

1.15 String Manipulation

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Create String objects with a string literal or with the String constructor, and explain why String needs no import.
- Join strings, numbers, booleans, and other objects with + and +=, and predict the exact text that results.
- Use index positions and length() to describe where each character of a string sits.
- Extract parts of a string with substring and find text with indexOf, including the -1 result.
- Predict when a String method throws a StringIndexOutOfBoundsException.
- Explain what it means that strings are immutable, and compare strings with equals and compareTo.

What to remember from earlier

- A string literal is text in double quotes, printed with System.out.println. ([Topic 1.3: Expressions and Output](#), [1.3.3 Literals and String Literals](#))
- String is a reference type. ([Topic 1.2: Variables and Data Types](#), [1.2.2 Primitive Types and Reference Types](#))
- An object is made from a class, and new calls a constructor. ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.1 Constructors and the new Keyword](#))
- A reference variable holds a reference to an object. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.4 Reference Variables](#))
- Every class is a subclass of the Object class. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.3 Every Class Is a Subclass of Object](#))
- You call an instance method with the dot operator, as in name.length(). ([Topic 1.14: Calling Instance Methods](#), [1.14.2 The Dot Operator](#))

1.15.1 Creating String Objects

Almost every program works with text: names, messages, passwords, file names, answers typed by a user. Since Topic 1.3 you have been writing text as string literals and printing it. In Topics 1.12 to 1.14 you learned that `String` is a class, so text in a Java program is stored as objects. This chapter is about what you can do with those objects. First, though, how does a `String` object come into existence?

A **String** object holds a **sequence of characters**: letters, digits, spaces, punctuation, one after another in a fixed order. `"Number 23"` is nine characters long: N, u, m, b, e, r, a space, 2, and 3. The digits in it are characters, not a number you can do arithmetic with.

There are two ways to make one:

1. Write a **string literal**, text inside double quotes. Java creates a `String` object holding that text.

2. Call the `String` class's **constructor** with `new`, the same way you created objects in Topic 1.13. Passing a literal, as in `new String("Denver")`, creates a new `String` object with the same characters.

KEY IDEA

A `String` object is a sequence of characters. You can create one with a string literal (`"Falcons"`) or by calling the `String` constructor (`new String("Denver")`).

WORKED EXAMPLE Two ways to create a String

Look at lines 5 and 6: one uses a literal, the other calls the constructor.

CreateStrings.java

```
1 public class CreateStrings
2 {
3     public static void main(String[] args)
4     {
5         String team = "Falcons";
6         String city = new String("Denver");
7         String jersey = "Number 23";
8         System.out.println(team);
9         System.out.println(city);
10        System.out.println(jersey);
11    }
12 }
```

Line	What Java does
5	Creates a <code>String</code> object holding <code>Falcons</code> ; <code>team</code> holds a reference to it
6	<code>new</code> calls the <code>String</code> constructor, which builds an object holding <code>Denver</code>
7	Creates a <code>String</code> object holding <code>Number 23</code> (the space and digits are characters)
8–10	Prints the characters of each object

```
Falcons
Denver
Number 23
```

The literal is shorter and is what programmers use almost all the time. Both forms give you a `String` object you can use in exactly the same ways.

Now notice something that is **not** in this program: an `import` statement. When you used `Scanner` in Topic 1.4, the program needed `import java.util.Scanner`; at the top, because `Scanner` lives in the `java.util` **package** (a named group of related classes). `String` lives in a package called **java.lang**, and Java makes every class in `java.lang` available to every program automatically. Java does this because classes like `String` are needed by nearly every program, so requiring an `import` for them would just be clutter.

WATCH OUT

Forgetting the quotes. Without double quotes, Java does not see text; it sees a name, and it goes looking for a variable called `Falcons`:

MissingQuotes.java

```
1 public class MissingQuotes
2 {
3     public static void main(String[] args)
4     {
5         String team = Falcons;
6         System.out.println(team);
7     }
8 }
```

The compiler reports `MissingQuotes.java:5: error: cannot find symbol, because no variable named Falcons exists`. Students make this slip because the word looks like text to a person. The fix is `String team = "Falcons";`. The quotes are what tell Java "these characters are the value."

PAUSE AND TRY**Question 1** Multiple choice

A program contains `String title = "Hamlet";` and has no `import` statements at all. What happens when it is compiled?

- A. A compile error, because `String` must be imported before it can be used
- B. It compiles, because `String` is in `java.lang`, which every program can use without an import
- C. It compiles only if the line is changed to `String title = new String("Hamlet");`
- D. It compiles, but the program crashes when it reaches that line

1.15.2 Concatenating Strings

Suppose a program knows a student's first name and last name in two separate variables, and it needs to print the full name. Or it needs to build a message like `Hello, Ada!` from pieces. You need a way to glue strings together.

In plain words: put the second string's characters right after the first string's characters, with nothing added and nothing removed. Joining strings end to end is called **concatenation**. In Java, the `+` operator concatenates when it is used with a `String`. The result is a **new** `String` object; the two original strings are left exactly as they were.

KEY IDEA

`+` with strings concatenates: it creates a new `String` holding the first string's characters followed by the second's.

WORKED EXAMPLE Building a full name

Watch the space in line 3: it is its own string literal, " ", holding one space character.

```
1 String first = "Ada";
2 String last = "LoveLace";
3 String full = first + " " + last;
4 System.out.println(full);
5 System.out.println(first);
```

Java evaluates line 3 from left to right, one + at a time:

Step	Expression	Result (a new String)
1	first + " "	"Ada "
2	"Ada " + last	"Ada LoveLace"
3	full gets a reference to	"Ada LoveLace"

```
Ada LoveLace
Ada
```

The second line of output shows that `first` still holds `Ada`. Concatenation built a new object; it did not change `first`.

Now vary the tool. Compound assignment from Topic 1.6 works with strings too:

`greeting += name;` means `greeting = greeting + name;`. It concatenates, creates a new `String`, and makes the variable refer to that new object. This is handy for building a message one piece at a time:

```
1 String name = "Ada";
2 String greeting = "Hello";
3 greeting += ", ";
4 greeting += name;
5 greeting += "!";
6 System.out.println(greeting);
```

After line	greeting refers to
2	"Hello"
3	"Hello, "
4	"Hello, Ada"
5	"Hello, Ada!"

```
Hello, Ada!
```

Each `+=` made a brand-new `String` object and moved `greeting` to it. You will see why that matters in 1.15.7.

WATCH OUT

Expecting + to add spaces. Concatenation adds exactly what you give it and nothing else:

```
1 String first = "Ada";
2 String last = "LoveLace";
3 System.out.println(first + last);
```

AdaLoveLace

This compiles and runs, so it is a logic error. Students expect a space because people read two words as separate. Java does not. Put the space in yourself: `first + " " + last`.

PAUSE AND TRY

Question 2 Tracing

What does this code print? Track what `playlist` and `genre` refer to after each line.

```
1 String playlist = "Rock";
2 String genre = "Jazz";
3 playlist += " and ";
4 playlist += genre;
5 genre = "Pop";
6 System.out.println(playlist);
7 System.out.println(genre);
```

1.15.3 Concatenating Numbers and Booleans

Programs constantly mix text with values: `Score: 95`, `Price: $4.5`, `Passed: true`. The number 95 is an `int`, not a `String`. Can `+` join them?

Yes. When one side of `+` is a `String` and the other side is a primitive value (`int`, `double`, or `boolean`), Java first turns the primitive into its text form, then concatenates. The `int` 95 becomes the characters 9 and 5. Java does this on its own, without you asking, so it is called an **implicit conversion**.

WORKED EXAMPLE Text plus values

```
1 int score = 95;
2 double price = 4.5;
3 boolean passed = true;
4 System.out.println("Score: " + score);
5 System.out.println("Price: $" + price);
6 System.out.println("Passed: " + passed);
```

Score: 95
Price: \$4.5
Passed: true

In each line, the left side of `+` is a `String`, so Java converts the right side to text (95, 4.5, true) and concatenates. The variables themselves do not change: `score` is still the `int` 95.

The tricky part is that `+` has two jobs. With two numbers it **adds**; with a `String` on either side it **concatenates**. Java decides one `+` at a time, moving left to right (the same left-to-right rule for equal-precedence operators you learned in Topic 1.3). So the order of the pieces changes the result:

```
1 System.out.println("Total: " + 3 + 4);
2 System.out.println(3 + 4 + " points");
3 System.out.println("Total: " + (3 + 4));
4 System.out.println("Half: " + 7 / 2);
```

```
Total: 34
7 points
Total: 7
Half: 3
```

Line	First <code>+</code>	Second <code>+</code>	Result
1	"Total: " + 3 → "Total: 3" (string)	"Total: 3" + 4 → concatenate	"Total: 34"
2	3 + 4 → 7 (two ints: add)	7 + " points" → concatenate	"7 points"
3	Parentheses first: 3 + 4 → 7	"Total: " + 7 → concatenate	"Total: 7"
4	/ has higher precedence: 7 / 2 → 3 (int)	"Half: " + 3 → concatenate	"Half: 3"

Once the running result is a `String`, every later `+` in that chain concatenates. Before that, `+` between two numbers still adds. Precedence still applies too: `*`, `/`, and `%` happen before any `+`.

KEY IDEA

A `+` with a `String` on either side concatenates, turning a number or boolean into text. Java works left to right, so `"Total: " + 3 + 4` gives `"Total: 34"`; use parentheses to add first.

WATCH OUT

`"Total: " + 3 + 4` is **not** `Total: 7`. Students read the line as "the label, then 3 plus 4," but Java has no idea which pieces you meant to group. By the time it reaches `4`, the left side is already the string `"Total: 3"`, so it concatenates. If you want the sum, write `"Total: " + (3 + 4)`.

PAUSE AND TRY

Question 3 Tracing

Write the exact output.

```

1 int goals = 2;
2 int assists = 3;
3 System.out.println("Points: " + goals + assists);
4 System.out.println(goals + assists + " points");
5 System.out.println(1 + 2 + "3" + 4 + 5);

```

1.15.4 Indices and the length Method

To work with part of a string, such as its first letter or the text after an @, you need a way to say *where* in the string you mean. Java numbers the positions.

Each character in a string has a position number called its **index** (plural: **indices**). The first character is at index **0**, not 1. Then each character, including spaces and punctuation, gets the next number. Think of the index as "how many characters come before this one": the first character has zero characters before it.

```

index:      0 1 2 3 4 5
character:  p l a n e t

```

The **length** of a string is how many characters it has. You get it by calling the instance method `length()`, whose header is `int length()`: it takes no arguments and returns an `int`. You call it with the dot operator from Topic 1.14, as in `word.length()`. "planet" has length 6, but its last index is 5, because counting started at 0.

KEY IDEA

Indices run from 0 to `length() - 1`. The last character is always at index `length() - 1`.

WORKED EXAMPLE Counting characters

```

1 String word = "planet";
2 String cheer = "Go Bears!";
3 String empty = "";
4 System.out.println(word.length());
5 System.out.println(cheer.length());
6 System.out.println(empty.length());

```

```

6
9
0

```

String	Characters counted	length()	Last index
"planet"	p l a n e t	6	5
"Go Bears!"	G o (space) B e a r s !	9	8
" "	none	0	none

The space in "Go Bears!" is a character, and so is the !. Every one of them takes an index:

```
index:      0 1 2 3 4 5 6 7 8
character:  G o   B e a r s !
```

The last string, "", is the **empty string**: a `String` object with no characters at all. Its length is 0 and it has no valid indices.

WATCH OUT

Treating the length as the last index. Because "planet" has 6 characters, students often say the last character is at index 6. It is at index 5. Index 6 would be a seventh character, and there is none. In 1.15.6 you will see that asking for it crashes the program.

PAUSE AND TRY

Question 4 Multiple choice

For `String motto = "Be kind.";`, which statement is true?

- A. `motto.length()` is 7, and the `k` is at index 4.
- B. `motto.length()` is 8, and the `k` is at index 3.
- C. `motto.length()` is 8, and the last index is 8.
- D. `motto.length()` is 6, because spaces and periods are not counted.

1.15.5 Extracting Text with substring

With indices you can describe any part of a string. Now you need a method that hands that part back to you: the first three letters of a word, everything after the @ in an email address, one single letter. A part of a string is called a **substring**, and the `String` method that gets one has the same name.

`substring` comes in two versions. They are on the **Java Quick Reference**, the sheet of method headers you can look at during the AP exam:

Header	Returns
<code>String substring(int from, int to)</code>	The characters from index <code>from</code> up to, but not including, <code>to</code>
<code>String substring(int from)</code>	The characters from index <code>from</code> to the end of the string

Take the two-parameter version apart:

- `from` is the index of the first character you want. It is **included**.
- `to` is the index where you stop. The character at `to` is **not included**; the last character you get is at `to - 1`.
- So the result has `to - from` characters.

Why stop just before `to`? Because then `to - from` is the length of the result, and `substring(0, word.length())` is the whole word without any "minus one." The one-parameter

version is a shortcut: `word.substring(from)` gives the same result as `word.substring(from, word.length())`.

KEY IDEA

`substring(from, to)` returns the characters at indices `from` through `to - 1`; it includes `from` and stops before `to`. `substring(from)` goes from `from` to the end.

WORKED EXAMPLE Pieces of sunflower

First the index diagram:

```
index:      0 1 2 3 4 5 6 7 8
character:  s u n f l o w e r
```

```
1 String word = "sunflower";
2 System.out.println(word.substring(0, 3));
3 System.out.println(word.substring(3, 6));
4 System.out.println(word.substring(3));
5 System.out.println(word.substring(4, 5));
```

Call	Indices taken	Count	Result
<code>substring(0, 3)</code>	0, 1, 2	$3 - 0 = 3$	"sun"
<code>substring(3, 6)</code>	3, 4, 5	$6 - 3 = 3$	"flo"
<code>substring(3)</code>	3 through 8	$9 - 3 = 6$	"flower"
<code>substring(4, 5)</code>	4	$5 - 4 = 1$	"l"

```
sun
flo
flower
l
```

The last call shows a pattern worth remembering: `substring(index, index + 1)` gives a one-character string, the character at `index`. This is how you get a single letter out of a string in this course.

Now vary the indices to the edges. What happens when `from` equals `to`, or when `from` is the length?

```
1 String word = "sunflower";
2 System.out.println "[" + word.substring(9) + "]";
3 System.out.println "[" + word.substring(2, 2) + "]";
4 System.out.println "[" + word.substring(8) + "]";
```

```
[]
>[]
[r]
```

The brackets are concatenated around each result so you can see it. `substring(9)` starts at index 9, which is just past the last character, so there is nothing to take: the result is the empty string. `substring(2, 2)` asks for "from 2, stopping before 2," which is also zero characters. And `substring(8)` starts at the last index, so it returns the last character, "r". Both empty results are legal; the next subsection shows which indices are not.

WATCH OUT

Expecting the `to` character to be included. A student wants "fLow" from "sunflower". The `f` is at index 3 and the `w` is at index 6, so they write:

```
1 String word = "sunflower";
2 String part = word.substring(3, 6);
3 System.out.println(part);
```

fLo

The `w` at index 6 is where `substring` *stops*, so it is left out. The fix is to use one past the last index you want:

```
1 String word = "sunflower";
2 String part = word.substring(3, 7);
3 System.out.println(part);
```

fLow

A quick check: "fLow" has 4 characters, and $7 - 3$ is 4.

PAUSE AND TRY**Question 5** Tracing

Write the exact output.

```
1 String sport = "basketball";
2 System.out.println(sport.substring(6));
3 System.out.println(sport.substring(0, 6));
4 System.out.println(sport.substring(2, 3));
5 System.out.println(sport.substring(5, 7) + sport.substring(0, 1));
```

Question 6 Debugging

This code is supposed to print the three-letter abbreviation `Sat`, but it prints `Sa`. Find the error and write the corrected line.

```
1 String day = "Saturday";
2 String shortName = day.substring(0, 2);
3 System.out.println(shortName);
```

1.15.6 Out-of-Range Indices

What if the program asks for characters that are not there, such as index 7 of a six-letter word? In Topic 1.1 you saw that some impossible requests are not caught by the compiler but crash the program while it runs. Asking for a missing index is one of them.

The rules, for a string of length `n`:

- A single character position must be from 0 to `n - 1`.
- For `substring(from, to)`, both values must be from 0 to `n`, and `from` must not be bigger than `to`. (`to` may equal `n` because the character at `to` is never taken.)
- For `substring(from)`, `from` must be from 0 to `n`.

Break a rule and Java throws a **StringIndexOutOfBoundsException**, a run-time error whose name says exactly what happened: a string index was out of bounds (outside the allowed range). The compiler cannot catch it, because it checks only that you passed `int` values, not what those values will be when the line runs.

KEY IDEA

Valid indices run from 0 to `length() - 1`. Asking `substring` for an index outside its allowed range throws a `StringIndexOutOfBoundsException` while the program runs.

WORKED EXAMPLE One index too far

BadSubstring.java

```

1 public class BadSubstring
2 {
3     public static void main(String[] args)
4     {
5         String word = "planet";
6         System.out.println("Word: " + word);
7         System.out.println(word.substring(0, 7));
8         System.out.println("Done");
9     }
10 }
```

Moment	What happens	Console
Compile	Every line follows Java's rules, so it compiles	(nothing)
Line 6	Concatenates and prints	Word: planet
Line 7	"planet" has length 6, so <code>to</code> may be at most 6. 7 is out of range; exception thrown	exception message
Line 8	Never runs	(nothing)

```

Word: planet
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [0, 7) out
of bounds for length 6
    at BadSubstring.main(BadSubstring.java:7)
```

Read the message in parts: the exception's name, then the range that was requested (`[0, 7)` means "from 0, stopping before 7"), the string's length, and the line number. Older versions of Java word the middle part differently (for example `begin 0, end 7, length 6`), but the exception name and line number are what matter.

Now vary which rule is broken. Each of these also compiles and then throws the same exception:

Call on "planet"	Problem
<code>word.substring(7)</code>	Start is past the end (largest is 6)
<code>word.substring(4, 2)</code>	Start is bigger than stop
<code>word.substring(-1, 3)</code>	Negative index

```
1 String word = "planet";
2 System.out.println(word.substring(7));
```

```
1 String word = "planet";
2 System.out.println(word.substring(4, 2));
```

```
1 String word = "planet";
2 System.out.println(word.substring(-1, 3));
```

Compare those with calls that sit right at the edge and are fine:

```
1 String word = "planet";
2 System.out.println "[" + word.substring(0, 6) + "]";
3 System.out.println "[" + word.substring(6) + "]";
4 System.out.println "[" + word.substring(5, 6) + "]";
```

```
[planet]
[]
[t]
```

`substring(0, 6)` uses `to = 6`, the length, which is allowed. `substring(6)` starts at the length and returns the empty string. `substring(5, 6)` is the single-character pattern for the last index.

WATCH OUT

Being too careful with the end. Once students learn that index `length()` does not exist, some start subtracting 1 everywhere, including in the `to` argument:

```
1 String word = "planet";
2 System.out.println(word.substring(0, word.length() - 1));
3 System.out.println(word.substring(0, word.length()));
```

```
pplane
planet
```

The first line drops the `t`. Because `substring` already stops *before* `to`, the `to` argument may be `length()`, and `substring(0, word.length())` is the whole word. The `- 1` belongs only where you name a single character's position, such as the last index, `length() - 1`.

PAUSE AND TRY

Question 7 Multiple choice

For `String code = "AP2026";`, which call throws a `StringIndexOutOfBoundsException`?

- A. `code.substring(6)`
- B. `code.substring(0, 6)`
- C. `code.substring(3, 3)`
- D. `code.substring(2, 7)`

1.15.7 Strings Are Immutable

You have called `substring` and used `+=` many times now. Here is a question you may not have asked: when you call `word.substring(0, 3)`, does `word` lose its other letters?

No. A `String` object can never be changed once it is created. Every method you call on a string leaves that string exactly as it was and **returns** a new string as its answer. An object that cannot be changed after it is created is **immutable**. Methods like `substring` are like asking someone to read you part of a page: you get the words, but the page is not torn.

Why would Java design strings this way? Because several variables can refer to the same `String` object (Topic 1.13: a reference variable holds the object's address, and copying the variable copies the address). If one part of a program could change the object, every other variable referring to it would see its text change unexpectedly. Because strings are immutable, sharing them is always safe.

KEY IDEA

Strings are immutable: no method changes a `String` object. Methods return a new `String`, and to keep it you must store it, for example `word = word.substring(0, 3);`.

WORKED EXAMPLE Two variables, one string

Topic 1.13 showed that assigning one reference variable to another makes both refer to the same object. Watch what `+=` does when two variables share a string.

```
1 String original = "cat";
2 String other = original;
3 original += "s";
4 System.out.println(original);
5 System.out.println(other);
```

After line	<code>original</code> refers to	<code>other</code> refers to	Objects that exist
------------	---------------------------------	------------------------------	--------------------

1	object A	(none yet)	A = "cat"
2	object A	object A	A = "cat"
3	object B	object A	A = "cat", B = "cats"

```
cats
cat
```

Line 3 did not add an s to object A. It concatenated "cat" and "s" into a new object B and moved original to it. other still refers to A, which still holds cat, because nothing can change A.

Now vary it: call a method but do nothing with what it returns.

```
1 String word = "sunflower";
2 word.substring(0, 3);
3 System.out.println(word);
```

```
sunflower
```

Line 2 runs. substring builds the new string "sun" and returns it, but nothing stores the result, so it is thrown away. word still refers to the original, unchanged "sunflower".

WATCH OUT

Calling a method and expecting the string to change. Students write `word.substring(0, 3);` on its own line, the way they might write an instruction to "cut the word down." It compiles and runs, but it has no visible effect, which makes it a confusing logic error. Store the returned string:

```
1 String word = "sunflower";
2 word = word.substring(0, 3);
3 System.out.println(word);
```

```
sun
```

The original "sunflower" object was still not changed. word was moved to refer to the new object "sun".

PAUSE AND TRY

Question 8 Tracing

Write the exact output and explain why the first line is what it is.

```

1 String motto = "Practice daily";
2 String copy = motto;
3 motto.substring(9);
4 copy = copy.substring(0, 8);
5 System.out.println(motto);
6 System.out.println(copy);

```

1.15.8 Finding Text with indexOf

`substring` needs index numbers, but often you do not know them in advance. Where does the username end in an email address? Where is the space in a full name? You need to search for the text and get back where it is.

The method `int indexOf(String str)` searches the string for `str` and returns the index where its **first occurrence** starts. If `str` is not in the string anywhere, it returns `-1`. Java uses `-1` because it can never be a real index (indices start at 0), so it can only mean "not found."

KEY IDEA

`indexOf(str)` returns the index where the first occurrence of `str` begins, or `-1` if `str` does not appear.

WORKED EXAMPLE Searching a sentence

```

index:      0 1 2 3 4 5 6 7 8 9 ...
character:  t h e   c a t   s a ...

```

```

1 String sentence = "the cat sat on the mat";
2 System.out.println(sentence.indexOf("the"));
3 System.out.println(sentence.indexOf("at"));
4 System.out.println(sentence.indexOf("dog"));
5 System.out.println(sentence.indexOf("The"));

```

```

0
5
-1
-1

```

Call	Reasoning	Result
<code>indexOf("the")</code>	the appears twice; the first one starts at index 0	0
<code>indexOf("at")</code>	First at is inside cat: c is 4, a is 5	5
<code>indexOf("dog")</code>	Not in the sentence	-1
<code>indexOf("The")</code>	Uppercase T is a different character from t, so The never appears	-1

For a search string longer than one character, the result is the index of its *first* character. And the search is case-sensitive: uppercase and lowercase letters do not match each other.

Now combine `indexOf` with `substring`. The `@` in an email address separates the user name from the domain, but its position depends on the address. Let `indexOf` find it:

```
1 String email = "ada@school.org";
2 int at = email.indexOf("@");
3 String user = email.substring(0, at);
4 String domain = email.substring(at + 1);
5 System.out.println(at);
6 System.out.println(user);
7 System.out.println(domain);
```

```
3
ada
school.org
```

`at` is 3. `substring(0, at)` takes indices 0 to 2, stopping right before the `@`. `substring(at + 1)` starts one past the `@` and runs to the end. This code works for any address with an `@`, however long the user name is.

WATCH OUT

Feeding -1 straight into `substring`. If the text is not found, `indexOf` returns -1, and -1 is never a valid index:

NoAtSign.java

```
1 public class NoAtSign
2 {
3     public static void main(String[] args)
4     {
5         String email = "ada.school.org";
6         int at = email.indexOf("@");
7         System.out.println(at);
8         String user = email.substring(0, at);
9         System.out.println(user);
10    }
11 }
```

```
-1
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [0, -1)
out of bounds for length 14
    at NoAtSign.main(NoAtSign.java:8)
```

Students forget that "not found" is a possible answer. Line 7 prints -1 normally; line 8 crashes. For now, use `indexOf` results in `substring` only when you know the text is present. In Unit 2 you will learn how a program can check the result first.

PAUSE AND TRY**Question 9** Tracing

Write the exact output.

```
1 String song = "Here Comes the Sun";
2 int spot = song.indexOf("the");
3 System.out.println(spot);
4 System.out.println(song.substring(spot));
5 System.out.println(song.indexOf("sun"));
```

1.15.9 Comparing Strings with equals and compareTo

Programs need to compare text: did the user type the right password? Which of two names comes first alphabetically? Two `String` methods answer these questions. Both are on the Java Quick Reference.

equals: the same characters?

`boolean equals(Object other)` returns `true` if the string has exactly the same sequence of characters as `other`, and `false` otherwise. The parameter type is `Object`, the class every class extends (Topic 1.12), but in this course you will always pass it a `String`. "Exactly the same" means same length, same characters, same order, same capitalization.

WORKED EXAMPLE Checking a password

```
1 String password = "Tiger42";
2 System.out.println(password.equals("Tiger42"));
3 System.out.println(password.equals("tiger42"));
4 System.out.println(password.equals("Tiger42 "));
5 System.out.println(password.equals(new String("Tiger42")));
```

```
true
false
false
true
```

Compared with	Same characters?	Result
"Tiger42"	Yes	true
"tiger42"	No: t vs T	false
"Tiger42 "	No: extra space	false
new String("Tiger42")	Yes	true

The last line compares against a different object created with the constructor. `equals` does not care that they are separate objects; it compares the characters.

compareTo: which comes first?

`int compareTo(String other)` compares two strings by dictionary (alphabetical) order and returns an `int`:

<code>a.compareTo(b)</code> returns	Meaning
a negative number	a comes before b
zero	a and b are equal
a positive number	a comes after b

Only the sign matters. The method promises negative, zero, or positive; do not try to memorize particular numbers.

WORKED EXAMPLE Alphabetical order

```
1 System.out.println("apple".compareTo("banana"));
2 System.out.println("apple".compareTo("cherry"));
3 System.out.println("banana".compareTo("apple"));
4 System.out.println("pear".compareTo("pear"));
5 System.out.println("car".compareTo("cart"));
```

```
-1
-2
1
0
-1
```

Call	Reasoning	Sign
<code>"apple".compareTo("banana")</code>	a comes before b	negative
<code>"apple".compareTo("cherry")</code>	a comes before c (a different negative number)	negative
<code>"banana".compareTo("apple")</code>	Same pair, reversed order of the call	positive
<code>"pear".compareTo("pear")</code>	Same characters	zero
<code>"car".compareTo("cart")</code>	car matches the start of cart; shorter is first, as in a dictionary	negative

KEY IDEA

`a.equals(b)` is true when the two strings have exactly the same characters. `a.compareTo(b)` is negative if a comes first alphabetically, zero if they are equal, and positive if a comes after.

One detail of "alphabetical": Java puts every uppercase letter before every lowercase letter, so "Zoo" comes before "apple":

```
1 System.out.println("Zoo".compareTo("apple"));
2 System.out.println("zoo".compareTo("apple"));
```

```
-7
25
```

With all-lowercase (or all-uppercase) words, Java's order matches the dictionary.

WATCH OUT

Reading compareTo backwards. `a.compareTo(b)` describes where `a` is relative to `b`. A negative result means `a` is first. Students sometimes read negative as "`a` is less important" or flip which string is which. Say it out loud as a sentence: "apple compared to banana is negative, so apple comes before banana."

PAUSE AND TRY

Question 10 Multiple choice

Which call returns a negative number?

- A. `"dog".compareTo("cat")`
- B. `"cat".compareTo("cats")`
- C. `"cat".compareTo("cat")`
- D. `"cow".compareTo("cat")`

Question 11 Multiple choice

Given `String answer = "Paris";`, which line prints `true`?

- A. `System.out.println(answer.equals("paris"));`
- B. `System.out.println(answer.equals("Paris "));`
- C. `System.out.println(answer.equals("Par" + "is"));`
- D. `System.out.println(answer.equals("PARIS"));`

1.15.10 Concatenating Other Objects

In 1.15.3 you saw that `+` turns a number into text. What happens if the thing on the other side of `+` is an object that is not a `String`, such as a `Scanner`? Java still needs text to concatenate, so it needs some way to ask any object for a text version of itself.

Topic 1.12 told you that every class in Java is a subclass of the `Object` class. `Object` has a method `String toString()` that returns a string representing the object. Because every class is a subclass of `Object`, **every object has a `toString` method**. So when you concatenate a `String` with any object, Java calls that object's `toString` method for you and joins the text it returns.

The `toString` that `Object` provides is very general. Many classes replace it with their own version that describes their objects better. When a subclass has a public method with the same signature as a public method in its superclass but its own behavior, that is called **method overriding**. `String` overrides `toString` to return the string's own characters, and `Scanner` overrides it to return a description of the scanner. (Writing your own `toString` is not part of this course; you only need to know that Java calls it.)

KEY IDEA

Concatenating a String with any object calls that object's `toString` method automatically. Every object has one, because every class is a subclass of `Object`.

WORKED EXAMPLE Concatenating a Scanner

The text a `Scanner` returns from `toString` is long, so this program prints only parts of it, using `substring` and `indexOf`.

ScannerText.java

```
1 import java.util.Scanner;
2
3 public class ScannerText
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner(System.in);
8         String note = "Input: " + input;
9         String described = input.toString();
10        System.out.println(note.substring(0, 24));
11        System.out.println(described.substring(0, 17));
12        System.out.println(note.indexOf("["));
13    }
14 }
```

```
Input: java.util.Scanner
java.util.Scanner
24
```

Line	What happens
8	<code>input</code> is not a String, so Java calls <code>input.toString()</code> and concatenates "Input: " with the text it returns
9	Calls <code>toString</code> directly; same text
10	The first 24 characters of <code>note</code> : the 7 characters of "Input: " plus <code>java.util.Scanner</code> , the class's full name (17 characters)
11	The first 17 characters of the text from <code>toString</code>
12	The description continues with a <code>[</code> right after the class name, at index 24

`Scanner`'s version of `toString` starts with the class's full name and then lists settings inside brackets. It does not contain anything a user typed.

Now vary the object. A `String` made with the constructor is also an object, and its `toString` returns its own characters:

```
1 String city = new String("Denver");
2 String label = "City: " + city;
3 System.out.println(label);
4 System.out.println(city.toString());
```

```
City: Denver
Denver
```

That is why concatenating two strings works the way you expect: the text `String`'s `toString` returns is just the string itself.

WATCH OUT

Thinking `"Input: " + input` **reads what the user typed.** Concatenating a `Scanner` does not read any input. It only calls `toString`, which describes the `Scanner` object. To get what the user typed, you still call a reading method such as `input.nextLine()`, as in Topic 1.4, and concatenate the `String` it returns.

PAUSE AND TRY

Question 12 Multiple choice

Why can Java concatenate a `String` with an object of any class?

- A. Java converts every object to the characters the user typed into it.
- B. Every class is a subclass of `Object`, which has a `toString` method, so Java can call the object's `toString` to get text.
- C. Only `String` and `Scanner` objects can be concatenated; other objects cause a compile error.
- D. Java copies the object's memory address into the string and never calls a method.

Putting it together

A school's sign-up page builds a username from a full name: the first letter of the first name, then the whole last name, then the number of characters in the full name. Here is the code, using `indexOf`, `substring`, `concatenation`, `length()`, and `compareTo`:

```
1 String fullName = "Grace Hopper";
2 int space = fullName.indexOf(" ");
3 String first = fullName.substring(0, space);
4 String last = fullName.substring(space + 1);
5 String username = first.substring(0, 1) + last + fullName.length();
6 System.out.println(username);
7 System.out.println(first.compareTo(last));
8 System.out.println(fullName);
```

Index diagram for `"Grace Hopper"` (length 12):

```
index:      0 1 2 3 4 5 6 7 8 9 10 11
character:  G r a c e   H o p p e r
```

Line	Code	Value produced	Why
------	------	----------------	-----

2	<code>fullName.indexOf(" ")</code>	<code>space = 5</code>	The space is the first (and only) one
3	<code>fullName.substring(0, space)</code>	<code>first = "Grace"</code>	Indices 0–4, stopping before the space
4	<code>fullName.substring(space + 1)</code>	<code>last = "Hopper"</code>	From index 6 to the end
5	<code>first.substring(0, 1)</code>	<code>"G"</code>	Single-character pattern: index 0 to 0 + 1
5	<code>"G" + last + fullName.length()</code>	<code>username = "GHopper12"</code>	Left to right: "GHopper", then the int 12 becomes text
7	<code>first.compareTo(last)</code>	<code>negative</code>	G comes before H
8	<code>fullName</code>	<code>"Grace Hopper"</code>	Immutable: no call changed it

```
GHopper12
-1
Grace Hopper
```

Every method call returned a new value and left `fullName` alone, which is why line 8 still prints the full name. If the name had no space, `indexOf` would return `-1` and line 3 would throw a `StringIndexOutOfBoundsException`, as in 1.15.8.

Common mistakes

The Watch Out boxes explain each mistake. This table is a quick lookup for every `String` method in this chapter: what it returns and when it fails.

Method header	Returns	Exception or common slip
<code>int length()</code>	Number of characters	The last index is <code>length() - 1</code> , not <code>length()</code>
<code>String substring(int from, int to)</code>	Characters at <code>from</code> through <code>to - 1</code>	Throws if an index is negative or above <code>length()</code> , or if <code>from</code> is greater than <code>to</code> ; the character at <code>to</code> is left out
<code>String substring(int from)</code>	Characters from <code>from</code> to the end	Throws if <code>from</code> is below 0 or above <code>length()</code> ; <code>substring(length())</code> is <code>""</code>
<code>int indexOf(String str)</code>	Index of first occurrence, or <code>-1</code>	Case-sensitive; <code>-1</code> passed to <code>substring</code> throws
<code>boolean equals(Object other)</code>	true if exactly the same characters	Capitalization and spaces count
<code>int compareTo(String other)</code>	Negative, zero, or positive	Negative means the calling string comes first; uppercase letters come before lowercase

+ and += with a String	A new String	Left to right: "Total: " + 3 + 4 is "Total: 34"; no spaces added
------------------------	--------------	--

And for every method: the original string never changes. Store the result if you want to keep it.

Vocabulary

Term	Definition	Example
String	An object that holds a sequence of characters	"Falcons"
sequence of characters	Characters (letters, digits, spaces, punctuation) in a fixed order	Number 23 is 9 characters
string literal	Text in double quotes; creates a String object	"Denver"
constructor	Called with new to create an object	new String("Denver")
package / java.lang	A named group of classes / the package whose classes, including String, need no import	Scanner needs an import; String does not
concatenation	Joining strings end to end into a new String with + or +=	"Ada" + " " + "LoveLace"
implicit conversion	Java turning a primitive value into text on its own during concatenation	"Score: " + 95
index (plural indices)	The position number of a character, starting at 0	In "planet", p is at index 0
length	The number of characters in a string, from length()	"planet".length() is 6
empty string	A String with no characters, length 0	""
substring	A part of a string; also the method that returns one	"sunflower".substring(0, 3) is "sun"
Java Quick Reference	The sheet of method headers available during the AP exam	Lists substring, indexOf, and others
StringIndexOutOfBoundsException	The run-time error thrown when a String method is given an index outside its allowed range	"planet".substring(0, 7)
immutable	Cannot be changed after it is created; String methods return new strings instead	word.substring(0, 3); leaves word unchanged

toString	The method every object has that returns a string representing the object	"Input: " + input calls <code>input.toString()</code>
method overriding	A subclass giving its own version of a superclass's public method, with the same signature	Scanner overrides <code>toString</code>

Chapter summary

- A **String** is an immutable object holding a sequence of characters. Create one with a literal or the `String` constructor; `String` is in `java.lang`, so no import is needed.
- `+` and `+=` **concatenate** into a new `String`. A primitive on the other side is converted to text; any other object is converted by calling its `toString` method. Java works left to right, so the order of the pieces matters.
- Characters are numbered by **index** from 0 to `length() - 1`.
- `substring(from, to)` returns indices from `to` to `to - 1`; `substring(from)` goes to the end; `substring(i, i + 1)` is the single character at `i`. `indexOf(str)` finds where `str` first starts, or returns `-1`.
- An index outside the allowed range throws a **StringIndexOutOfBoundsException** while the program runs.
- `equals` tells whether two strings have exactly the same characters; `compareTo` returns a negative, zero, or positive number to show alphabetical order.

Review questions

Question 13 Multiple choice

What does this print?

```
1 System.out.println("A" + 2 * 3 + 1);
```

- A. A231
- B. A61
- C. A7
- D. A2*31

Question 14 Tracing

Write the exact output.

```
1 String date = "2026-10-07";
2 int dash = date.indexOf("-");
3 String year = date.substring(0, dash);
4 String rest = date.substring(dash + 1);
5 String month = rest.substring(0, rest.indexOf("-"));
6 System.out.println(year);
7 System.out.println(rest);
8 System.out.println(month);
```

Question 15 Debugging

This code should print `Last letter: t`, but it compiles and then throws a `StringIndexOutOfBoundsException`. Explain why and write the corrected line.

```
1 String word = "rocket";
2 String last = word.substring(word.length(), word.length() + 1);
3 System.out.println("Last letter: " + last);
```

Question 16 Multiple choice

What does this print?

```
1 String a = "sun";
2 String b = a;
3 a = a + "flower";
4 b.substring(1);
5 System.out.println(a + " " + b);
```

- A. sunflower sunflower
- B. sunflower un
- C. sunflower sun
- D. sun sun

Question 17 Tracing

This program compiles. Write everything it prints before it stops, and name the exception it throws and the line that throws it.

Login.java

```
1 public class Login
2 {
3     public static void main(String[] args)
4     {
5         String entry = "maya.lee@school";
6         System.out.println("Checking " + entry);
7         int at = entry.indexOf("@");
8         System.out.println("Name: " + entry.substring(0, at));
9         int dot = entry.indexOf(".org");
10        System.out.println("Site: " + entry.substring(at + 1, dot));
11        System.out.println("Welcome!");
12    }
13 }
```

Question 18 Tracing

Write the exact output. For the two `compareTo` lines, it is enough to say whether the value is negative, zero, or positive.

```
1 String first = "Mario";
2 String second = "Maria";
3 System.out.println(first.compareTo(second));
4 System.out.println(first.substring(0, 4).equals(second.substring(0, 4)));
5 System.out.println(second.equals("maria"));
6 System.out.println(second.compareTo(first));
```

Question 19 Tracing

Write the exact output.

```
1 String code = "";
2 code += 4;
3 code += 2;
4 code = 1 + code;
5 System.out.println(code + 1);
6 System.out.println(code.length() + 1);
```

Question 20 Short answer

Look at this program.

ReaderReport.java

```
1 import java.util.Scanner;
2
3 public class ReaderReport
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner(System.in);
8         String report = "Reader: " + reader;
9         System.out.println(report.indexOf("Scanner"));
10        System.out.println(report.substring(0, 12));
11    }
12 }
```

(a) It prints 18 and then Reader: java. Explain where the text java comes from, naming the method Java calls. (b) Explain why the program needs `import java.util.Scanner;` but no import for `String`.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

☐ I can create a `String` with a literal or the constructor, and explain why `String` needs no import.

Reread: 1.15.1 Creating String Objects

☐ I can predict the result of joining strings with `+` and `+=`. Reread: 1.15.2 Concatenating Strings

☐ I can predict when `+` adds and when it concatenates, working left to right. Reread: 1.15.3 Concatenating Numbers and Booleans

☐ I can find the index of any character and the length of a string. Reread: 1.15.4 Indices and the length Method

☐ I can predict the result of substring with one or two arguments, including single characters.

Reread: 1.15.5 Extracting Text with substring

☐ I can tell which substring calls throw a `StringIndexOutOfBoundsException`. Reread: 1.15.6 Out-of-Range Indices

☐ I can explain why calling a method on a `String` never changes it. Reread: 1.15.7 Strings Are Immutable

☐ I can predict what `indexOf` returns, including `-1`. Reread: 1.15.8 Finding Text with `indexOf`

- ☐ I can predict the result of equals and the sign of compareTo. Reread: 1.15.9 Comparing Strings with equals and compareTo
- ☐ I can explain what happens when a String is concatenated with any object. Reread: 1.15.10 Concatenating Other Objects

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. Classes in the `java.lang` package, including `String`, are available by default, so no import is needed.

- **A** is wrong: that is true for `Scanner` (in `java.util`), not for `String`. This choice mixes up the two packages.
- **C** is wrong: a string literal creates a `String` object just as well as the constructor does. Both are allowed.
- **D** is wrong: nothing about this line is impossible at run time; it just creates a `String` object.

Reread *1.15.1 Creating String Objects* if you chose differently.

Question 2

```
Rock and Jazz
Pop
```

After line	playlist	genre
2	"Rock"	"Jazz"
3	"Rock and "	"Jazz"
4	"Rock and Jazz"	"Jazz"
5	"Rock and Jazz"	"Pop"

Line 4 copies the characters of `genre` into a new string at that moment. Changing `genre` on line 5 makes `genre` refer to a different string, but it cannot reach back into `playlist`, which already holds `Rock and Jazz`. Notice the spaces in `" "` and `" "` on line 3; without them the result would be `RockandJazz`.

Reread *1.15.2 Concatenating Strings* if your output differed.

Question 3

```
Points: 23
5 points
3345
```

- Line 3: `"Points: " + goals` is a string, so `+ assists` concatenates: `Points: 2`, then `Points: 23`.
- Line 4: `goals + assists` is two ints, so it adds to 5; then `5 + " points"` concatenates.
- Line 5: `1 + 2` adds to 3. `3 + "3"` concatenates to `"33"`. From then on the left side is a string, so `+ 4` gives `"334"` and `+ 5` gives `"3345"`.

Reread *1.15.3 Concatenating Numbers and Booleans* if your output differed.

Question 4

B. The characters are B, e, space, k, i, n, d, period: 8 in all. Counting from 0, B is 0, e is 1, the space is 2, and k is 3.

- **A** is wrong: it skips counting the space (giving 7) and counts positions from 1 (giving 4 for `k`).
- **C** is wrong: the length is 8, but the last index is always one less, 7.
- **D** is wrong: every character counts, including spaces and punctuation.

Reread 1.15.4 *Indices and the length Method* if you chose differently.

Question 5

```
index:      0 1 2 3 4 5 6 7 8 9
character:  b a s k e t b a l l
```

```
ball
basket
s
tbb
```

- `substring(6)`: index 6 to the end, "ball".
- `substring(0, 6)`: indices 0 to 5, "basket".
- `substring(2, 3)`: just index 2, "s".
- `substring(5, 7)` is indices 5 and 6, "tb". `substring(0, 1)` is "b". Concatenated: "tbb".

Reread 1.15.5 *Extracting Text with substring* if your output differed.

Question 6

Error: the `to` argument is 2, so only indices 0 and 1 are taken. The character at index `to` is never included.

Corrected line:

```
1 String shortName = day.substring(0, 3);
```

Three characters are wanted, and `3 - 0` is 3: indices 0, 1, and 2, which are `s`, `a`, `t`. The original compiled and ran without complaint, so this was a logic error.

Reread 1.15.5 *Extracting Text with substring* if you missed it.

Question 7

D. "AP2026" has length 6, so `to` can be at most 6. Asking to stop before index 7 would need a character at index 6, which does not exist.

- **A** is wrong: `from` may equal the length; the result is the empty string "".
- **B** is wrong: `to` may equal the length, because the character at `to` is not taken. The result is the whole string, "AP2026".
- **C** is wrong: `from` equal to `to` is legal and returns "".

Reread 1.15.6 *Out-of-Range Indices* if you chose differently.

Question 8

```
Practice daily
Practice
```

After line	motto refers to	copy refers to
1	"Practice daily"	

2	"Practice daily"	"Practice daily"
3	"Practice daily"	"Practice daily"
4	"Practice daily"	"Practice"

Line 3 builds "daily" but throws it away, because the result is not stored. Line 4 stores a new string, "Practice" (indices 0 to 7), in `copy`. That moves `copy` to a new object; it does not change the object `motto` refers to. Strings are immutable, so `motto` still prints `Practice daily`.

Reread 1.15.7 *Strings Are Immutable* if your output differed.

Question 9

```
11
the Sun
-1
```

Counting from 0: Here is 0–3, space 4, Comes is 5–9, space 10, so `the` starts at index 11. `substring(11)` runs from there to the end: `the Sun`. The last search is for `sun` with a lowercase `s`; the string contains only `Sun`, and the search is case-sensitive, so the result is -1.

Reread 1.15.8 *Finding Text with indexOf* if your output differed.

Question 10

B. `cat` matches the beginning of `cats` and is shorter, so it comes first. The string the method is called on comes first, so the result is negative.

- **A** is wrong: `dog` comes after `cat` (`d` is after `c`), so the result is positive.
- **C** is wrong: identical strings give zero.
- **D** is wrong: both start with `c`, but the second letters are `o` and `a`, and `o` comes after `a`, so `cow` is after `cat`: positive.

Reread 1.15.9 *Comparing Strings with equals and compareTo* if you chose differently.

Question 11

C. `"Par" + "is"` is concatenated first, producing a string with the characters `Paris`, exactly matching `answer`. `equals` compares characters, so it returns `true`.

- **A** is wrong: lowercase `p` is a different character from `P`.
- **B** is wrong: the trailing space is a sixth character, so the strings differ.
- **D** is wrong: capitalization differs in four letters.

Reread 1.15.9 *Comparing Strings with equals and compareTo* if you chose differently.

Question 12

B. Because every class extends `Object`, every object has a `toString` method (perhaps overridden with a class-specific version). Concatenation calls it automatically.

- **A** is wrong: concatenation does not read input. A `Scanner`'s `text` describes the scanner, not what was typed.
- **C** is wrong: any object can be concatenated, because every object has `toString`.
- **D** is wrong: Java does call a method, `toString`. Classes like `String` and `Scanner` override it to return their own text.

Reread 1.15.10 *Concatenating Other Objects* if you chose differently.

Question 13

B. `*` has higher precedence than `+`, so `2 * 3` is computed first, giving the `int` 6. Then the `+` operators run left to right: `"A" + 6` is `"A6"`, and `"A6" + 1` is `"A61"`.

- **A** is wrong: it treats `*` as if it ran after concatenation. Precedence from Topic 1.3 still applies inside a concatenation.
- **C** is wrong: it adds `6 + 1` first, but the left `+` runs first and already produces a `String`, so `+` concatenates.
- **D** is wrong: `*` is an operator, not text; only characters inside quotes become part of the string.

Reread 1.15.3 *Concatenating Numbers and Booleans* if you chose differently.

Question 14

```
index:      0 1 2 3 4 5 6 7 8 9
character:  2 0 2 6 - 1 0 - 0 7
```

```
2026
10-07
10
```

Line	Value
2	<code>dash = 4</code> (the first <code>-</code>)
3	<code>year = indices 0-3 = "2026"</code>
4	<code>rest = from index 5 = "10-07"</code>
5	<code>rest.indexOf("-")</code> is 2 (counting in <code>rest</code> , not date), so <code>month = rest.substring(0, 2) = "10"</code>

The key step is line 5: `indexOf` counts positions in the string it is called on. In `rest`, the dash is at index 2. Also, `year` holds the characters `2026`, not a number.

Reread 1.15.8 *Finding Text with indexOf* and 1.15.5 *Extracting Text with substring* if your output differed.

Question 15

Why: `word.length()` is 6, so line 2 asks for `substring(6, 7)`, the character at index 6. `"rocket"` has indices 0 to 5, so `to = 7` is beyond the length and Java throws `StringIndexOutOfBoundsException`. Nothing is printed, because line 3 never runs.

Corrected line:

```
1 String last = word.substring(word.length() - 1, word.length());
```

This is `substring(5, 6)`, the single-character pattern for index 5, which is `t`.
(`word.substring(word.length() - 1)` gives the same result.)

Reread 1.15.6 *Out-of-Range Indices* if you missed the error.

Question 16

C. After line 2, `a` and `b` refer to the same `"sun"` object. Line 3 creates a new string `"sunflower"` and moves only `a` to it; the `"sun"` object is immutable and unchanged, so `b` still holds `sun`. Line 4 builds `"un"` but does not store it, so `b` is still `sun`.

- **A** is wrong: it assumes that changing `a` changes the shared object. Concatenation makes a new object instead.
- **B** is wrong: `b.substring(1)` returns `"un"` but does not change `b`, and the result was not stored.
- **D** is wrong: line 3 stores the new string in `a`, so `a` does change what it refers to.

Reread *1.15.7 Strings Are Immutable* if you chose differently.

Question 17

```
Checking maya.lee@school
Name: maya.lee
```

Then line 10 throws `java.lang.StringIndexOutOfBoundsException`, and `Welcome!` is never printed.

Line	Value
6	Prints <code>Checking maya.lee@school</code>
7	<code>at = 8</code> (<code>maya.lee</code> is 8 characters: indices 0–7)
8	<code>substring(0, 8) = "maya.lee"</code> ; prints <code>Name: maya.lee</code>
9	<code>.org</code> does not appear, so <code>dot = -1</code>
10	<code>substring(9, -1)</code> : <code>-1</code> is a negative index, and <code>from</code> is greater than <code>to</code> . Exception

Note that the `.` in `maya.lee` does not match: the search is for the four characters `.org` together. The program compiled because the compiler checks types, not the values `indexOf` will return.

Reread *1.15.8 Finding Text with indexOf* and *1.15.6 Out-of-Range Indices* if your answer differed.

Question 18

Line	Expression	Result
3	<code>"Mario".compareTo("Maria")</code>	Positive: the first difference is <code>o</code> vs <code>a</code> , and <code>o</code> comes after <code>a</code>
4	<code>"Mari".equals("Mari")</code> (both substrings are indices 0–3)	<code>true</code>
5	<code>"Maria".equals("maria")</code>	<code>false</code> : <code>M</code> and <code>m</code> are different characters
6	<code>"Maria".compareTo("Mario")</code>	Negative: the same pair in the other order

The exact output is:

```
14
true
false
-14
```

The specific numbers (14 and -14) are not something you need to predict; their signs are. Swapping the two strings in `compareTo` flips the sign.

Reread *1.15.9 Comparing Strings with equals and compareTo* if your answer differed.

Question 19

```
1421
4
```

After line	code	Why
1	<code>""</code>	The empty string, length 0
2	<code>"4"</code>	<code>"" + 4</code> concatenates; the <code>int 4</code> becomes the text <code>4</code>
3	<code>"42"</code>	<code>"4" + 2</code> concatenates
4	<code>"142"</code>	<code>1 + "42"</code> : one side is a <code>String</code> , so the <code>int 1</code> becomes text and goes in front

Line 5 prints `code + 1`, which concatenates: `1421`. Line 6 is different: `code.length()` is the `int 3`, and `3 + 1` is two `ints`, so it adds: `4`. Starting from the empty string is a common way to make sure `+=` concatenates even when only numbers are added.

Reread *1.15.3 Concatenating Numbers and Booleans* and *1.15.4 Indices and the length Method* if your output differed.

Question 20

Model answer:

(a) `reader` is a `Scanner`, not a `String`, so concatenating it with `"Reader: "` makes Java call `reader.toString()`. `Scanner` overrides `toString` to return a description that starts with its full class name, `java.util.Scanner`. `"Reader: "` is 8 characters (indices 0–7), and `java.util.` is 10 more, so `Scanner` starts at index 18; the first 12 characters are `Reader: java.`

(b) `Scanner` is in the `java.util` package, which must be imported. `String` is in `java.lang`, whose classes are available to every program by default.

A complete answer must:

- say that concatenation calls the object's `toString` method, and that the text describes the `Scanner` rather than anything typed;
- say that `String` is in `java.lang`, which needs no import, while `Scanner` is in `java.util`.

Reread *1.15.10 Concatenating Other Objects* and *1.15.1 Creating String Objects* if your answer differed.