. **Import**. Show `TicketCount`. Line 1, `import java.util.Scanner;`, is an **import statement**: "in this file, `Scanner` means the class `Scanner` from `java.util`." It goes above the class, ends with a semicolon, and runs nothing; it is information for the compiler. Run it: `Tickets: 19`.
4. **The import is only a shortcut**. Show `FullName`: no import line, and `java.util.Scanner` written out twice on line 5. Same output, `Tickets: 19`. Programmers rarely write it this way, but it shows what the import line does. Slide 8 shows the two versions side by side.
5. **Available by default**. Ask: "Why has `String` never needed an import?" Classes in `java.lang` are **available by default**: every program can use their short names. Every other package must be imported.
6. **Predict** (slide 9). Show `MissingImport` and ask what happens. Run it: `cannot find symbol with symbol: class Scanner`, reported for each `Scanner` on line 5. Nothing runs.
7. **Quick check**. Show `WrongPackage` and ask what is wrong with line 1. `Scanner` is in `java.util`, not `java.lang`; the compiler reports `cannot find symbol for class Scanner in package java.lang`, and then every use of `Scanner` fails. `WrongPackageFixed` displays 15.

MISCONCEPTION Every library class is available without an import

`String`, `Math`, and `System` never needed an import, so students assume `Scanner` will not either. Only `java.lang` is available by default. Give students one question to ask of every class they use: "Which package is it in?" If the answer is anything but `java.lang`, the file needs an import line.

MISCONCEPTION The import line runs code or copies it into the program

Students picture `import` loading or running the `Scanner` code before `main` starts. It does neither: it only tells the compiler which class a short name means. `FullName` is the proof: with no import line at all, the program compiles and runs the same, because the full name gives the compiler the same information.

Day 1 guided practice: break and fix the import line

Students type `TicketCount` into the IDE and run it (`Tickets: 19`). Then, one change at a time, they predict, compile, record the result, and undo the change:

1. Delete line 1. Expected: `cannot find symbol, symbol: class Scanner`, for each `Scanner` on the line that creates the `Scanner` (as in `MissingImport`). Nothing runs.
2. Put line 1 back as `import java.lang.Scanner;`. Expected: `cannot find symbol for class Scanner in package java.lang`, then errors where `Scanner` is used (as in `WrongPackage`).
3. Put back `import java.util.Scanner;`, then delete it again and write `java.util.Scanner` in both places on the line that creates the `Scanner`. Expected: it compiles and displays `Tickets: 19` (as in `FullName`).

Debrief: ask which change showed that the import line is a shortcut (3), and which `symbol:` line pointed at a missing import (1).

TEACHER NOTE

If computers are not available, make each change on the projector after students predict. Students who finish early can start puzzle worksheet question 1.

Day 1 independent practice

Hand out the Check Out the Library puzzle worksheet. Question 1 (who wrote each class in `TicketBooth`, its package, and whether it needs an import) and the Think First box are homework. Students keep the worksheet for Day 2.

Day 1 closure

Show slide 10 (How many import lines?). Students answer both questions on scrap paper: a program that uses `String`, `Math`, `Scanner`, and `Integer` needs one import line, `import java.util.Scanner;;` and a `LocalDate` class in the package `java.time` needs `import java.time.LocalDate;`. Watch for `import java.time;` (only the package) and the parts in the wrong order. Ask a volunteer to write the `LocalDate` line on the board. Run `ImportLocalDate` to show it compiling, but do not display its code: it contains `null`, which Topic 1.13 teaches.

Day 2 warm-up: packages and import, from memory

Students answer three questions on scrap paper, without notes:

1. What is the full name of `Scanner`? (`java.util.Scanner`)
2. Which of `String`, `Math`, `Scanner`, and `Integer` needs an import line? (Only `Scanner`.)
3. A program uses `Scanner` with no import line. What does the compiler report?
(`cannot find symbol, with symbol: class Scanner`.)

Then ask: "Yesterday you used `Math.sqrt` and `Math.abs`. How would you find out what else `Math` can do, without reading its code?" Collect guesses and move on to Part 3 (slide 11).

Day 2 direct instruction: API documentation

1. **Interface.** A microwave's buttons are its interface: you use it without knowing its wiring. An **application programming interface (API)** is the set of classes, and the parts of those classes, that a library offers for other programs to use. The CED says "application program interface"; both mean the same thing.
2. **Documentation** (1.7.A.1). The **API specification**, usually called the **documentation**, describes the API: which classes exist, which package each one is in, what each is for, and how to use each part. Java's documentation is free online, with one page per class. It is how programmers learn what a class can do without reading its code.
3. **Read a page** (slide 12). Show the simplified `Math` page and take it apart from the top: the package (`java.lang`, so no import), the class name, and the **Method Summary**. A **method** is a named piece of code that does one job. Each row gives what kind of value comes back, the name, what goes in the parentheses, and what it does. Read past `static`; Topic 1.10 explains it, and Topic 1.9 teaches these method headers in full.
4. **Find the method** (slide 13, left). A weather program needs the size of a temperature change as a positive number. `41 - 63` is `-22`. Which row does the job? `abs`, "the absolute value of a," so the program displays `22`. Run `TemperatureSwing`, the same program with `58` and `71` degrees: `13`.
5. **Order is in the documentation** (slide 13, right). Show `PowOrder` and ask for predictions before running. The `pow` row says "a raised to the power b", so `Math.pow(2.0, 10.0)` is `1024.0` and `Math.pow(10.0, 2.0)` is `100.0`. Only the documentation tells you which value is the base. `PowTry` is a second example if students need one (`9.0`, then `8.0`).
6. **Do not guess** (slide 14). Slide 14's `Math.cube(3.0)` gives `cannot find symbol with symbol: method cube(double)`; the fix is `Math.pow(3.0, 3.0)`. Run `GuessedName` for a second case: `Math.squareRoot(144.0)` fails the same way, and the fix is to look up the real name, `sqrt`.
7. **Pause and try** (slide 15). Students predict `6.0`, `9.0`, and `4`. Students who write `6` and `9` ignored the type the documentation says comes back.

TEACHER NOTE

If you want to show the real `Math` page online, point only at the package line, the description, and the heading "Method Summary." Do not walk through its rows: it lists many more methods, several with the same name, which Topics 1.9–1.11 explain. Slide 12's page is simplified and written in the guide's own words; say so.

MISCONCEPTION A method name can be guessed

`squareRoot` or `power` sounds reasonable, so students type it and expect Java to understand. Java knows only the names the class actually has. Make "check the documentation" the first step whenever students use a library method today, and point out that the `symbol:` line names the method the compiler looked for.

MISCONCEPTION The documentation shows how the method works inside

Students expect documentation to contain the method's code, and some worry that using code they cannot see is cheating or unsafe. Documentation describes what each part does and how to use it, not its code; that is the point of an API. The microwave analogy helps: you trust the button labeled "Popcorn" without opening the case.

Day 2 direct instruction: attributes and behaviors

Start Part 4 (slide 16).

1. **Two kinds of entries** (1.7.A.2, slide 17). Describe a phone: some things are facts (its battery level, its phone number) and some are actions (make a call, take a photo). An **attribute** is data related to a class, stored in variables. A **behavior** is what the class's objects can do, or what can be done with them, defined by methods. Test: a noun phrase ("the largest value") is an attribute; a verb phrase ("give back the square root") is a behavior.
2. **Field Summary.** Sketch the `Integer` entries on the board: a Field Summary with `MAX_VALUE` and `MIN_VALUE`, and a Method Summary with `parseInt`. `MAX_VALUE` and `MIN_VALUE` are in the **Field Summary**, where documentation lists the attributes programs may use; a **field** is Java's name for a variable that belongs to a class. `parseInt` is in the Method Summary, so it is a behavior. Do not use `parseInt` in code; it is here only to sort. Run `IntegerLimits`: an attribute is read by name with no parentheses (`2147483647`, `-2147483648`), the same constants students used in Topic 1.5.
3. **A behavior of a String.** The `String` documentation lists `length()`, "the number of characters in this string." Slide 17 shows it on two objects: `"Ada"` gives 3 and `"Grace Hopper"` gives 12. Run `CityLengths`: 6 and 11. The space in `"Kansas City"` counts. A behavior is written with parentheses, even when nothing goes inside, and the result depends on which object does it. Say once that calls such as `home.length()` are taught fully in Topic 1.14.
4. **Not every attribute is listed.** A `String` must store its characters somewhere, but that variable is hidden inside the class. The documentation describes it in words ("holds a sequence of characters") and programs use it through behaviors such as `length()`.
5. **Watch out** (slide 18). Run `AttributeParens`: `Integer.MAX_VALUE()` gives cannot find symbol with symbol: method `MAX_VALUE()`. The parentheses told Java to look for a method. Slide 18 also shows the mirror-image slip, `city.length` with no parentheses, which gives symbol: variable `length`. Field Summary entries are written without parentheses; Method Summary entries with them.

MISCONCEPTION Attributes and behaviors swapped

Students remember both words but attach them to the wrong things, or think an attribute is "something you do to an attribute." Keep the fact-or-action test on the board all period, and connect it to the two summaries: Field Summary means data in variables, so no parentheses; Method Summary means actions, so parentheses.

MISCONCEPTION Parentheses after every name

After a day of `Math.sqrt(...)` and `length()`, students add parentheses to `Integer.MAX_VALUE`, or drop them from `length`. Both give cannot find symbol; the symbol: line says method `MAX_VALUE()` or variable `length`, which tells you which kind of entry Java looked for. Puzzle worksheet question 3 has the first slip, and a decoy tile in question 4 has the second.

Day 2 direct instruction: classes are types that make objects

1. **A class is a type** (1.7.A.1, slide 19). In Topic 1.2, students sorted types into primitive and reference. Now the reason `String` is capitalized: it is a class, and each class defines its own **reference type**. `String city` can hold text; `Scanner reader` can hold a scanner; neither can hold the other. What a reference variable actually stores is Topic 1.12.
2. **A class is a blueprint**. A blueprint describes every house built from it, but you live in a house, not a blueprint. One particular thing made from a class is an **object**, also called an **instance**. "Denver" and "Kansas City" are two `String` objects, each with its own characters. The analogy has a limit: an object has only the attributes and behaviors its class defines.
3. **Walk through TripInfo**. Slide 19's table does the same for three declarations. For each declaration, ask: what type, and is an object created? `miles` is `int` (no object); `city` is `String`, and the literal "Denver" is a `String` object; `reader` is `Scanner`, and `new Scanner("3")` creates a `Scanner` object (`new` creates an object; Topic 1.13 explains it); `days` is `int`. Run it.
4. **Math is different**. The program uses `Math` without a `Math` variable or a `Math` object: its behaviors are used through the class name. Topic 1.10 explains this kind of method.
5. **Wrong type**. Run `TextInInt`: `int city = "Denver";` gives incompatible types: `String` cannot be converted to `int`. The message names the value's type and the variable's type. Declare it with the class that matches the object.

MISCONCEPTION String is just another primitive type

Students list `String` beside `int` and `double` because they use it the same way. It is a class from `java.lang`, so it defines a reference type, and its values are objects with behaviors such as `length()`. Ask, "Can an `int` do anything?" It cannot; a `String` can tell you its length.

End with slide 20 (Attribute or behavior?): pairs sort four things the `Scanner` documentation describes. (a) and (c) are attributes; (b) and (d) are behaviors. Push on (c): reading a number changes how far the scanner has read, so behaviors can use and change attributes.

Day 2 guided practice: take apart NameLength

Show `NameLength` without running it. A site wants usernames that are exactly 12 characters long, and the program reports how many characters `jo_rivera` is away from that. Pairs:

1. Circle every class name and label it "ours" or "library."
2. Fill a table with one row per library class: package, import needed, and whether the program declares a variable of that type or uses the class only through its name.
3. List every behavior the program uses.
4. Predict the output, then run it.
5. Change `"jo_rivera"` to `"jordan_rivera"` and predict again.

Answers: `NameLength` is ours; `Scanner` (`java.util`, import, type of `reader` and an object made with `new`), `String` (`java.lang`, no import, type of `name` and an object from the literal), `Math` (`java.lang`, no import, used only through the class name), and `System` (`java.lang`, no import, output only).

Behaviors: `nextInt`, `length`, `abs`, `println`. Output 3: `"jo_rivera"` has 9 characters, and `Math.abs(12 - 9)` is 3. After the change, `"jordan_rivera"` has 13 characters and `Math.abs(12 - 13)` is 1; `abs` keeps the answer positive whether the name is too short or too long.

Day 2 independent practice

Puzzle worksheet questions 2–9 (reading the `Math` documentation, be-the-compiler, code tiles and a snippet pool with `Math` and `length()`, the attribute-and-behavior face-off, a one-token fix, two truths and a lie about reference types and imports, and who am I) are homework.

Day 2 closure and exit ticket

Point to slide 21's cards and ask the class to finish two sentences aloud: "An attribute is ... and a behavior is ..." (data stored in variables; what objects can do, defined by methods) and "A class does two jobs: ..." (it defines a type, and it is a blueprint for objects). Then show slide 22 (Look it up, don't guess) and have one pair explain with the sentence starter. Closure takes about two minutes. Then give the exit ticket (about five minutes, timed separately). Question 1 checks sorting a class's attributes and behaviors and writing an import line (1.7.A.1, 1.7.A.2); question 2 checks packages and import (1.7.A.1). Re-teach the weaker one at the start of Topic 1.8.