

2.10 Implementing String Algorithms

Self-Study Guide · About 50 minutes · With answer key

By the end you will be able to

- Use a loop and substring to visit every character of a string, one at a time, without going past the end.
- Use a boolean flag to decide whether any character, or every character, of a string has a property.
- Examine substrings longer than one character and choose loop bounds that reach every one of them.
- Count how many characters or substrings of a string meet a condition, including overlapping matches.
- Build a new string piece by piece, including a string with the characters in reverse order.
- Trace string algorithms exactly and find off-by-one and setup mistakes in them.

What to remember from earlier

- A string's characters sit at indices 0 through `length()` minus 1. ([Topic 1.15: String Manipulation, 1.15.4 Indices and the length Method](#))
- `substring(i, i + 1)` returns the one-character string at index `i`. ([Topic 1.15: String Manipulation, 1.15.5 Extracting Text with substring](#))
- An index past the end of a string throws a `StringIndexOutOfBoundsException`. ([Topic 1.15: String Manipulation, 1.15.6 Out-of-Range Indices](#))
- Strings are immutable, so `+=` builds a new string and stores it in the variable. ([Topic 1.15: String Manipulation, 1.15.7 Strings Are Immutable](#))
- A counter set to 0 before a loop and increased inside it counts how often something happens. ([Topic 2.9: Implementing Selection and Iteration Algorithms, 2.9.3 Counting Values That Meet a Condition](#))

2.10.1 Visiting Every Character

A website wants to know whether a new password contains a digit. A spelling game wants to count the vowels in a word. A puzzle app wants to know whether a phrase reads the same backward. In Topic 1.15 you could pull out *one* character with `substring`, but you had to know its index ahead of time, and a password could be 6 characters or 60. These questions need a program that looks at **every** character, however long the string is.

That is a job for a loop. The idea in plain words: start at the first position, look at the character there, move one position to the right, and repeat until you run out of characters. Going through a string one position at a time like this is called a **traversal** of the string.

Two facts from Topic 1.15 make this work:

- The indices of a string run from 0 to `length() - 1`.
- `word.substring(i, i + 1)` is the single character at index `i`, as a one-character `String`.

So a `for` loop from Topic 2.8 whose loop control variable `i` takes the values 0, 1, 2, ..., `length() - 1` can use `i` as the index.

```
1 for (int i = 0; i < word.length(); i++)
2 {
3     String letter = word.substring(i, i + 1);
4     // work with letter here
5 }
```

Take the header apart one piece at a time:

Piece	What it does
<code>int i = 0</code>	Starts at index 0, the first character
<code>i < word.length()</code>	Keeps going while <code>i</code> is a real index; the last value that passes is <code>length() - 1</code>
<code>i++</code>	Moves one position to the right after each pass through the body
<code>substring(i, i + 1)</code>	Gets the character at index <code>i</code> . It is a <code>String</code> of length 1, so you compare it with equals, not <code>==</code>

KEY IDEA

To visit every character, loop `i` from 0 while `i < word.length()`, and use `word.substring(i, i + 1)` to get the character at index `i`.

WORKED EXAMPLE Printing each character with its index

Watch the value of `i` and which character line 4 pulls out on each pass.

```
1 String word = "otter";
2 for (int i = 0; i < word.length(); i++)
3 {
4     String letter = word.substring(i, i + 1);
5     System.out.println(i + " " + letter);
6 }
```

```
index:    0 1 2 3 4
character: o t t e r
```

i	i < 5?	word.substring(i, i + 1)	Printed
0	true	"o"	0 o
1	true	"t"	1 t
2	true	"t"	2 t
3	true	"e"	3 e
4	true	"r"	4 r
5	false	(loop ends)	

```
0 o
1 t
2 t
3 e
4 r
```

`word.length()` is 5, so the condition is `i < 5`. When `i` becomes 5 the condition is false and the body never runs with `i = 5`. That matters, because index 5 does not exist in "otter".

Now change one thing: the update. With `i += 2` instead of `i++`, the loop jumps two positions each time, so it visits only the characters at indices 0, 2, 4, and so on.

```
1 String word = "computer";
2 for (int i = 0; i < word.length(); i += 2)
3 {
4     System.out.println(word.substring(i, i + 1));
5 }
```

```
index:      0 1 2 3 4 5 6 7
character:  c o m p u t e r
```

`i` takes the values 0, 2, 4, and 6, so the loop prints `c`, `m`, `u`, and `e`, one per line. After 6 comes 8, and `8 < 8` is false. The condition `i < word.length()` still keeps every `substring(i, i + 1)` call safe, because `i` is never allowed to reach the length. The loop header decides **which** positions get visited; the body decides **what happens** at each one.

WATCH OUT

Using `<=` instead of `<`. Because the length is 5, it feels natural to let `i` go "up to" 5:

```
1 String word = "otter";
2 for (int i = 0; i <= word.length(); i++)
3 {
4     System.out.println(word.substring(i, i + 1));
5 }
```

The first five passes print `o`, `t`, `t`, `e`, and `r`. Then `i` becomes 5, `5 <= 5` is true, and line 4 asks for `word.substring(5, 6)`. There is no index 5, so the program crashes with an exception like this (the exact wording can vary between Java versions):

```
java.lang.StringIndexOutOfBoundsException: Range [5, 6) out of bounds for length 5
```

This is an off-by-one error from Topic 2.7: the loop runs one time too many. It happens because the **length** counts characters starting from 1, while **indices** start from 0, so the last index is always one less than the length. The fix is `i < word.length()`.

PAUSE AND TRY**Question 1** Tracing

Write the exact output.

```
1 String word = "planet";
2 for (int i = 1; i < word.length(); i += 2)
3 {
4     System.out.print(word.substring(i, i + 1));
5 }
6 System.out.println();
```

Question 2 Multiple choice

The loop body is `String letter = city.substring(i, i + 1);`. Which header makes the loop visit every character of `city` exactly once without an exception?

- A. `for (int i = 0; i <= city.length(); i++)`
- B. `for (int i = 1; i < city.length(); i++)`
- C. `for (int i = 0; i < city.length(); i++)`
- D. `for (int i = 0; i < city.length() - 1; i++)`

2.10.2 Checking a Property with a Flag

Back to the password: does it contain at least one digit? Visiting every character is not enough by itself. The program also has to **remember** whether it has seen a digit yet, and it can only give an answer after the loop has looked at all of the characters.

Think of a lifeguard scanning a beach for a red umbrella. Before they start, the answer is "not seen yet." The moment they spot one, they raise a flag. They might keep scanning, but the flag stays up, and at the end the flag tells you the answer. In code, that flag is a `boolean` variable. A **flag** is a `boolean` variable that starts with one value before a loop and is switched to the other value inside the loop when something you are watching for happens.

How does the program test whether a character is a digit? With a trick that uses `indexOf` from Topic 1.15. Put all the allowed characters in one string, `"0123456789"`, and ask where the character appears in it. If the character is a digit, `indexOf` returns its position, a number 0 or more. If it is not a digit, `indexOf` returns -1.

Expression	Result	Meaning
<code>"0123456789".indexOf("7")</code>	7	is a digit
<code>"0123456789".indexOf("x")</code>	-1	not a digit
<code>"aeiou".indexOf("e")</code>	1	is a vowel
<code>"aeiou".indexOf("t")</code>	-1	not a vowel

KEY IDEA

To check whether any character has a property: set a flag to false before the loop, set it to true inside the loop when a character has the property, and read the flag only after the loop ends.

WORKED EXAMPLE Does the password contain a digit?

Line 3 sets the flag before the loop. Look for the only line that ever changes it.

```

1 String password = "tiger7x";
2 String digits = "0123456789";
3 boolean hasDigit = false;
4 for (int i = 0; i < password.length(); i++)
5 {
6     String letter = password.substring(i, i + 1);
7     if (digits.indexOf(letter) >= 0)
8     {
9         hasDigit = true;
10    }
11 }
12 System.out.println("Has a digit: " + hasDigit);

```

```

index:      0 1 2 3 4 5 6
character:  t i g e r 7 x

```

i	letter	digits.indexOf(letter)	hasDigit after the pass
0	"t"	-1	false
1	"i"	-1	false
2	"g"	-1	false
3	"e"	-1	false
4	"r"	-1	false
5	"7"	7	true
6	"x"	-1	true

```
Has a digit: true
```

On pass 5 the `if` condition is true, so line 9 raises the flag. On pass 6 the condition is false, and since there is no `else`, nothing touches `hasDigit`. It stays true. The print statement is after the loop, because only after every character has been checked can the program be sure of the answer.

Now flip the question: is **every** character a digit, as a PIN must be? The flag starts true ("no problem found yet") and switches to false the moment a character fails. One bad character is enough to make the answer false, just as one digit was enough to make the first answer true.

```

1 String pin = "40x2";
2 String digits = "0123456789";
3 boolean allDigits = true;
4 for (int i = 0; i < pin.length(); i++)
5 {
6     String letter = pin.substring(i, i + 1);
7     if (digits.indexOf(letter) == -1)
8     {
9         allDigits = false;
10    }
11 }
12 System.out.println("All digits: " + allDigits);

```

For "40x2", the characters 4 and 0 leave the flag alone. At x, `indexOf` returns -1, the condition `== -1` is true, and `allDigits` becomes false. The 2 does not change it back. The output is All digits: false.

Question	Flag starts	Switch it when a character...	Changed back?
Does any character have it?	false	has the property	Never
Does every character have it?	true	does not have the property	Never

WATCH OUT

Adding an else that resets the flag. It looks tidy to write "if it is a digit, true; otherwise, false":

```

1 String password = "tiger7x";
2 String digits = "0123456789";
3 boolean hasDigit = false;
4 for (int i = 0; i < password.length(); i++)
5 {
6     String letter = password.substring(i, i + 1);
7     if (digits.indexOf(letter) >= 0)
8     {
9         hasDigit = true;
10    }
11    else
12    {
13        hasDigit = false;
14    }
15 }
16 System.out.println("Has a digit: " + hasDigit);

```

This prints Has a digit: false, even though "tiger7x" has a 7. The flag does become true at index 5, but the very next pass sees x, runs the `else`, and sets it back to false. With an `else` like this, the flag only describes the **last** character checked, not the whole string. Students write it because each single pass seems to deserve its own answer. The fix is to delete the `else`: once the flag has been switched, nothing should switch it back.

PAUSE AND TRY**Question 3** Multiple choice

What does this code print, and what question is it answering?

```
1 String name = "ice cream";
2 boolean noSpaces = true;
3 for (int i = 0; i < name.length(); i++)
4 {
5     if (name.substring(i, i + 1).equals(" "))
6     {
7         noSpaces = false;
8     }
9 }
10 System.out.println(noSpaces);
```

- A. `true`; it checks whether the string contains a space
- B. `false`; it checks whether the string has no spaces at all
- C. `true`; it checks whether the string has no spaces at all
- D. `false`; it checks whether the last character is a space

Question 4 Short answer

In the PIN example, a student changes line 3 to `boolean allDigits = false;` and runs it with the PIN "4072". What is printed, and why does the flag have to start as `true` for this question?

2.10.3 Looking at Longer Substrings

Some properties are not about one character. Does a word contain a double letter, like the `ll` in `balloon`? Does a sentence contain the word `cat` somewhere? For a fixed piece of text like `"cat"`, `indexOf` from Topic 1.15 can answer, but it cannot answer "are any two neighboring characters the same?", because there is no single text to search for. A loop can.

The plan is the same traversal as before, but at each position `i` the program looks at more than one character: `word.substring(i, i + 2)` is the two characters starting at `i`, `word.substring(i, i + 3)` is three, and in general `word.substring(i, i + n)` is the `n` characters starting at index `i`.

That changes where the loop must stop. The **end** of each substring, `i + n`, may be at most `word.length()`. So the last starting index is `word.length() - n`.

```
index:      0 1 2 3 4 5 6
character:  b a l l o o n
two-character pieces start at 0 through 5:  ba al ll lo oo on
```

For "balloon", the length is 7, so the last two-character piece starts at $7 - 2 = 5$. Either of these conditions stops the loop in the right place; they mean the same thing for whole numbers:

```
1 i <= word.length() - n
2 i < word.length() - n + 1
```

For two characters that is `i <= word.length() - 2`, which is the same as `i < word.length() - 1`.

KEY IDEA

To look at every substring of length `n`, loop `i` from 0 while `i <= word.length() - n`, and use `word.substring(i, i + n)`.

WORKED EXAMPLE Finding double letters

Each pass compares the character at `i` with the one right after it, at `i + 1`. Notice the loop condition on line 3.

```

1 String word = "balloon";
2 boolean hasDouble = false;
3 for (int i = 0; i < word.length() - 1; i++)
4 {
5     String first = word.substring(i, i + 1);
6     String second = word.substring(i + 1, i + 2);
7     if (first.equals(second))
8     {
9         hasDouble = true;
10        System.out.println("Double at index " + i);
11    }
12 }
13 System.out.println("Has a double letter: " + hasDouble);

```

i	first	second	Equal?	Printed
0	"b"	"a"	no	
1	"a"	"l"	no	
2	"l"	"l"	yes	Double at index 2
3	"l"	"o"	no	
4	"o"	"o"	yes	Double at index 4
5	"o"	"n"	no	

```

Double at index 2
Double at index 4
Has a double letter: true

```

The condition `i < word.length() - 1` is `i < 6`, so the last pass has `i = 5` and looks at indices 5 and 6. A sixth pass with `i = 6` would need index 7 for `second`, which does not exist. The flag works exactly as in 2.10.2; this loop also prints where each double letter starts.

Now vary the length. To look for "cat", every piece is three characters, so the loop stops at `length() - 3`:

```

1 String text = "concatenate";
2 for (int i = 0; i <= text.length() - 3; i++)
3 {
4     String piece = text.substring(i, i + 3);
5     if (piece.equals("cat"))
6     {
7         System.out.println("cat starts at index " + i);
8     }
9 }

```

```

index:      0 1 2 3 4 5 6 7 8 9 10
character:  c o n c a t e n a t e

```

"concatenate" has length 11, so *i* goes from 0 to 8. The pieces are con, onc, nca, cat, ate, ten, ena, nat, ate. Only the piece starting at index 3 equals "cat", so the output is cat starts at index 3. The last piece, ate, starts at 8 and ends at index 10, the last character, so no piece is missed.

WATCH OUT

Keeping the one-character bound. Copying the header from 2.10.1 without changing it crashes as soon as the piece would run past the end:

```

1 String word = "balloon";
2 for (int i = 0; i < word.length(); i++)
3 {
4     System.out.println(word.substring(i, i + 2));
5 }

```

It prints the six pieces ba through on, one per line. Then *i* becomes 6, $6 < 7$ is true, and line 4 asks for `word.substring(6, 8)`. The string has only 7 characters, so Java throws an exception like

```
java.lang.StringIndexOutOfBoundsException: Range [6, 8) out of bounds for length 7
(wording varies). The bound must subtract the extra characters each piece needs:
i < word.length() - 1 for pairs.
```

PAUSE AND TRY

Question 5 Tracing

Write the exact output.

```

1 String word = "frog";
2 for (int i = 0; i < word.length() - 1; i++)
3 {
4     System.out.println(word.substring(i, i + 2));
5 }

```

Question 6 Multiple choice

A loop body uses `word.substring(i, i + 4)`. Which condition lets `i` reach every possible starting index without an exception?

- A. `i < word.length()`
- B. `i < word.length() - 4`
- C. `i <= word.length() - 4`
- D. `i <= word.length() - 3`

2.10.4 Counting Matches

A flag answers "yes or no?". Often the question is "how many?" How many vowels are in a word? How many times does "ana" appear in "banana"? In Topic 2.9 you counted how often a condition was met by using a **counter**: an `int` set to 0 before the loop and increased by 1 inside an `if`. Counting in a string is that same algorithm, with the traversal from this chapter supplying the values to test.

The difference from a flag is small but important. A flag stops changing once it is switched; a counter keeps going up every time the condition is true, so at the end it holds the total.

KEY IDEA

To count the characters or substrings that meet a condition, set a counter to 0 before the loop, add 1 inside the loop each time the condition is true, and read the counter after the loop.

WORKED EXAMPLE Counting vowels

The vowel test is the `indexOf` trick from 2.10.2. This time a match adds 1 instead of raising a flag.

```

1 String word = "education";
2 String vowels = "aeiou";
3 int count = 0;
4 for (int i = 0; i < word.length(); i++)
5 {
6     String letter = word.substring(i, i + 1);
7     if (vowels.indexOf(letter) >= 0)
8     {
9         count++;
10    }
11 }
12 System.out.println("Vowels: " + count);

```

index:	0	1	2	3	4	5	6	7	8
character:	e	d	u	c	a	t	i	o	n
count:	1	1	2	2	3	3	4	5	5

The bottom row is the value of `count` after each pass.

Vowels: 5

The counter goes up on the passes for e, u, a, i, and o, and stays the same for the consonants. Line 12 runs once, after the loop, and prints the total.

Now count a longer substring. The loop bound comes from 2.10.3: three-character pieces start at 0 through `length() - 3`.

```
1 String word = "banana";
2 int count = 0;
3 for (int i = 0; i <= word.length() - 3; i++)
4 {
5     if (word.substring(i, i + 3).equals("ana"))
6     {
7         count++;
8     }
9 }
10 System.out.println("ana appears " + count + " times");
```

```
index:      0 1 2 3 4 5
character:  b a n a n a
```

i	word.substring(i, i + 3)	Matches "ana"?	count
0	"ban"	no	0
1	"ana"	yes	1
2	"nan"	no	1
3	"ana"	yes	2

ana appears 2 times

Look at the two matches: one uses indices 1–3 and the other uses 3–5. They share the a at index 3. Because the loop checks **every** starting position, it counts **overlapping matches**: matches that share characters with each other. If you circled ana in banana with a pen and refused to reuse a letter, you would find only one. The loop finds two. When you trace a counting loop, check each starting index on its own; do not skip ahead past a match.

WATCH OUT

Declaring the counter inside the loop. Putting `int count = 0;` inside the braces seems natural, since the counting happens there:

```
1 String word = "education";
2 for (int i = 0; i < word.length(); i++)
3 {
4     int count = 0;
5     if ("aeiou".indexOf(word.substring(i, i + 1)) >= 0)
6     {
7         count++;
8     }
9 }
10 System.out.println("Vowels: " + count);
```

This does not compile. Java reports `error: cannot find symbol at count` on line 10. A variable declared inside a loop body exists only inside those braces, so the print statement after the loop cannot see it. (Even if it could, the counter would be reset to 0 on every pass and could never get past 1.) The fix is to declare `int count = 0;` before the loop, as in the vowel example, so there is one counter for the whole traversal.

PAUSE AND TRY

Question 7 Tracing

Write the exact output, and list each index where a match is found.

```
1 String word = "mississippi";
2 int count = 0;
3 for (int i = 0; i < word.length() - 1; i++)
4 {
5     if (word.substring(i, i + 2).equals("ss"))
6     {
7         count++;
8     }
9 }
10 System.out.println(count);
```

Question 8 Multiple choice

What does this print?

```
1 String word = "aaaa";
2 int count = 0;
3 for (int i = 0; i < word.length() - 1; i++)
4 {
5     if (word.substring(i, i + 2).equals("aa"))
6     {
7         count++;
8     }
9 }
10 System.out.println(count);
```

- A. 1
- B. 2

- C. 3
- D. 4

2.10.5 Building a New String

So far the algorithms have produced a `boolean` or an `int`. Sometimes the answer has to be a new string: a phrase with its spaces removed, a username made from parts of a name, a message with certain characters replaced. Strings are immutable (Topic 1.15), so the program cannot edit the original. Instead it builds a new string one piece at a time.

The pattern mirrors the counter. Before the loop, create a result variable holding the **empty string**, `""`, which has no characters. Inside the loop, add a piece to the end with `+=` when you want it. After the loop, the result holds every piece that was added, in order. This is the same **accumulator** idea from Topic 2.9, where a variable collected a running total a little at a time; here it collects text instead of a number.

Each `result += letter;` concatenates and makes `result` refer to a **new** string object; the old one is not changed. The original string is never touched at all.

KEY IDEA

To build a new string, start a result variable at `""` before the loop and use `+=` inside the loop to add the pieces you want, in order.

WORKED EXAMPLE Removing spaces

Line 6 uses `!` from Topic 2.5: the condition is true when the character is **not** a space.

```

1 String phrase = "a big red dog";
2 String result = "";
3 for (int i = 0; i < phrase.length(); i++)
4 {
5     String letter = phrase.substring(i, i + 1);
6     if (!letter.equals(" "))
7     {
8         result += letter;
9     }
10 }
11 System.out.println(result);
12 System.out.println(phrase);

```

i	letter	Added?	result after the pass
0	"a"	yes	"a"
1	" "	no	"a"
2	"b"	yes	"ab"
3	"i"	yes	"abi"
4	"g"	yes	"abig"

5	" "	no	"abig"
6-8	r, e, d	yes	"abigred"
9	" "	no	"abigred"
10-12	d, o, g	yes	"abigreddog"

```
abigreddog
a big red dog
```

The second line shows that `phrase` is unchanged: the algorithm made a new string instead of editing the old one.

Now vary what gets added. With an `if/else`, every pass adds something, but what it adds depends on the character. Here a space is replaced by an underscore and every other character is copied:

```
1 String phrase = "hop to it";
2 String result = "";
3 for (int i = 0; i < phrase.length(); i++)
4 {
5     String letter = phrase.substring(i, i + 1);
6     if (letter.equals(" "))
7     {
8         result += "_";
9     }
10    else
11    {
12        result += letter;
13    }
14 }
15 System.out.println(result);
```

The output is `hop_to_it`. The result has the same length as the original, 9 characters, because exactly one piece is added on each pass. In the space-removing example, some passes added nothing, so the result was shorter.

WATCH OUT

Not starting the result at "". Declaring the variable without giving it a value looks harmless:

```
1 String phrase = "a big red dog";
2 String result;
3 for (int i = 0; i < phrase.length(); i++)
4 {
5     result += phrase.substring(i, i + 1);
6 }
7 System.out.println(result);
```

This does not compile. Java reports

error: variable `result` might not have been initialized on line 5. `result += piece` means `result = result + piece`, so the first pass needs a value for `result` to add to, and there is

none. The fix is `String result = ""`. The empty string is the right starting value because adding pieces to it leaves exactly those pieces, just as 0 is the right starting value for a counter.

PAUSE AND TRY

Question 9 Tracing

Write the exact output.

```
1 String word = "hop";
2 String result = "";
3 for (int i = 0; i < word.length(); i++)
4 {
5     String letter = word.substring(i, i + 1);
6     result += letter + letter;
7 }
8 System.out.println(result);
```

Question 10 Multiple choice

What does this print?

```
1 String word = "banana";
2 String result = "";
3 for (int i = 0; i < word.length(); i++)
4 {
5     String letter = word.substring(i, i + 1);
6     if ("aeiou".indexOf(letter) == -1)
7     {
8         result += letter;
9     }
10 }
11 System.out.println(result);
```

- A. aaa
- B. bnn
- C. banana
- D. bn

2.10.6 Reversing a String

Is "racecar" the same backward? What is "drawer" spelled backward? Both questions need a new string with the characters in the opposite order. That is a special case of building a new string from 2.10.5: the pieces are the same characters, and only the order changes. There are two ways to get that order.

Way 1: go forward, but add each character to the front. The concatenation `letter + reversed` puts the new character **before** everything collected so far. Each new character pushes the earlier ones to the right, so the first character ends up last.

WORKED EXAMPLE Reversing by adding to the front

Watch line 6: `letter` comes first in the concatenation.

```

1 String word = "stop";
2 String reversed = "";
3 for (int i = 0; i < word.length(); i++)
4 {
5     String letter = word.substring(i, i + 1);
6     reversed = letter + reversed;
7 }
8 System.out.println(reversed);

```

i	letter	letter + reversed	reversed after the pass
0	"s"	"s" + ""	"s"
1	"t"	"t" + "s"	"ts"
2	"o"	"o" + "ts"	"ots"
3	"p"	"p" + "ots"	"pots"

pots

Way 2: go backward, and add each character to the end. Change the loop instead of the concatenation. Start at the last index, `word.length() - 1`, keep going while `i >= 0`, and subtract 1 each time. Then the ordinary `+=` adds the characters in reverse order.

```

1 String word = "drawer";
2 String reversed = "";
3 for (int i = word.length() - 1; i >= 0; i--)
4 {
5     reversed += word.substring(i, i + 1);
6 }
7 System.out.println(reversed);

```

```

index:      0 1 2 3 4 5
character:  d r a w e r

```

`i` takes the values 5, 4, 3, 2, 1, 0, which add `r`, `e`, `w`, `a`, `r`, `d`. The output is `reward`. The condition `i >= 0` includes index 0, so the first character is added last; after that `i` becomes -1 and the loop ends.

Both ways give a correct reverse. Use whichever reads more clearly to you, but be ready to trace either one.

Way	Loop direction	How a character is added
1	0 up to <code>length() - 1</code>	<code>reversed = letter + reversed;</code> (to the front)
2	<code>length() - 1</code> down to 0	<code>reversed += letter;</code> (to the end)

KEY IDEA

To reverse a string, either loop forward and add each character to the front (letter + reversed), or loop from the last index down to 0 and add each character to the end.

WATCH OUT

Starting the backward loop at `length()`. The length feels like "the end," so it is tempting to start there:

```
1 String word = "drawer";
2 String reversed = "";
3 for (int i = word.length(); i >= 0; i--)
4 {
5     reversed += word.substring(i, i + 1);
6 }
7 System.out.println(reversed);
```

Nothing is printed. On the very first pass, `i` is 6 and line 5 asks for `word.substring(6, 7)`, which is past the end of a 6-character string. The program stops with a `StringIndexOutOfBoundsException` (in one Java version the message reads `Range [6, 7) out of bounds for length 6`). It is the same off-by-one mix-up as in 2.10.1, at the other end of the string: the last index is `length() - 1`. Fix the start to `int i = word.length() - 1`.

PAUSE AND TRY**Question 11** Tracing

A student wants to reverse "stop" and writes the code below. Write the exact output and explain why it is not reversed.

```
1 String word = "stop";
2 String reversed = "";
3 for (int i = 0; i < word.length(); i++)
4 {
5     String letter = word.substring(i, i + 1);
6     reversed = reversed + letter;
7 }
8 System.out.println(reversed);
```

Putting it together

A **palindrome** is text that reads the same forward and backward, such as `level` or `racecar`. The phrase `never odd or even` is a palindrome too, if you ignore the spaces. Checking it uses two algorithms from this chapter, one after the other, and then a comparison from Topic 1.15.

```

1 String phrase = "never odd or even";
2 String letters = "";
3 for (int i = 0; i < phrase.length(); i++)
4 {
5     String letter = phrase.substring(i, i + 1);
6     if (!letter.equals(" "))
7     {
8         letters += letter;
9     }
10 }
11 String reversed = "";
12 for (int i = letters.length() - 1; i >= 0; i--)
13 {
14     reversed += letters.substring(i, i + 1);
15 }
16 System.out.println(letters);
17 System.out.println(reversed);
18 System.out.println("Palindrome: " + letters.equals(reversed));

```

Step 1, lines 2–10: build a new string without spaces (2.10.5). `letters` starts as "", and every character that is not a space is added to the end. After the loop, `letters` holds `neveroddoreven`, 14 characters.

Step 2, lines 11–15: reverse it (2.10.6). The second loop starts at index 13, the last index of `letters` (not of `phrase`), and moves down to 0, adding each character to the end of `reversed`.

```

index:      0 1 2 3 4 5 6 7 8 9 10 11 12 13
letters:    n e v e r o d d o r e v e n

```

Reading the row from index 13 down to 0 gives `neveroddoreven`, the same characters in the same order, so `reversed` is also `neveroddoreven`.

Step 3, line 18: compare. `letters.equals(reversed)` checks whether the two strings hold the same characters, and it is true.

```

neveroddoreven
neveroddoreven
Palindrome: true

```

Notice that the second loop uses `letters.length()`, not `phrase.length()`. The `phrase` has 17 characters, so a second loop written as `for (int i = phrase.length() - 1; i >= 0; i--)` would start at index 16 of `letters`, which has only 14 characters, and the program would crash with a `StringIndexOutOfBoundsException` before printing anything. Whenever one algorithm feeds another, each loop must be bounded by the string it actually reads.

Vocabulary

Term	Definition	Example
traversal	Visiting the positions of a string one at a time with a loop	<pre>for (int i = 0; i < s.length(); i++)</pre>

flag	A <code>boolean</code> variable set before a loop and switched inside it when something you watch for happens	<code>hasDigit = true;</code>
counter	An <code>int</code> set to 0 before a loop and increased by 1 each time a condition is true	<code>count++;</code>
overlapping matches	Matches that share characters; a loop that checks every starting index counts all of them	<code>"ana"</code> appears twice in <code>"banana"</code>
empty string	A string with no characters, the starting value for building a new string	<code>String result = "";</code>
accumulator	A variable that collects a result across the passes of a loop	<code>result += letter;</code>
palindrome	Text that reads the same forward and backward	<code>racecar</code>

Chapter summary

- A **traversal** visits every character: loop `i` from 0 while `i < s.length()` and use `s.substring(i, i + 1)`. The last index is `length() - 1`, and going one past it throws a `StringIndexOutOfBoundsException`.
- To look at substrings of length `n`, use `s.substring(i, i + n)` and stop at `i <= s.length() - n`.
- A **flag** answers a yes-or-no question about a string: start false and switch to true to ask "does any character...?", or start true and switch to false to ask "does every character...?". Never switch it back inside the loop.
- A **counter** answers "how many?": start at 0 before the loop and add 1 for each match. Checking every starting index counts **overlapping matches**.
- To build a new string, start at `""` and add pieces with `+=`. To reverse, either add each character to the front while looping forward, or loop from the last index down to 0 and add to the end.

Review questions

Question 12 Tracing

Write the exact output, and show the value of `result` after each pass.

```

1 String code = "abcdefg";
2 String result = "";
3 for (int i = 0; i < code.length(); i += 2)
4 {
5     result = code.substring(i, i + 1) + result;
6 }
7 System.out.println(result);

```

Question 13 Debugging

This code should count how many times "the" appears in "bathe the", which is 2. It prints 1. Find the error and write the corrected line.

```
1 String text = "bathe the";
2 int count = 0;
3 for (int i = 0; i < text.length() - 3; i++)
4 {
5     if (text.substring(i, i + 3).equals("the"))
6     {
7         count++;
8     }
9 }
10 System.out.println(count);
```

Question 14 Multiple choice

Assume String word = "drawer"; and String reversed = ""; . Which of these segments leave reversed equal to "reward"?

```
1 // I
2 for (int i = 0; i < word.length(); i++)
3 {
4     reversed = word.substring(i, i + 1) + reversed;
5 }
6
7 // II
8 for (int i = word.length() - 1; i >= 0; i--)
9 {
10     reversed += word.substring(i, i + 1);
11 }
12
13 // III
14 for (int i = word.length() - 1; i > 0; i--)
15 {
16     reversed += word.substring(i, i + 1);
17 }
```

- A. I only
- B. II only
- C. I and II only
- D. I, II, and III

Question 15 Tracing

This version of a check stops looking as soon as it finds what it wants. Write the exact output, and give the values of `i` and `found` each time the `while` condition is checked.

```
1 String word = "rhythmic";
2 int i = 0;
3 boolean found = false;
4 while (i < word.length() && !found)
5 {
6     if ("aeiou".indexOf(word.substring(i, i + 1)) >= 0)
7     {
8         found = true;
9     }
10    else
11    {
12        i++;
13    }
14 }
15 System.out.println(found + " " + i);
```

Question 16 Short answer

A student counts the words in a sentence by counting the spaces and adding 1, since words are separated by spaces. Their code prints `Words: 4` for the sentence below, which has three words. Explain why, and state what must be true about a sentence for this algorithm to give the right count.

```
1 String sentence = "go home now";
2 int spaces = 0;
3 for (int i = 0; i < sentence.length(); i++)
4 {
5     if (sentence.substring(i, i + 1).equals(" "))
6     {
7         spaces++;
8     }
9 }
10 System.out.println("Words: " + (spaces + 1));
```

Question 17 Debugging

This code should print `level` and then `true`, because `"level"` is a palindrome. It prints `level` and then `false`. Find the error and write the corrected line.

```
1 String word = "level";
2 String reversed = "";
3 for (int i = word.length() - 1; i >= 0; i--)
4 {
5     reversed += word.substring(i, i + 1);
6 }
7 System.out.println(reversed);
8 System.out.println(reversed == word);
```

Reflection

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

- ☐ I can write the loop header that visits every character of a string and explain why it uses less than, not less than or equal. Reread: 2.10.1 Visiting Every Character
- ☐ I can use a flag to check whether any character, or every character, has a property. Reread: 2.10.2 Checking a Property with a Flag
- ☐ I can choose the loop bound that reaches every substring of a given length. Reread: 2.10.3 Looking at Longer Substrings
- ☐ I can count the characters or substrings that meet a condition, including overlapping matches. Reread: 2.10.4 Counting Matches
- ☐ I can trace an algorithm that builds a new string with +=. Reread: 2.10.5 Building a New String
- ☐ I can reverse a string in two ways and spot a loop that only copies it. Reread: 2.10.6 Reversing a String

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
lnt
```

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

The loop starts at 1 and adds 2 each time, so `i` takes the values 1, 3, and 5, which are `l`, `n`, and `t`. Next `i` would be 7, and `7 < 6` is false, so the loop ends. Line 4 uses `print`, so the three letters appear on one line, and line 6 ends that line.

Reread 2.10.1 *Visiting Every Character* if your output differed.

Question 2

C. Indices run from 0 to `length() - 1`. Starting at 0 and continuing while `i < city.length()` hits each of those exactly once.

- **A** is wrong: the last pass has `i` equal to the length, and `substring(i, i + 1)` then throws a `StringIndexOutOfBoundsException`. This comes from treating the length as the last index.
- **B** is wrong: it never visits index 0, so the first character is skipped. This comes from counting positions starting at 1.
- **D** is wrong: it stops at `length() - 2`, so the last character is skipped. This is an off-by-one error in the other direction, one pass too few.

Reread 2.10.1 *Visiting Every Character* if you chose differently.

Question 3

B. The flag `noSpaces` starts true and switches to false the first time a character equals " ". That is the "every character" pattern: it is true only if no character is a space. "ice cream" has a space at index 3, so the flag becomes false and stays false.

- **A** is wrong: the flag is switched to `false`, not `true`, when a space is found, and the name `noSpaces` describes the opposite question.
- **C** is wrong: the question is right, but the string does contain a space, so the answer is `false`.
- **D** is wrong: there is no `else`, so the flag reflects every character, not only the last one. The last character, `m`, is not a space anyway.

Reread 2.10.2 *Checking a Property with a Flag* if you chose differently.

Question 4

It prints `All digits: false`, even though every character of "4072" is a digit. The loop only ever sets the flag to `false` (when it finds a non-digit). It never sets it to `true`. So if it starts false, nothing can make it true, and the answer is false for every PIN.

A complete answer says:

- The "every character" check starts by assuming the answer is yes (`true`) because no bad character has been seen yet.

- The loop's only job is to switch the flag to `false` when it finds a character that fails.
- Starting at `false` means the flag never has a chance to be true.

Reread 2.10.2 *Checking a Property with a Flag* if your explanation differed.

Question 5

```
fr
ro
og
```

"frog" has length 4, so the condition is `i < 3` and `i` takes the values 0, 1, and 2. The pieces are `substring(0, 2) = fr`, `substring(1, 3) = ro`, and `substring(2, 4) = og`. The pieces overlap: each one shares a character with the next.

Reread 2.10.3 *Looking at Longer Substrings* if your output differed.

Question 6

C. The end of the piece, `i + 4`, can be at most `word.length()`, so the largest `i` is `word.length() - 4`, and `<=` includes it.

- **A** is wrong: near the end, `i + 4` passes the length and `substring` throws a `StringIndexOutOfBoundsException`. This is the one-character bound copied without change.
- **B** is wrong: it stops one too early and never looks at the last four characters, an off-by-one error that gives no crash, only a wrong answer.
- **D** is wrong: it allows `i = word.length() - 3`, and then `i + 4` is one past the length, so the last pass crashes.

Reread 2.10.3 *Looking at Longer Substrings* if you chose differently.

Question 7

```
2
```

```
index:    0 1 2 3 4 5 6 7 8 9 10
character: m i s s i s s i p p i
```

The loop checks pairs starting at 0 through 9. The pair is "ss" at index 2 (s s at 2 and 3) and at index 5 (s s at 5 and 6). The pair starting at 3 is si and the one at 4 is is, so neither counts. The counter ends at 2.

Reread 2.10.4 *Counting Matches* if your output differed.

Question 8

C. The pairs start at indices 0, 1, and 2 (`length() - 1` is 3, so `i < 3`). Each pair is "aa", so the counter reaches 3. These are overlapping matches: the pair at index 1 shares a character with the pairs at 0 and 2.

- **A** is wrong: it treats the whole string as one match.
- **B** is wrong: it counts without reusing characters (indices 0–1 and 2–3). The loop checks every starting index, so overlaps count.
- **D** is wrong: there are four characters but only three pairs; a pair starting at index 3 would need index 4.

Reread 2.10.4 *Counting Matches* if you chose differently.

Question 9

hhoopp

i	letter	letter + letter	result
0	"h"	"hh"	"hh"
1	"o"	"oo"	"hhoo"
2	"p"	"pp"	"hhoopp"

On each pass the right side, `letter + letter`, is built first and then added to the end of `result`.

Reread 2.10.5 *Building a New String* if your output differed.

Question 10

B. `"aeiou".indexOf(letter) == -1` is true when the letter is **not** a vowel, so only `b`, `n`, and `n` are added, in order.

- **A** is wrong: that keeps the vowels. It reads `== -1` as "is a vowel," the opposite of what it tests.
- **C** is wrong: the original is copied only if every character passes the test; the `a`s fail it.
- **D** is wrong: both `n`s are added. The result keeps every character that passes, including repeats.

Reread 2.10.5 *Building a New String* if you chose differently.

Question 11

stop

i	letter	reversed after the pass
0	"s"	"s"
1	"t"	"st"
2	"o"	"sto"
3	"p"	"stop"

The loop goes forward **and** adds each character to the end, so the characters land in their original order: this is a copy, not a reverse. To reverse, change one of the two: write `reversed = letter + reversed`; (add to the front), or loop from `word.length() - 1` down to 0.

Reread 2.10.6 *Reversing a String* if your output differed.

Question 12

geca

i	code.substring(i, i + 1)	result after the pass
0	"a"	"a"
2	"c"	"ca"
4	"e"	"eca"

6	"g"	"geca"
---	-----	--------

Two ideas combine here. The update `i += 2` visits only indices 0, 2, 4, and 6 (then `8 < 7` is false). Adding each character to the **front** reverses the order of the characters that were visited.

Reread 2.10.1 *Visiting Every Character* and 2.10.6 *Reversing a String* if your output differed.

Question 13

The error is on line 3. The string has length 9, and "the" appears at index 2 (inside bathe) and at index 6. The condition `i < text.length() - 3` is `i < 6`, so the loop never tries index 6, the last three-character piece. This is an off-by-one error that loops one time too few. It does not crash, so the wrong answer is easy to miss.

Corrected line:

```
1 for (int i = 0; i <= text.length() - 3; i++)
```

Now `i` runs from 0 to 6, the piece starting at 6 is "the", and the program prints 2.

(`i < text.length() - 2` is another correct way to write the same bound.)

Reread 2.10.3 *Looking at Longer Substrings* if you missed the error.

Question 14

C. Segment I loops forward and adds each character to the front. Segment II loops backward from the last index through 0 and adds to the end. Both are correct reverses.

- **A** and **B** are wrong because each leaves out a correct segment. The two ways of reversing are equally valid.
- **D** is wrong: segment III uses `i > 0`, so it never visits index 0 and leaves out the `d`. It produces `rewar`. This is an off-by-one error that loops one time too few.

Reread 2.10.6 *Reversing a String* if you chose differently.

Question 15

```
true 6
```

```
index:    0 1 2 3 4 5 6 7
character: r h y t h m i c
```

i	found	Condition	What the body does
0	false	true	r is not a vowel; <code>i = 1</code>
1	false	true	h; <code>i = 2</code>
2	false	true	y is not in "aeiou"; <code>i = 3</code>
3	false	true	t; <code>i = 4</code>
4	false	true	h; <code>i = 5</code>
5	false	true	m; <code>i = 6</code>
6	false	true	the letter i is a vowel; <code>found = true</code>
6	true	false	(loop ends)

This is the flag from 2.10.2 used as part of a `while` condition (Topic 2.7). When `found` becomes true, `!found` is false, so the `&&` condition is false and the loop ends. Because `i` is increased only in the `else`, it still holds 6, the index of the first vowel. Note that `y` is not in `"aeiou"`, so this code does not count it as a vowel.

Reread 2.10.2 *Checking a Property with a Flag* if your trace differed.

Question 16

There are **two** spaces between `go` and `home` (indices 2 and 3) and one between `home` and `now` (index 8). The loop counts every space, so `spaces` is 3, and `3 + 1` is 4.

A complete answer says:

- The counter counts every character that is a space, not every gap between words.
- The algorithm assumes exactly one space between each pair of words.
- It also assumes no space at the start or end of the sentence; each of those would add 1 to the count as well.

Reread 2.10.4 *Counting Matches* if your explanation differed.

Question 17

The error is on line 8. The reverse is built correctly: line 7 prints `level`. But `reversed == word` compares **references** (Topic 2.6), asking whether the two variables refer to the very same object. `reversed` was built by concatenation inside the loop, so it is a different `String` object from the literal `"level"`, even though the characters match. `==` gives `false`.

Corrected line:

```
1 System.out.println(reversed.equals(word));
```

`equals` (Topic 1.15) compares the characters, so it prints `true`.

Reread *Putting it together* if you missed the error.