

1.15 String Manipulation

Lesson Plan

Pacing: 5 class periods of 45 minutes

Learning objectives: 1.15.A, 1.15.B

Essential knowledge: 1.15.A.1, 1.15.A.2, 1.15.A.3, 1.15.A.4, 1.15.A.5, 1.15.B.1, 1.15.B.2, 1.15.B.3

Materials

- Projector
- Whiteboard and markers
- IDE on every computer
- String Puzzles slide deck
- Printed String Puzzles puzzle worksheet and exit ticket

Agenda

Minutes	Activity
5	Day 1 warm-up — take a file name apart (slide 2), then print the pieces
8	Day 1 direct instruction — creating String objects (slides 3–5)
9	Day 1 direct instruction — concatenation with + and += (slides 8–9)
9	Day 1 direct instruction — numbers, booleans, and left to right (slide 10)
7	Day 1 guided practice — predict the scoreboard
5	Day 1 independent practice — join or add? (slide 11) and puzzle worksheet question 3
2	Day 1 closure — when does + add?
5	Day 2 warm-up — count the characters
10	Day 2 direct instruction — indices and length() (slide 6)
12	Day 2 direct instruction — extracting text with substring (slides 12–13)
8	Day 2 guided practice — slice a team name
8	Day 2 independent practice — puzzle worksheet questions 1 and 4
2	Day 2 closure — included or not?
5	Day 3 warm-up — the last letter
12	Day 3 direct instruction — out-of-range indices (slide 15)
12	Day 3 direct instruction — strings are immutable (slide 7)
8	Day 3 guided practice — legal or exception?
6	Day 3 independent practice — puzzle worksheet questions 2 and 7
2	Day 3 closure — store it or lose it
5	Day 4 warm-up — where is the space?
12	Day 4 direct instruction — finding text with indexOf (slides 14 and 16)
12	Day 4 direct instruction — equals and compareTo (slides 17–19)

8	Day 4 guided practice — decode a product code
6	Day 4 independent practice — puzzle worksheet questions 5, 6, and 8
2	Day 4 closure — say it as a sentence
5	Day 5 warm-up — what is the text of an object?
10	Day 5 direct instruction — concatenating other objects (back to slide 9)
5	Day 5 guided practice — putting it together
17	Day 5 independent practice — code-writing lab (BadgeMaker), then puzzle worksheet questions 9 and 10
3	Day 5 closure — back to frog.png and words to know (slides 20–21)
5	Day 5 exit ticket

Overview

Five periods on the last topic of Unit 1. There are two learning objectives: 1.15.A, create `String` objects and determine the result of creating and combining them, and 1.15.B, call methods on `String` objects and determine the result. Each day covers one coherent slice of the study guide:

Day	Guide subsections	Content	Essential knowledge
1	1.15.1–1.15.3	Literals and the constructor, <code>java.lang</code> , <code>+</code> and <code>+=</code> , left to right	1.15.A.1, 1.15.A.2, 1.15.A.4
2	1.15.4–1.15.5	Indices, <code>length()</code> , both versions of <code>substring</code> , single characters	1.15.B.1, 1.15.B.2, 1.15.B.3
3	1.15.6–1.15.7	<code>StringIndexOutOfBoundsException</code> , immutability	1.15.B.1, 1.15.A.3
4	1.15.8–1.15.9	<code>indexOf</code> and <code>-1</code> , <code>equals</code> , <code>compareTo</code>	1.15.B.2
5	1.15.10	<code>toString</code> , method overriding; code-writing lab; exit ticket	1.15.A.5, all of 1.15.B

The CED's suggested skills are 2.C (write program code involving procedural abstractions), 3.C (determine the result or output of such code), 4.A (describe the behavior of code), and 4.B (describe the initial conditions code needs to work as intended). Every day has prediction before running (3.C, 4.A). Days 3 and 4 ask what must be true for a call to work, such as "the text must be present before you pass `indexOf`'s result to `substring`" (4.B). Day 5's lab has students write a complete program (2.C).

The String Puzzles deck is used across all five days, out of order where the days need it: Day 1 slides 2–5 and 8–11; Day 2 slides 6 and 12–13; Day 3 slides 15 and 7; Day 4 slides 14 and 16–19; Day 5 a return to slide 9, then slides 20–21. Its speaker notes hold the answers to every Pause and Try slide. The String Puzzles worksheet (ten questions, about 30 minutes) is handed out on Day 1 and done a few questions at a time, in class and as homework, through Day 5. It is cumulative through Topic 1.15 and focuses on creating, joining, cutting, searching, and comparing strings.

Stay inside the topic boundary:

- This is the end of Unit 1. No `if` statements, loops, relational operators (`<`, `==`), or `!` appear anywhere, including in teacher notes and model solutions. When a program could crash because text is missing, the answer is "make sure the text is there"; checking for it is Unit 2.
- The `String` methods are the six on the Java Quick Reference: `length()`, both `substring` versions, `indexOf(String str)`, `equals(Object other)`, and `compareTo(String other)`. No other `String` method appears; a single character is always `substring(index, index + 1)`. `toString` is the one method from `Object` the topic needs (1.15.A.5).
- The CED excludes overriding `toString` yourself and using `equals` to compare a `String` with an object of another type. Every `equals` call here passes a `String`.
- `compareTo` results are read by sign only. Some example programs print the actual number; tell students not to memorize it.
- Method headers for `main` start with `public static`. Students can read past them; they are explained in a later topic.

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. Line numbers here count the lines as shown.

- **Day 1, creating and combining:** `PrintPieces` (warm-up), `CreateStrings`, `CountCheck` (confirms the character counts quoted on Days 1 and 2), `MissingQuotes`, `FullName`, `NoSpace`, `BuildGreeting`, `PlaylistTrace`, `MixedValues`, `LeftToRight`, `Precedence`, and `ScoreLine` (guided practice).
- **Day 2, indices and substring:** `Lengths`, `Sunflower`, `SunflowerLetters` (the one-character calls), `EmptySubstrings`, `Flow With FlowFixed`, and `Seahawks` (guided practice).
- **Day 3, out of range and immutable:** `LastLetter` with `LastLetterFixed` (warm-up), `BadSubstring`, `StartTooBig`, `Backwards`, `NegativeStart`, `PlanetCalls`, `Plane`, `SharedString`, `IgnoredResult`, `SavedResult`, `Alias`, and `SummerCalls` with `SummerTooFar` and `SummerNegative` (guided practice).
- **Day 4, searching and comparing:** `FindSpace` (warm-up), `FindText`, `EmailParts`, `NoAtSign`, `EqualsDemo`, `CompareDemo`, `CapitalFirst`, `ProductCode` (guided practice), and `RomeOslo` (closure).
- **Day 5, other objects and the lab:** `ScannerText`, `CityText`, `Username`, and `BadgeMaker` (the lab's model solution) with `BadgeMakerSample` (the same program reading two fixed lines from a string instead of the keyboard).
- **Printed only (exit ticket):** `ExitSantaFe` and `ExitEmail`. The puzzle worksheet and its key are separate Word files; their code is printed on the page.

`BadgeMaker` reads the keyboard, so it waits for input when you run it. The exception messages quoted here (for example `Range [0, 7) out of bounds for length 6`) come from a recent JDK; older versions word the middle part differently, but the exception name and line number are the same. The `Scanner` text from `toString` is long and its details vary, so the examples print only its first characters, which always start `java.util.Scanner`.

Day 1 warm-up: print the pieces

Open with slide 2 (Take a file name apart): pairs spend two minutes writing plain-English steps to get `frog`, `png`, and `frog.jpg` from `"frog.png"`. Collect the recipes and say that this week gives each step a `String` tool; the class comes back to it on Day 5. Then show `PrintPieces`. Students write the output. Expected: `Ada Lovelace` on one line, built from three `print/println` calls, the way Topic 1.14 printed a label and a value.

Then ask: "What if we wanted `Ada Lovelace` stored in **one** variable, to print later or measure with `length()`?" Collect ideas; some students will guess `first + last`. Do not confirm yet; that guess is today's subject.

Day 1 direct instruction: creating String objects

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

1. **What a String is** (1.15.A.1, slide 5). A `String` object holds a **sequence of characters**: letters, digits, spaces, punctuation, in a fixed order. `"Number 23"` is nine characters; the digits are characters, not a number.
2. **Two ways to create one** (1.15.A.1). A **string literal** (`"Falcons"`) creates a `String` object. Calling the constructor with `new` (Topic 1.13), as in `new String("Denver")`, creates one too. Run `CreateStrings: Falcons, Denver, Number 23`. Both forms give an object you use the same way; programmers almost always write the literal. Slide 5 shows both forms with `"lily"` and `new String("pad")`.
3. **No import** (1.15.A.2). Ask: "What does a `Scanner` program need at the top that `CreateStrings` does not?" The `import` line. `String` is in the **java.lang** package, and classes in `java.lang` are available by default. `Scanner` is in `java.util` and must be imported.
4. **Predict: MissingQuotes**. Without quotes, `Falcons` is a name, so Java looks for a variable: `cannot find symbol` on line 5.

MISCONCEPTION Forgetting the quotes

The word looks like text to a person, so students leave off the quotes. Java then reads it as a variable name and reports `cannot find symbol`. Say it every time: "The quotes are what tell Java these characters are the value."

Day 1 direct instruction: concatenation with + and +=

Skip ahead to Part 2 (slide 8); slides 6 and 7 wait for Days 2 and 3.

1. **Concatenation** (1.15.A.4, slide 9). With a `String`, `+` joins two strings end to end into a **new** `String` object. Run `FullName: first + " " + last` gives `Ada Lovelace`, and `first` still prints `Ada`. Concatenation built a new object; it did not change `first`. This answers the warm-up question.
2. **Predict: NoSpace**. `first + last` prints `AdaLovelace`. Concatenation adds exactly what you give it; the space has to be its own literal, `" "`.
3. **+= builds a string in steps** (1.15.A.4). `greeting += name;` means `greeting = greeting + name;` (Topic 1.6). Trace `BuildGreeting` on the board, one row per line: `"Hello"`, `"Hello, "`, `"Hello, Ada"`, `"Hello, Ada!"`. Each `+=` made a new `String` and moved `greeting` to it.

4. **Quick check: `PlaylistTrace`.** Output `Rock` and `Jazz`, then `Pop`. Line 4 copied `genre`'s characters into a new string at that moment; changing `genre` on line 5 cannot reach back into `playlist`.

MISCONCEPTION Expecting + to add spaces

Students read `first + last` as two words and expect a space between them. Java adds nothing that is not written. `NoSpace` compiles and runs, so this is a logic error you find only by comparing the output with what you expected.

Day 1 direct instruction: numbers, booleans, and left to right

1. **Implicit conversion** (1.15.A.4). When one side of `+` is a `String` and the other is a primitive, Java turns the primitive into text first. Run `MixedValues`: `Score: 95, Price: $4.5, Passed: true`. The variables do not change; `score` is still the `int 95`.
2. **`+` has two jobs.** Two numbers: add. A `String` on either side: concatenate. Java decides one `+` at a time, **left to right**, after `*`, `/`, and `%` (Topic 1.3 precedence).
3. **Predict: `LeftToRight`** (slide 10). `"Total: " + 3 + 4` is `Total: 34`; `3 + 4 + " points"` is `7 points`; `"Total: " + (3 + 4)` is `Total: 7`; `"Half: " + 7 / 2` is `Half: 3` (integer division happens first). Slide 10 shows the same idea with `"Lap " + 2 + 1`, `2 + 1 + " laps"`, and `"Lap " + (2 + 1)`; fill in its step table one `+` at a time, then do the same on the board for `LeftToRight`.
4. **Predict: Precedence.** `"A" + 2 * 3 + 1` prints `A61`: `2 * 3` first, then left to right.

MISCONCEPTION Expecting a + chain to add the numbers

Students group the numbers in their heads. Java does not: by the time it reaches the `4`, the left side is already the string `"Total: 3"`. Have students rewrite each chain with the running result under each `+`, and remind them that parentheses force the addition:
`"Total: " + (3 + 4)`.

Day 1 guided practice: predict the scoreboard

Show `ScoreLine`. Pairs write all six lines of output before you run it. Expected:

Line	Output	Why
4	Record: 4-2	Each <code>+</code> has a <code>String</code> on the left, so each one concatenates
5	Games: 42	<code>"Games: " + 4</code> is already a <code>String</code> , so <code>+</code> 2 concatenates
6	Games: 6	Parentheses add first
7	6 games	<code>4 + 2</code> is two <code>int</code> s, so it adds before the <code>String</code> appears
8	Half: 2	<code>/</code> runs before <code>+</code> : <code>4 / 2</code> is 2
9	Playoffs: true	A <code>boolean</code> is converted to text, like a number

Day 1 independent practice

Students work slide 11 (Join or add?) alone, saying "add" or "join" for every `+: x is 10` and `3345`. Then hand out the String Puzzles worksheet; students do question 3 (the match-up of four nearly identical `println` statements). Whatever is not finished is homework. Check statement 4 of question 3: `4 + "" + 3` is 43, because the empty string turns the rest of the chain into joining.

Day 1 closure: when does + add?

Ask: "When does `+` add, and when does it concatenate?" Collect the rule in the class's words: it adds only when both sides are numbers; once the running result is a `String`, every later `+` concatenates. Ask one student how to make it add anyway (parentheses).

Day 2 warm-up: count the characters

Write `"Go Bears!"` on the board. Ask: "How many characters? Which position is the B?" Most students say 8 or 9, and position 4. Do not settle it; the answers are 9 and 3 (`CountCheck` confirms both), and today explains why.

Day 2 direct instruction: indices and length()

1. **Index** (1.15.B.1, slide 6). Each character has a position number, its **index**, starting at **0**. The index is "how many characters come before this one." Slide 6 shows the index diagram for `"frog"`; draw another for `"planet"`: p 0 through t 5.
2. **int length()** (1.15.B.2) returns the number of characters. Call it with the dot operator (Topic 1.14). Run Lengths: 6, 9, 0. In `"Go Bears!"` the space and the `!` are characters, so the length is 9 and B is at index 3.
3. **The empty string**. `""` is a `String` with no characters: length 0 and no valid indices.
4. **The rule** (1.15.B.1). Valid indices run from 0 to `length() - 1`. The last character is always at `length() - 1`.

MISCONCEPTION Treating the length as the last index

`"planet"` has 6 characters, so students say the last one is at index 6. It is at 5; index 6 would be a seventh character. Have students write the index row under every string this week until counting from 0 is automatic.

Day 2 direct instruction: extracting text with substring

Start Part 3 (slide 12).

1. **Two versions** (1.15.B.2, slide 13), both on the Java Quick Reference:
`substring(int from, int to)` returns the characters from index `from` up to, but **not including**, `to`; `substring(int from)` returns the characters from `from` to the end. The result of the first

has to - from characters. Slide 13 shows all three forms on "lilypad": `substring(0, 4)` is lily, `substring(4)` is pad, and `substring(3, 4)` is y.

2. **Predict: Sunflower.** Draw the index diagram (s 0 to r 8). `substring(0, 3)` is sun, `substring(3, 6)` is flo, `substring(3)` is flower, and `substring(4, 5)` is l.
3. **One character** (1.15.B.3). `substring(index, index + 1)` is the one-character string at index. This is how the course gets a single letter.
4. **Edges: EmptySubstrings.** `substring(9)` on a 9-character word starts just past the end and returns ""; `substring(2, 2)` takes zero characters and returns ""; `substring(8)` returns the last character, r. Both empty results are legal.
5. **Predict: Flow.** A student wants flow (f at 3, w at 6) and writes `substring(3, 6)`; it prints flo. `FlowFixed` uses `substring(3, 7)` and prints flow. Check: 4 characters, and $7 - 3$ is 4.

MISCONCEPTION Expecting the to character to be included

Students name the index of the last character they want as to. Teach two checks: "to is one past the last character I want" and "to - from is how many characters I get."

Day 2 guided practice: slice a team name

Write `String team = "Seahawks";` on the board. Pairs draw the index diagram and write a call that returns each of "Sea", "hawks", "h", and "awk", plus the value of `team.length()`. Answers: `substring(0, 3)`, `substring(3)`, `substring(3, 4)`, `substring(4, 7)`, and 8. Run `Seahawks` to confirm; its last line shows the alternative in the note below.

TEACHER NOTE

`team.substring(3, 8)` also returns "hawks", since 8 is the length. Accept it, and point out that the one-argument version saves you from working out the length.

Day 2 independent practice

Students work on puzzle worksheet questions 1 (what `"hopper".substring(1, 4)` joined with the length prints) and 4 (code tiles that flip "hopper" with `substring`, `length()` / 2, and `+=`). Whatever is not finished is homework.

Day 2 closure: included or not?

Ask two quick questions: "In `substring(from, to)`, is the character at from included? At to?" (Yes; no.) "How many characters does `substring(2, 7)` return?" (5.)

Day 3 warm-up: the last letter

Show `LastLetter`. It is supposed to print `Last letter: t`. Ask students to predict what happens. Run it: it compiles, then stops with a `StringIndexOutOfBoundsException`, and nothing is printed. `word.length()` is 6, so it asks for `substring(6, 7)`, and `"rocket"` has no index 6. Show `LastLetterFixed`, which uses `substring(word.length() - 1, word.length())` and prints `Last letter: t`.

Day 3 direct instruction: out-of-range indices

1. **The rules** (1.15.B.1). For a string of length `n`: a character's index runs from 0 to `n - 1`. In `substring(from, to)`, both values must be from 0 to `n`, and `from` must not be bigger than `to`. In `substring(from)`, `from` must be from 0 to `n`. (`to` may equal `n` because the character at `to` is never taken.)
2. **Predict: BadSubstring** (slide 15). It compiles and prints `Word: planet`, then line 7 throws `java.lang.StringIndexOutOfBoundsException: Range [0, 7) out of bounds for length 6`, and line 8 never runs. Read the message in parts: the exception name, the range requested, the length, and the line number. It is a run-time error (Topic 1.1): the compiler checks that the arguments are `ints`, not what their values will be. Slide 15 shows the same crash with `"frog".substring(2, 5): Range [2, 5) out of bounds for length 4`.
3. **Three more ways to break the rule**. `StartTooBig (substring(7))`, `Backwards (substring(4, 2))`, and `NegativeStart (substring(-1, 3))` all compile and throw the same exception on `"planet"`.
4. **Right at the edge is fine: PlanetCalls**. `substring(0, 6)` is `[planet]`, `substring(6)` is `[]`, and `substring(5, 6)` is `[t]`.
5. **Predict: Plane**. `substring(0, word.length() - 1)` prints `plane: the` - 1 drops the `t`. `substring(0, word.length())` is the whole word, `planet`.

MISCONCEPTION Being too careful with the end

Once students learn that index `length()` does not exist, some subtract 1 everywhere, including in the `to` argument, and lose the last character. The `- 1` belongs only where you name one character's own position, as in `length() - 1` for the last index. As a `to` argument, `length()` is correct.

Day 3 direct instruction: strings are immutable

1. **Immutable** (1.15.A.3, slide 7). Once a `String` object is created, it can never be changed. Every `String` method leaves the string alone and **returns** a new string.
2. **Why it is safe**. Several variables can refer to the same `String` object (Topic 1.12). If one could change it, all the others would see their text change. Immutability makes sharing safe.
3. **Predict: SharedString**. `original` and `other` share `"cat"`. `original += "s"` builds a new object `"cats"` and moves only `original`. Output: `cats`, then `cat`. Draw both objects and the arrows.
4. **Predict: IgnoredResult**. Slide 7 shows the same trap with `pond.substring(1);` `pond` still prints `frog` until the result is stored. `word.substring(0, 3);` on its own line builds `"sun"` and throws it away; `word` still prints `sunflower`. `SavedResult` stores it, `word = word.substring(0, 3);`, and prints `sun`. The `"sunflower"` object still did not change; `word` now refers to a new object.

5. **Check: Alias.** Students choose the output before you reveal it: `sunflower sun`. Line 3 moves only `a`; line 4's result is not stored, so `b` is still `sun`.

MISCONCEPTION Calling a method and expecting the string to change

Students write `word.substring(0, 3);` as an instruction to "cut the word down." It compiles and runs with no visible effect. Ask of every `String` call: "Where does the returned string go?" If the answer is "nowhere," it is lost.

Day 3 guided practice: legal or exception?

Write `String tag = "summer";` (length 6) and the six calls below on the board. For each call, pairs write the result in brackets or "exception":

Call	Result	Reason
<code>tag.substring(6)</code>	<code>[]</code>	from may equal the length
<code>tag.substring(2, 7)</code>	exception	to is above the length
<code>tag.substring(3, 3)</code>	<code>[]</code>	from equal to to takes nothing
<code>tag.substring(-1)</code>	exception	negative index
<code>tag.substring(0, tag.length())</code>	<code>[summer]</code>	to may equal the length
<code>tag.substring(5, 6)</code>	<code>[r]</code>	one character, the last index

Then ask what `tag` holds after `tag.substring(0, 3);` runs on its own line: still `summer`. Run `SummerCalls` for the legal calls and the last question; run `SummerTooFar` and `SummerNegative` to show the two exceptions (Range `[2, 7)` out of bounds for length 6 and Range `[-1, 6)` out of bounds for length 6).

Day 3 independent practice

Students work on puzzle worksheet questions 2 (be the compiler: `length` without parentheses, a `substring` result stored in an `int`, and three that compile) and 7 (two truths and a lie about whether `s` changes after `substring`). Whatever is not finished is homework. Check question 7: students who circle claim 1 or 2 still think `substring` changed `s`.

Day 3 closure: store it or lose it

Ask the class to finish the sentence together: "A `String` method never changes the string; if I want the result, I ..." (store it in a variable, or use it right away, such as printing it).

Day 4 warm-up: where is the space?

Write `"Grace Hopper"` on the board. Ask: "What is the index of the space? How would you get `"Grace"` with `substring`?" (5; `substring(0, 5)`.) Then ask: "Would the same call work for

"Ada LoveLace"?" No: that space is at index 3. A program needs to **find** the space. That is today's first method.

Day 4 direct instruction: finding text with indexOf

Go to slide 14.

1. `int indexOf(String str)` (1.15.B.2) returns the index where the **first occurrence** of `str` begins, or `-1` if it does not appear. `-1` can never be a real index, so it can only mean "not found." Students met this method in Topic 1.14; today it works with `substring`.
2. **Predict: FindText.** In "the cat sat on the mat": "the" is 0 (the first of two), "at" is 5 (inside cat), "dog" is `-1`, and "The" is `-1` because the search is case-sensitive.
3. **Combine with substring: EmailParts.** Slide 14's right side does the same with "tadpole": `poLe` starts at 3, so `substring(0, 3)` is tad. `at` is 3; `substring(0, at)` is ada (stops right before the @); `substring(at + 1)` is school.org (starts one past it). It works for any address with an @, however long the user name.
4. **Predict: NoAtSign.** No @, so `at` is `-1`. Line 7 prints `-1`; line 8, `substring(0, -1)`, throws a `StringIndexOutOfBoundsException`. The code works only if the text is present: that is the condition the code needs (CED skill 4.B). Checking for it is Unit 2.
5. **Pause and try** (slide 16). Pairs find, cut, and join "pollywog": 5, 5, and wpolly.

MISCONCEPTION Feeding `-1` straight into `substring`

Students forget that "not found" is a possible answer. Ask, every time an `indexOf` result goes into `substring`: "What does this line need to be true to work?" State the answer as a condition: "the text must contain an @."

Day 4 direct instruction: equals and compareTo

Start Part 4 (slide 17).

1. `boolean equals(Object other)` (1.15.B.2, slide 18) returns `true` when the two strings have exactly the same characters: same length, same order, same capitalization. In this course the argument is always a `String`. Run `EqualsDemo`: `true`, `false` (t vs T), `false` (extra space), `true` (a separate object made with the constructor, same characters).
2. `int compareTo(String other)` (1.15.B.2) compares alphabetical order. `a.compareTo(b)` is negative if `a` comes first, zero if they are equal, and positive if `a` comes after. Only the sign matters. Run `CompareDemo` and read only the signs: negative, negative, positive, zero, negative (car before cart: shorter is first, as in a dictionary).
3. **Uppercase first: CapitalFirst.** Java puts every uppercase letter before every lowercase letter, so "Zoo" comes before "apple" (negative) while "zoo" comes after it (positive). With words in one case, Java's order matches the dictionary.

MISCONCEPTION Reading compareTo backwards

Students flip which string is which, or read negative as "smaller" in some other sense. Have them say it as a sentence: "apple compared to banana is negative, so apple comes before banana." The string in front of the dot is the subject of the sentence.

End with slide 19 (Negative, zero, or positive?): negative, positive (do is a prefix of dog, so it comes first), zero, and false.

Day 4 guided practice: decode a product code

Show `ProductCode`. The code "BLU-0042-XL" holds a color, an item number, and a size, separated by dashes. Pairs trace the variables and predict all four lines of output, saying only the sign for the last one. Expected: 3 BLU 0042-XL, 0042 XL, false, and a negative number.

Line	Value
2	<code>dash = 3</code> (B 0, L 1, U 2, dash 3)
3	<code>color = "BLU"</code>
4	<code>rest = "0042-XL"</code>
5	In rest, the dash is at 4, so <code>number = "0042"</code>
6	<code>size = rest.substring(5) = "XL"</code>
9	<code>"BLU".equals("blu")</code> is false: capitalization counts
10	<code>"BLU".compareTo("RED")</code> is negative: B comes before R

Point out that "0042" keeps its zeros because it is text, and that line 7 concatenates the `int` 3 with strings.

Day 4 independent practice

Students work on puzzle worksheet questions 5 (a snippet pool that turns "Grace Hopper" into Hopper, Grace with `indexOf` and `substring`), 6 (the `substring` and `indexOf` face-off), and 8 (a pencil check of `length`, `indexOf`, `substring`, `equals`, and `compareTo` on "lighthouse"). Whatever is not finished is homework.

Day 4 closure: say it as a sentence

Write `"Rome".compareTo("Oslo")` on the board. A volunteer says the sentence: "Rome compared to Oslo is positive, so Rome comes after Oslo." Then ask what `indexOf` returns when the text is missing (-1).

Day 5 warm-up: what is the text of an object?

Ask: "In `"Score: " + 95`, Java turns 95 into the characters 9 and 5. What could Java do with `"Input: " + input` when input is a `Scanner`?" Collect guesses ("what the user typed," "an error," "nothing"). Today's first idea answers it.

Day 5 direct instruction: concatenating other objects

1. **Every object has `toString`** (1.15.A.5). Every class is a subclass of `Object` (Topic 1.12), and `Object` has a method `String toString()` that returns a string representing the object. So every object has one.
2. **Concatenation calls it.** When a `String` is concatenated with any object, Java calls that object's `toString` automatically and joins the returned text. Go back to slide 9: its last sentence and its `"lily" + s` example say this, and the Fair Question box after puzzle worksheet question 3 does too.
3. **Method overriding** (1.15.A.5). Classes often replace `Object`'s general `toString` with their own. When a subclass has a public method with the same signature as a public method in its superclass but its own behavior, that is **method overriding**. `String` overrides `toString` to return its own characters; `Scanner` overrides it to describe the scanner. Writing your own `toString` is outside the course.
4. **Run `ScannerText`.** `"Input: " + input` calls `input.toString()`. The program prints only the first characters: `Input: java.util.Scanner, java.util.Scanner, and 24` (the `[` that starts the scanner's settings). Nothing the user typed appears, and the program never waits for typing.
5. **Run `CityText`.** A `String` made with the constructor is also an object; its `toString` returns its own characters: `City: Denver, Denver`.

MISCONCEPTION Concatenating a `Scanner` reads what the user typed

`"Input: " + input` looks as if it should show the input. It only calls `toString`, which describes the `Scanner` object; it reads nothing. To get what the user typed, call a reading method such as `input.nextLine()` (Topic 1.4) and concatenate the `String` it returns.

Day 5 guided practice: putting it together

Show `Username`. Pairs predict the three lines, giving only the sign for the second. Expected: `GHopper12`, a negative number (`Grace` before `Hopper`), and `Grace Hopper`. Build the trace table on the board: `indexOf` finds the space at 5, two `substring` calls split the name, `substring(0, 1)` is the first letter, the `int 12` is concatenated last, and `fullName` never changed.

Day 5 independent practice: code-writing lab

Write the task on the board, and students write it in the IDE: a `BadgeMaker` program that reads a full name and a graduation year from the keyboard and prints a badge line, initials, an ID, and the number of letters. Test input: `Katherine Johnson` and `2028`. Expected last four lines:

Badge: `Johnson, Katherine`, Initials: `KJ`, ID: `KJohnson28`, and Letters: `16`. Students who finish test it with a second name of their choice and then finish the puzzle worksheet: questions 9 (who

am I, for the `String` methods) and 10 (why no `String` method can change its string). Questions not finished are homework.

TEACHER NOTE

Watch for three slips:

- Reading the name with `next()` gets only the first name; `nextInt()` then finds the last name and throws an `InputMismatchException`. The name needs `nextLine()`.
- `"Letters: " + fullName.length() - 1` does not compile (bad operand types for binary operator '-'): left to right, the `+` makes a `String` first, and Java cannot subtract from a `String`. Parentheses fix it:
`"Letters: " + (fullName.length() - 1)`.
- For the year's last two digits, `"" + year` turns the `int` into a `String` (as in `BuildCode` from the study guide), and `substring(2)` takes the last two characters of a four-digit year.

Day 5 closure: the `String` method table

Show slide 21 (Back to frog.png): pairs fill in the four blanks using the week's tools (`indexOf(".")`, `substring(0, dot)`, `substring(dot + 1)`, and `name + ".jpg"`), which answers the Day 1 warm-up. Then show slide 20 (Words to know): point to `substring`, `indexOf`, and `compareTo` and ask what each returns and how it can go wrong. End with the one rule that applies to all of them: the original string never changes.

Day 5 exit ticket

Give the exit ticket (about five minutes). Question 1 checks determining the result of `indexOf`, `substring`, concatenation, and an ignored result, plus the out-of-range exception (1.15.A.3, 1.15.A.4, 1.15.B.1, 1.15.B.2); question 2 checks writing code that combines `indexOf`, `substring`, `length()`, and concatenation (1.15.A.4, 1.15.B.2). Any puzzle worksheet questions still not finished are homework; collect the whole worksheet next class. Re-teach the weakest area before the Unit 1 assessment.