

1.14 Calling Instance Methods

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

By the end you will be able to

- Tell an instance method from a class method by reading its header, and explain why an instance method needs an object.
- Call an instance method with the dot operator on a variable that refers to an object.
- Pass the right arguments to an instance method and use the value it returns, or recognize when it returns nothing.
- Predict what a sequence of calls on a `String` or a `Scanner` produces, including calls on two different objects.
- Explain why calling a method on null compiles but throws a `NullPointerException`, and fix it.

What to remember from earlier

- A signature is the method name plus the ordered parameter types, and arguments must match it. ([Topic 1.9: Method Signatures](#), [1.9.5 Method Signatures](#))
- A void method returns nothing; a non-void method returns a value you must store or use. ([Topic 1.9: Method Signatures](#), [1.9.7 Return Values](#))
- Class methods are called with the class name and a dot, and their headers include static. ([Topic 1.10: Calling Class Methods](#), [1.10.2 Calling with the Class Name and a Dot](#))
- An object is one specific instance of a class, with its own attributes. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.1 Classes and Objects](#))
- A reference variable holds a reference to the object, or null when there is none. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.4 Reference Variables](#))
- You create objects with `new` and a constructor, as in `new Scanner(System.in)`. ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.1 Constructors and the new Keyword](#))

1.14.1 Instance Methods Belong to Objects

Suppose a sign-up form needs to know how many characters a city name has. With `Math.abs(-6)` from Topic 1.10, everything the method needs arrives as an argument: you hand it `-6`, and it hands back `6`. But "how many characters?" has no single answer. It depends on *which* city name you mean: "`Boston`" has 6 characters and "`Chicago`" has 7. The method needs to work with the characters stored inside one particular `String` object.

Recall from Topic 1.12 that every object is an instance of its class and carries its own attributes. Two `String` objects are made from the same blueprint, but each one stores its own characters.

In plain words: some methods are questions you ask a specific object, such as "how long are you?" Each object answers using its own data, so the same question asked of two objects can get two different answers. A method like this is called an **instance method**: a method that is called on an object (an instance of a class) and works with that object's own attributes. Compare it with a **class method** from Topic 1.10, which belongs to the class as a whole and does not work with any particular object; anything it needs from you arrives as arguments.

KEY IDEA

An instance method is called on one specific object and uses that object's data. The same instance method called on two different objects can return two different results.

You can tell the two kinds apart in the documentation. A class method's header includes the word `static`; an instance method's header does not. Here are two headers you have seen in documentation:

Header (from the documentation)	Class	Kind of method
<code>static int abs(int x)</code>	Math	Class method (static is present)
<code>int length()</code>	String	Instance method (no static)

The `String` method `length()` returns an `int`: the number of characters in the string the method is called on. Every character counts, including spaces and punctuation. Its parameter list is empty, because the method needs nothing from you; the characters are already inside the object. (You will meet many more `String` methods in Topic 1.15. This chapter explains each one when it first appears.)

WORKED EXAMPLE One method, two objects

Two `String` variables refer to two different objects. Watch what `length()` returns for each. (The exact way to write the call, `city.length()`, is the subject of 1.14.2; for now, read it as "ask city for its length.")

```
1 String city = "Boston";
2 String team = "Red Sox";
3 System.out.println(city.length());
4 System.out.println(team.length());
```

Call (line)	Object it is called on	Characters counted	Returns
<code>city.length()</code> (3)	"Boston"	B-o-s-t-o-n	6
<code>team.length()</code> (4)	"Red Sox"	R-e-d-(space)-S-o-x	7

The output is:

```
6
7
```

The method is the same on both lines. The answers differ because each call works on a different object. Notice that the space in "Red Sox" counts as a character.

Now change one thing: put a class method and an instance method side by side, so you can see where each one gets its data.

```
1 int distance = Math.abs(-6);
2 String city = "Boston";
3 int letters = city.length();
4 System.out.println(distance);
5 System.out.println(letters);
```

```
6
6
```

Both lines print 6, but for different reasons. `Math.abs(-6)` gets its data from the argument `-6` inside the parentheses; no object is involved. `city.length()` gets its data from the object `city` refers to; the parentheses are empty because the object already holds the characters.

WATCH OUT

Calling an instance method through the class name. Because `Math.abs(-6)` uses the class name, students often try `String.length()` to get a string's length:

```
1 String city = "Boston";
2 int letters = String.length();
3 System.out.println(letters);
```

The compiler reports

non-static method `length()` cannot be referenced from a static context. "Non-static" means "instance method." The message is Java's way of asking: the length of *which* string? The class `String` is a blueprint, not a string with characters, so it has no length. The fix is to call the method on the object: `int letters = city.length();`.

PAUSE AND TRY

Question 1 Multiple choice

Which of these documentation headers describes an instance method?

- A. `static double sqrt(double x)`
- B. `static int abs(int x)`
- C. `int length()`
- D. `static double random()`

Question 2 Tracing

What does this code print?

```
1 String first = "Ana";
2 String last = "de la Cruz";
3 System.out.println(last.length());
4 System.out.println(first.length());
```

1.14.2 The Dot Operator

You have now seen calls like `city.length()`. It is time to take that form apart, because every instance method call in Java uses it, and knowing what each piece does tells you what Java does when the call runs.

In Topic 1.10 you wrote a class name, a dot, and a method name: `Math.abs(-6)`. The **dot operator** (`.`) connects a method name to the thing it belongs to. For a class method, that thing is the class. For an instance method, it is an object, named by a variable that refers to the object.

KEY IDEA

To call an instance method, write the object's variable name, a dot, the method name, and parentheses with any arguments: `objectVariable.methodName(arguments)`.

Here are the pieces of `int letters = city.length();`:

Piece	What it is	Why it is there
<code>city</code>	A variable of type <code>String</code>	Tells Java <i>which object</i> to call the method on
<code>.</code>	The dot operator	Says "use this method of that object"
<code>length</code>	The method name	Says which method to run
<code>()</code>	The argument list (empty)	Required on every call, even with no arguments; it is what makes a call
<code>int letters</code>	A variable to hold the result	Keeps the returned value (more on this in 1.14.3)

What happens when the call runs? Remember from Topic 1.12 that `city` holds a reference, like an address, not the object itself. Java reads the reference in `city`, follows it to the `String` object, and runs `length()` on that object. As with any method call (Topic 1.9), the program pauses at the call, runs the method, and then continues right where it left off, with the returned value in place of the call.

You have been using the dot operator on an object since Topic 1.1. `System.out` is an object whose job is printing to the screen, and `println` is one of its instance methods. `System.out.println("Hi")` calls `println` on that object.

WORKED EXAMPLE Two variables, one object

Since the dot operator follows whatever reference the variable holds, what happens when two variables hold the same reference? Watch `player` and `captain`:

```

1 String player = "Jordan";
2 String captain = player;
3 System.out.println(player.length());
4 System.out.println(captain.length());
5 captain = "Li";
6 System.out.println(player.length());
7 System.out.println(captain.length());

```

Line	What happens	Output
1	<code>player</code> gets a reference to the object "Jordan"	
2	The reference in <code>player</code> is copied into <code>captain</code> ; both point to "Jordan"	

3	Follow player to "Jordan", call length()	6
4	Follow captain to the same "Jordan" object, call length()	6
5	captain now refers to a different object, "Li"; player is unchanged	
6	Follow player, still "Jordan"	6
7	Follow captain, now "Li"	2

The output is:

```
6
6
6
2
```

The method runs on whatever object the variable refers to **at the moment of the call**. Line 2 copied a reference, not the characters, so lines 3 and 4 asked the same object. After line 5, the two variables lead to different objects, so the same call gives different answers.

WATCH OUT

Leaving off the parentheses. Students sometimes write `song.length` because there are no arguments to put inside:

```
1 String song = "Happy Birthday";
2 int letters = song.length;
3 System.out.println(letters);
```

The compiler reports `cannot find symbol, with symbol: variable length`. Without parentheses, Java reads `length` as a variable stored inside the object, not as a method call, and `String` has no variable by that name. The parentheses are what tell Java "run this method." The fix is `int letters = song.length();`.

PAUSE AND TRY

Question 3 Multiple choice

In the call `title.length()`, what job does `title` do?

- A. It is the argument passed to `length`.
- B. It names the class that `length` belongs to.
- C. It refers to the object that `length` is called on.
- D. It is the variable that stores the returned value.

Question 4 Tracing

What does this code print? Track which object each variable refers to after each line.

```

1 String pet = "Biscuit";
2 String other = pet;
3 pet = "Max";
4 System.out.println(other.length());
5 System.out.println(pet.length());

```

1.14.3 Arguments and Return Values

`length()` takes no arguments. But many questions you might ask a string need extra information. "Where is the @ in this email address?" needs to know *what* to look for. Instance methods handle this exactly the way the methods in Topic 1.9 did: through parameters and arguments.

Recall the rules from Topic 1.9. A method's signature lists its parameter types, and the arguments in a call must match them in number, order, and type. A non-void method returns a value, which you must store in a variable or use in an expression; a void method returns nothing. Every one of those rules applies to instance methods. The only new part is the object in front of the dot.

Here is a second `String` method. Its header is `int indexOf(String str)`. It searches the string it is called on for the text `str` and returns the **position** where that text first appears. Positions are counted from **0**, so the first character is at position 0, the second at 1, and so on. If the text does not appear at all, `indexOf` returns **-1**. (Topic 1.15 explains positions fully; this is enough for now.)

KEY IDEA

An instance method call has a type, just like a class method call. A non-void call is replaced by the value it returns, so you store that value or use it in an expression. A void call returns nothing and must stand alone as a statement.

WORKED EXAMPLE Using returned values in expressions

This code finds how many characters come after the @ in an email address.

```

1 String email = "sam@school.org";
2 int atSpot = email.indexOf("@");
3 int total = email.length();
4 int afterAt = total - atSpot - 1;
5 System.out.println(atSpot);
6 System.out.println(total);
7 System.out.println(afterAt);

```

Number the characters of "sam@school.org" first, starting from 0:

s	a	m	@	s	c	h	o	o	l	.	o	r	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Line	Evaluation	Stored
------	------------	--------

2	<code>email.indexOf("@")</code> : the argument "@" is a String, matching the parameter. @ first appears at position 3	<code>atSpot = 3</code>
3	<code>email.length()</code> : positions 0 to 13, so 14 characters	<code>total = 14</code>
4	<code>total - atSpot - 1</code> → <code>14 - 3 - 1</code> → 10	<code>afterAt = 10</code>

Output:

```
3
14
10
```

Both calls are non-void, so each returned `int` was stored in a variable and then used in arithmetic. You could also skip the variables and write

`email.length() - email.indexOf("@") - 1`; each call is replaced by its returned value, and the expression is evaluated as usual.

Now change one thing: keep the same object but change the argument.

```
1 String email = "sam@school.org";
2 System.out.println(email.indexOf("s"));
3 System.out.println(email.indexOf("school"));
4 System.out.println(email.indexOf("o"));
5 System.out.println(email.indexOf("!"));
```

```
0
4
7
-1
```

"s" first appears at position 0 (the s in sam; the later s in school does not matter, because `indexOf` reports the *first* match). "school" begins at position 4. "o" first appears at position 7, inside school. "!" never appears, so the result is -1. The object stayed the same; only the argument changed, and so did the answer. An instance method's result can depend on both the object and the arguments.

What about a void instance method? `println` is one: its header in the documentation is `void println(String x)`. It does its job (printing) and returns nothing, so it cannot be used where a value is needed:

```
1 int result = System.out.println("Hello");
```

The compiler reports incompatible types: `void` cannot be converted to `int`. There is no value to store, so a void call must be a statement by itself.

WATCH OUT

Calling a non-void method and ignoring what it returns. This compiles and runs, but it is a logic error:

```
1 String password = "tiger42";
2 password.length();
3 System.out.println("Password checked.");
```

Password checked.

Line 2 really does call `length()`, and the method really does return 7. But nothing stores or uses that 7, so it is thrown away. Students write this expecting the length to appear somewhere, but calling a method does not print its result. Use the returned value:

```
1 String password = "tiger42";
2 System.out.print("Password length: ");
3 System.out.println(password.length());
```

Password length: 7

PAUSE AND TRY

Question 5 Tracing

What does this code print?

```
1 String word = "banana";
2 int first = word.indexOf("an");
3 int size = word.length();
4 System.out.println(first + size);
5 System.out.println(word.indexOf("n") * 2);
6 System.out.println(word.indexOf("x"));
```

Question 6 Debugging

This code is meant to print the number of characters in "Denver", but it does not compile. Find the error, write the corrected line, and explain what the compiler would report.

```
1 String city = "Denver";
2 int letters = city.length("Denver");
3 System.out.println(letters);
```

1.14.4 Calling Methods on a Scanner

In Topics 1.4 and 1.13 you created a `Scanner` with `new Scanner(System.in)` and read keyboard input with calls like `input.nextInt()`. Now you can see what those calls were: instance methods called on a `Scanner` object with the dot operator. A `Scanner` is a good second example because it behaves differently from a `String` in one important way.

Here are the `Scanner` methods this chapter uses. A **token** is one chunk of input text separated from the next by spaces, such as `12` or `pears`.

Header	What it does
<code>int nextInt()</code>	Reads the next token as a whole number and returns it as an <code>int</code>
<code>String next()</code>	Reads the next token and returns it as a <code>String</code>
<code>void close()</code>	Tells the <code>Scanner</code> you are finished with it; returns nothing

Keyboard input changes every time a program runs, which makes its output impossible to predict on paper. So the examples here use a different `Scanner` constructor, one that takes a `String`. The `Scanner` then reads from that text instead of the keyboard. (Recall from Topic 1.13 that a class can have several constructors with different parameter lists.) The method calls work the same way either way.

Here is the new idea. A `Scanner` object remembers how far it has read. Each `nextInt()` or `next()` call returns the next token and moves the object's place forward. So calling the same method on the same `Scanner` twice gives two different results, because the first call changed where the object is. Calling `city.length()` twice always gives the same answer, because measuring a string does not change anything about it.

KEY IDEA

An instance method can use and update the state of the object it is called on. Each call to `nextInt()` or `next()` returns the next token and moves that `Scanner`'s place forward.

WORKED EXAMPLE Reading tokens one call at a time

ScannerTokens.java

```

1  import java.util.Scanner;
2
3  public class ScannerTokens
4  {
5      public static void main(String[] args)
6      {
7          Scanner reader = new Scanner("12 7 pears");
8          int first = reader.nextInt();
9          int second = reader.nextInt();
10         String fruit = reader.next();
11         System.out.println(first + second);
12         System.out.println(fruit);
13         reader.close();
14     }
15 }
```

The ▶ in the table marks where `reader` will read next.

Line	Call	Returns	Text left after the call
7	(constructor)		▶12 7 pears
8	<code>reader.nextInt()</code>	12	▶7 pears
9	<code>reader.nextInt()</code>	7	▶pears
10	<code>reader.next()</code>	"pears"	(nothing left)

Line 11 prints `first + second`, which is `12 + 7`. Line 12 prints the `String` returned on line 10. Line 13 calls the void method `close()` as a statement on its own, because it returns nothing.

```
19
pears
```

Lines 8 and 9 make the identical call, `reader.nextInt()`, yet return 12 and then 7. The call on line 8 moved `reader` forward.

Now change one thing: use two `Scanner` objects and alternate between them.

`TwoScanners.java`

```
1 import java.util.Scanner;
2
3 public class TwoScanners
4 {
5     public static void main(String[] args)
6     {
7         Scanner morning = new Scanner("5 10");
8         Scanner evening = new Scanner("20 40");
9         System.out.println(morning.nextInt());
10        System.out.println(evening.nextInt());
11        System.out.println(morning.nextInt());
12        System.out.println(evening.nextInt());
13    }
14 }
```

```
5
20
10
40
```

Each object keeps its own place. Reading from `evening` on line 10 does not move `morning`, so line 11 continues where `morning` stopped and returns 10. The variable in front of the dot decides whose data the call uses and whose place moves forward.

WATCH OUT

Asking for the wrong kind of token. `nextInt()` can only return a whole number. If the next token is a word, the call fails while the program runs:

WrongToken.java

```
1 import java.util.Scanner;
2
3 public class WrongToken
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner("pears 12");
8         System.out.println("Reading the count...");
9         int count = reader.nextInt();
10        System.out.println(count);
11    }
12 }
```

```
Reading the count...
Exception in thread "main" java.util.InputMismatchException
```

The program compiles, because `reader.nextInt()` is a correct call with a correct return type. The problem appears only at run time, when the next token turns out to be `pears`. Students forget that the call reads the *next* token, not "the first number anywhere." The fix is to read the tokens in the order they appear:

WrongTokenFixed.java

```
1 import java.util.Scanner;
2
3 public class WrongTokenFixed
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner("pears 12");
8         System.out.println("Reading the count...");
9         String fruit = reader.next();
10        int count = reader.nextInt();
11        System.out.println(count);
12    }
13 }
```

```
Reading the count...
12
```

PAUSE AND TRY

Question 7 Tracing

What does this program print?

ScoreReader.java

```
1 import java.util.Scanner;
2
3 public class ScoreReader
4 {
5     public static void main(String[] args)
6     {
7         Scanner scores = new Scanner("8 3 9");
8         int a = scores.nextInt();
9         int b = scores.nextInt();
10        System.out.println(a - b);
11        System.out.println(scores.nextInt());
12    }
13 }
```

Question 8 Short answer

`city.length()` returns the same value every time you call it, but `reader.nextInt()` usually returns a different value each time. Explain why, using the idea of an object's data.

1.14.5 Calling a Method on null

Every call so far had an object in front of the dot. But Topic 1.4 and Topic 1.13 introduced a reference variable that refers to no object at all: one holding **null**. Picture an address book entry with the address left blank. What happens if you try to visit it?

The dot operator works by following the reference in the variable to an object. When the variable holds **null**, there is nothing to follow and no object to run the method on. Java cannot carry out the call, so it throws an exception: a **NullPointerException**. This is a run-time error (Topic 1.1). The compiler does not catch it, because the compiler only checks that the variable's type, `String`, has a `length()` method, and it does. Whether the variable will hold an object when that line runs is something only running the program reveals.

KEY IDEA

Calling a method on a reference that is null compiles, but throws a `NullPointerException` when that line runs. The statements before it run normally; the statements after it do not.

WORKED EXAMPLE A call that works, then fails

NicknameNull.java

```

1 public class NicknameNull
2 {
3     public static void main(String[] args)
4     {
5         String nickname = "Sam";
6         System.out.println(nickname.length());
7         nickname = null;
8         System.out.println("Nickname removed.");
9         System.out.println(nickname.length());
10        System.out.println("Done.");
11    }
12 }

```

Line	nickname holds	What happens	Output
5	reference to "Sam"	Variable created	
6	reference to "Sam"	Follow the reference, call <code>length()</code>	3
7	<code>null</code>	The reference is replaced by <code>null</code>	
8	<code>null</code>	Prints a message; no method is called on <code>nickname</code>	Nickname removed.
9	<code>null</code>	No object to follow: <code>NullPointerException</code> , program stops	
10	<code>null</code>	Never runs	

The console shows:

```

3
Nickname removed.
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because ...
    at NicknameNull.main(NicknameNull.java:9)

```

Read the message the way you learned in Topic 1.1: the exception name tells you what kind of problem it is, `Cannot invoke "String.length()"` says which call failed, and `NicknameNull.java:9` gives the line. The end of the first line names the variable that was `null` (or, depending on how the program was compiled, a placeholder such as `<local1>`). The same call on line 6 worked; what changed is what the variable held.

Now change one thing: give `nickname` an empty string instead of `null`.

```

1 String nickname = "";
2 System.out.println(nickname.length());

```

0

No crash. `""` is a real `String` object that happens to contain zero characters, so there is an object to call `length()` on, and it returns 0. `null` means there is no object at all.

Value of <code>nickname</code>	Is there an object?	<code>nickname.length()</code>
"Sam"	Yes, 3 characters	3
" "	Yes, 0 characters	0
<code>null</code>	No	Throws <code>NullPointerException</code>

WATCH OUT

"It compiled, so the variable must have an object." Students often assume that any call the compiler accepts will run. The compiler checks types, not values, so it cannot tell that a variable will hold `null` when a line runs. When you trace code, track what each reference variable holds, and before every dot ask: *does this variable refer to an object right now?* The fix for a `NullPointerException` is to make sure the variable refers to a real object (for example, `nickname = "Sam";` or `nickname = " ";`) before the method is called on it.

PAUSE AND TRY**Question 9** Tracing

This program compiles. Write its complete console output (the first line of any exception message is enough).

CoachNull.java

```

1 public class CoachNull
2 {
3     public static void main(String[] args)
4     {
5         String coach = null;
6         String assistant = "Kim";
7         System.out.println(assistant.length());
8         String backup = coach;
9         System.out.println(backup.length());
10        System.out.println("Practice starts.");
11    }
12 }
```

Question 10 Multiple choice

After which declaration does `label.length()` throw a `NullPointerException`?

- A. `String label = "";`
- B. `String label = "null";`
- C. `String label = null;`
- D. `String label = " ";`

Putting it together

A café's order system receives each order as a line of text, such as `"3 bageLs"`, and prints a slip. The program below uses every idea in this chapter.

OrderSlip.java

```

1  import java.util.Scanner;
2
3  public class OrderSlip
4  {
5      public static void main(String[] args)
6      {
7          Scanner order = new Scanner("3 bagels");
8          int quantity = order.nextInt();
9          String item = order.next();
10         order.close();
11         System.out.print("Quantity: ");
12         System.out.println(quantity);
13         System.out.print("Item: ");
14         System.out.println(item);
15         System.out.print("Letters in item: ");
16         System.out.println(item.length());
17         System.out.print("Position of g: ");
18         System.out.println(item.indexOf("g"));
19     }
20 }

```

Line	Call or statement	Object the call uses	Returns / result
7	new Scanner("3 bagels")	(creates the object)	order refers to a new Scanner
8	order.nextInt()	the Scanner, at >3 bagels	3, stored in quantity
9	order.next()	the Scanner, at >bagels	"bagels", stored in item
10	order.close()	the Scanner	void: returns nothing
11–18	System.out.print...	the System.out object	prints labels and values
16	item.length()	the String "bagels"	6
18	item.indexOf("g")	the String "bagels"	2 (b 0, a 1, g 2)

```

Quantity: 3
Item: bagels
Letters in item: 6
Position of g: 2

```

Notice three things:

- **Each call names its object.** `order.nextInt()` reads from the `Scanner`; `item.length()` measures the `String`. The variable in front of the dot decides whose data is used.
- **Returned values are kept or used.** The `nextInt()` and `next()` results are stored in variables; the `length()` and `indexOf` results are passed straight to `println`. The void call `order.close()` stands alone.
- **Every dot had a real object behind it.** If the program also declared `String note = null;` and called `note.length()`, it would print whatever came before that line and then stop with a `NullPointerException`, exactly as in 1.14.5.

Common mistakes

The Watch Out boxes each taught one mistake. This table collects the messages Java gives, so you can look one up when you see it.

You wrote	Java says	Fix
<code>String.length()</code>	Compile error: non-static method <code>length()</code> cannot be referenced from a static context	Call it on an object: <code>city.length()</code>
<code>song.length</code>	Compile error: cannot find symbol ... variable <code>length</code>	Add parentheses: <code>song.length()</code>
<code>city.length("Denver")</code>	Compile error: method <code>length</code> in class <code>String</code> cannot be applied to given types	Match the parameter list: <code>city.length()</code>
<code>int result = System.out.println("Hello");</code>	Compile error: incompatible types: void cannot be converted to <code>int</code>	Call a void method as a statement by itself
<code>nextInt()</code> when the next token is a word	Run-time: <code>java.util.InputMismatchException</code> appears	Read tokens in the order they appear
A call on a variable holding <code>null</code>	Run-time: <code>java.lang.NullPointerException</code>	Make the variable refer to an object first

Vocabulary

Term	Definition	Example
instance method	A method called on a specific object that works with that object's data; no <code>static</code> in its header	<code>city.length()</code>
class method	(Topic 1.10) A method that belongs to the class itself; <code>static</code> in its header	<code>Math.abs(-6)</code>
dot operator	The <code>.</code> that connects an object variable (or class name) to the method being called	<code>email.indexOf("@")</code>
position	Where a character sits in a string, counting from 0	In <code>"sam"</code> , <code>m</code> is at 2
token	One chunk of input text separated from others by spaces	<code>"12 7 pears"</code> has 3 tokens
null	(Topics 1.4, 1.13) The value of a reference variable that refers to no object	<code>String nickname = null;</code>
NullPointerException	The exception thrown when a method is called on a <code>null</code> reference	<code>nickname.length()</code> when <code>null</code>

Chapter summary

- An **instance method** is called on one specific object and works with that object's data; its header has no `static`. The same method called on different objects can return different results.
- You call an instance method with the **dot operator**: `objectVariable.methodName(arguments)`. Java follows the reference in the variable to the object, so two variables that share a reference call methods on the same object.
- Instance methods follow the same rules as any method: arguments must match the parameter list, a non-void result must be stored or used, and a void method is called as a statement by itself.
- Some instance methods change the object's state. Each `Scanner` call such as `nextInt()` returns the next token and moves that `Scanner` forward, so repeated calls give different results.
- Calling a method on a variable that holds **null** compiles but throws a **NullPointerException** when that line runs, because there is no object to call the method on.

Review questions

Question 11 Multiple choice

The documentation shows these headers: `int indexOf(String str)` in the `String` class and `static int abs(int x)` in the `Math` class. Given `String code = "AB-42";`, which statement compiles and stores 2 in `n`?

- A. `int n = code.indexOf("-");`
- B. `int n = String.indexOf("-");`
- C. `int n = Math.abs(code);`
- D. `int n = code.indexOf;`

Question 12 Tracing

What does this code print?

```
1 String a = "summer";
2 String b = a;
3 a = "fall";
4 int x = b.length() - a.length();
5 int y = b.indexOf("m") + a.indexOf("l");
6 System.out.println(x);
7 System.out.println(y);
```

Question 13 Tracing

What does this program print? (Integer division from Topic 1.3 applies.)

PizzaOrder.java

```
1 import java.util.Scanner;
2
3 public class PizzaOrder
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("4 pizza 10");
8         int slices = input.nextInt();
9         String food = input.next();
10        int price = input.nextInt();
11        System.out.println(price / slices);
12        System.out.println(food.length() * slices);
13    }
14 }
```

Question 14 Debugging

This program compiles, prints 5, and then crashes. It was meant to print 5, 5, and Done. Explain what goes wrong, why the compiler did not catch it, and give a fix.

MiddleName.java

```
1 public class MiddleName
2 {
3     public static void main(String[] args)
4     {
5         String first = "Maria";
6         String middle = null;
7         System.out.println(first.length());
8         System.out.println(middle.length());
9         System.out.println("Done.");
10    }
11 }
```

Question 15 Short answer

A classmate asks: "Why do we write `Math.abs(-6)` with the class name, but `title.length()` with a variable name? Why not `String.length()`?" Write an answer that explains the difference.

Question 16 Multiple choice

What does this program print?

SharedScanner.java

```
1 import java.util.Scanner;
2
3 public class SharedScanner
4 {
5     public static void main(String[] args)
6     {
7         Scanner a = new Scanner("1 2 3");
8         Scanner b = a;
9         a.nextInt();
10        System.out.println(b.nextInt());
11    }
12 }
```

- A. 1
- B. 2
- C. 3
- D. Nothing; it throws an exception

Reflection

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

☐ I can tell an instance method from a class method by reading its header. Reread: 1.14.1 Instance Methods Belong to Objects

☐ I can call an instance method on an object with the dot operator and say which object it uses. Reread: 1.14.2 The Dot Operator

☐ I can pass matching arguments and store or use the value an instance method returns. Reread: 1.14.3 Arguments and Return Values

☐ I can trace a series of Scanner calls and explain why repeated calls return different values. Reread: 1.14.4 Calling Methods on a Scanner

☐ I can explain why calling a method on null throws a NullPointerException and how to fix it. Reread: 1.14.5 Calling a Method on null

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

C. Instance method headers do not include `static`. `length()` belongs to `String` and works with the characters of the object it is called on.

- **A** is wrong: `static` marks `sqrt` as a class method. It gets its data from its argument, not from an object.
- **B** is wrong: `abs` is also `static`, so it is a class method you call with `Math.abs(...)`.
- **D** is wrong: an empty parameter list does not make a method an instance method. `random()` has no parameters, but its header says `static`, so it is a class method. Look for `static`, not for empty parentheses.

Reread 1.14.1 *Instance Methods Belong to Objects* if you chose differently.

Question 2

```
10
3
```

`last` refers to `"de la Cruz"`: d, e, space, l, a, space, C, r, u, z. That is 10 characters, because both spaces count. `first` refers to `"Ana"`, which has 3. The `last` line runs first because it comes first in the code.

Reread 1.14.1 *Instance Methods Belong to Objects* if you got a different count.

Question 3

C. In `objectVariable.methodName(arguments)`, the part before the dot is the variable that refers to the object. Java follows its reference and runs `length()` on that object.

- **A** is wrong: arguments go inside the parentheses, and these parentheses are empty.
- **B** is wrong: `title` is a variable, not a class. Writing the class name (`String.length()`) does not compile, because a class has no characters to count.
- **D** is wrong: the call does not store anything by itself. To keep the result you would write something like `int size = title.length();`.

Reread 1.14.2 *The Dot Operator* if you chose differently.

Question 4

```
7
3
```

After line	pet refers to	other refers to
1	"Biscuit"	(not yet created)
2	"Biscuit"	"Biscuit"
3	"Max"	"Biscuit"

Line 2 copies the reference, so `other` points to the same "Biscuit" object. Line 3 gives `pet` a new reference but does not change `other`. So `other.length()` is 7 and `pet.length()` is 3.

Reread 1.14.2 *The Dot Operator* if you thought `other` changed too.

Question 5

```
7
4
-1
```

Number "banana": b 0, a 1, n 2, a 3, n 4, a 5.

Expression	Evaluation	Value
<code>word.indexOf("an")</code>	"an" first starts at position 1	1
<code>word.length()</code>	6 characters	6
<code>first + size</code>	1 + 6	7
<code>word.indexOf("n") * 2</code>	first n at position 2, then 2 * 2	4
<code>word.indexOf("x")</code>	no x in "banana"	-1

Reread 1.14.3 *Arguments and Return Values* if any value differed. A common slip is counting positions from 1 instead of 0.

Question 6

Error: line 2 passes an argument to `length`, but its header is `int length()`, with an empty parameter list. The method already knows which string to measure: the one it is called on.

Corrected line:

```
1 int letters = city.length();
```

The compiler reports method `length` in class `String` cannot be applied to given types, followed by required: no arguments and found: `String`. The arguments must match the parameter list in number, and zero parameters means zero arguments. With the fix, the code prints 6.

```
1 String city = "Denver";
2 int letters = city.length();
3 System.out.println(letters);
```

Reread 1.14.3 *Arguments and Return Values* if you missed the error.

Question 7

```
5
9
```

Line	Call	Returns	Text left
8	<code>scores.nextInt()</code>	8	3 9
9	<code>scores.nextInt()</code>	3	9
10	<code>prints a - b</code>		9

11	<code>scores.nextInt()</code>	9	(nothing)
----	-------------------------------	---	-----------

`a - b` is `8 - 3`, so 5. The call on line 11 continues where `scores` left off, so it returns 9, not 8.

Reread 1.14.4 *Calling Methods on a Scanner* if your output differed.

Question 8

Model answer: Both are instance methods, so each works with the data inside the object it is called on. `length()` only looks at the string's characters and changes nothing, so the object is the same on every call and the answer is the same. `nextInt()` returns the next token and also moves the `Scanner`'s place forward, so the next call starts from a different point and returns the following token.

A complete answer must say:

- each call uses the data of the object it is called on, and
- `nextInt()` changes the `Scanner`'s position, while `length()` changes nothing about the string.

Reread 1.14.4 *Calling Methods on a Scanner* if your reasoning differed.

Question 9

```
3
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because ...
```

Line 7 calls `length()` on `"Kim"` and prints 3. Line 8 copies the reference in `coach` into `backup`, and that reference is `null`, so `backup` holds `null` too. Line 9 calls `length()` on `null`, which throws a `NullPointerException`. Line 10 never runs. Copying `null` into a new variable does not create an object.

Reread 1.14.5 *Calling a Method on null* if your output differed.

Question 10

C. `null` (no quotes) means the variable refers to no object, so the call has nothing to run on.

- **A** is wrong: `""` is a real `String` object with zero characters. `length()` returns 0.
- **B** is wrong: `"null"` in quotes is an ordinary string of four letters. `length()` returns 4.
- **D** is wrong: `" "` is a string containing one space, and a space is a character. `length()` returns 1.

Reread 1.14.5 *Calling a Method on null* if you chose differently.

Question 11

A. `indexOf` is an instance method (no `static`), so it is called on an object with the dot operator. In `"AB-42"`, `A` is at position 0, `B` at 1, and `-` at 2, so the call returns 2.

```
1 String code = "AB-42";
2 int n = code.indexOf("-");
3 System.out.println(n);
```

- **B** is wrong: it calls an instance method through the class name. There is no string to search, and the compiler reports `non-static method ... cannot be referenced from a static context`.
- **C** is wrong: `Math.abs` is a class method, but its parameter is an `int`, and `code` is a `String`. The argument's type does not match the parameter list.

- **D** is wrong: without parentheses it is not a method call, so the compiler reports `cannot find symbol`. It also gives no argument to search for.

Reread 1.14.2 *The Dot Operator* and 1.14.3 *Arguments and Return Values* if you chose differently.

Question 12

2
4

Step	Evaluation	Result
2	b gets a copy of the reference in a; both refer to "summer"	
3	a now refers to "fall"; b still refers to "summer"	
4	b.length() - a.length() → 6 - 4	x = 2
5	b.indexOf("m"): s 0, u 1, m 2 → 2. a.indexOf("l"): f 0, a 1, l 2 → 2. 2 + 2	y = 4

The key step is line 3: giving a a new reference does not change b, so the calls on b still use "summer".

Reread 1.14.2 *The Dot Operator* if you used "fall" for b, or 1.14.3 *Arguments and Return Values* if a position was off.

Question 13

2
20

Line	Call	Returns	Stored in
8	input.nextInt()	4	slices
9	input.next()	"pizza"	food
10	input.nextInt()	10	price

Line 11: price / slices is 10 / 4. Both are int, so the result keeps only the whole-number part: 2.
Line 12: food.length() is 5 (p-i-z-z-a), times 4 is 20.

Reread 1.14.4 *Calling Methods on a Scanner* if the token values were off.

Question 14

What goes wrong: line 6 stores null in middle, so it refers to no object. Line 8 calls middle.length(), and there is no object to call it on, so Java throws a NullPointerException. Line 7 already printed 5; line 9 never runs.

Why it compiles: the compiler only checks that middle is a String and that String has a length() method. It does not track what value the variable will hold when line 8 runs.

Fix: make middle refer to a real String before calling the method, for example:

MiddleNameFixed.java

```
1 public class MiddleNameFixed
2 {
3     public static void main(String[] args)
4     {
5         String first = "Maria";
6         String middle = "Elena";
7         System.out.println(first.length());
8         System.out.println(middle.length());
9         System.out.println("Done.");
10    }
11 }
```

```
5
5
Done.
```

Reread 1.14.5 *Calling a Method on null* if your explanation differed.

Question 15

Model answer: `abs` is a class method. It gets everything it needs from its argument, so it does not need an object, and you call it through the class name. `length` is an instance method. It counts the characters of one particular string, so it must be called on an object, and the variable in front of the dot says which object. `String.length()` does not work because the class is only a blueprint: it has no characters of its own, so there is nothing to count.

A complete answer must say:

- class methods (`static`) are called with the class name and get their data from arguments;
- instance methods are called on an object, through a variable that refers to it, and use that object's data;
- `String.length()` has no object to work on, so it does not compile.

Reread 1.14.1 *Instance Methods Belong to Objects* if your answer differed.

Question 16

B. Line 8 copies the reference, so `a` and `b` refer to the **same** `Scanner` object. Line 9 calls `nextInt()` through `a`, which reads `1` and moves that object's place forward (the returned `1` is not stored, which is allowed). Line 10 calls `nextInt()` through `b`, on the same object, so it continues from where line 9 stopped and returns `2`.

- **A** is wrong: it treats `b` as a separate `Scanner` starting from the beginning. Only `new` creates a new object; line 8 just copies a reference.
- **C** is wrong: only one token was read before line 8, not two.
- **D** is wrong: there are three whole-number tokens, so neither call runs out of text or finds a word, and both variables refer to a real object.

Reread 1.14.2 *The Dot Operator* and 1.14.4 *Calling Methods on a Scanner* if you chose differently.