

1.9 Method Signatures

Self-Study Guide · About 65 minutes · With answer key

By the end you will be able to

- Explain what a method is and why you can use one without knowing how it was written.
- Trace a program as it jumps into a method and back out again.
- Read a method header, name its return type, name, and parameters, and write its signature.
- Decide whether a call's arguments match a method's parameters in number, order, and type, and use documentation to pick the right method.
- Explain why a method cannot change the caller's variable by changing its parameter.
- Tell void methods from non-void methods and use a returned value correctly.
- Explain what overloaded methods are and predict which version a call uses.

What to remember from earlier

- Statements in main run one at a time, top to bottom. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.3 Sequencing: Order Matters](#))
- An assignment stores the value on the right in the variable on the left. ([Topic 1.4: Assignment Statements and Input, 1.4.1 The Assignment Operator](#))
- An int widens to a double automatically, but a double needs a cast to become an int. ([Topic 1.5: Casting and Range of Variables, 1.5.1 Java Widens an int to a double](#))
- Compound operators such as += update a variable in place. ([Topic 1.6: Compound Assignment Operators, 1.6.1 Adding to a Variable in One Step](#))
- A precondition is true before a method is used, and a postcondition is true afterward. ([Topic 1.8: Documentation with Comments](#))

1.9.1 Methods Are Named Blocks of Code

Suppose the morning announcements remind students to bring their library books, and the reminder has to appear in several places in a program. You could type the same `System.out.println` statement three times. But if the wording ever changes, you would have to find and fix every copy, and missing one would leave the program inconsistent. It would be better to write the reminder once, give it a name, and then say "do the reminder now" wherever it is needed.

That is exactly what a method does. First, a word about braces. Since Topic 1.1 you have seen { and } grouping lines together. Any section of code enclosed in a pair of braces is called a **block** of code. The body of `main` is a block, and so is the whole class.

A **method** is a block of code with a name, and the code inside it runs only when the method is called. To **call** a method means to tell Java "run that method's code now." You write a call as the method's name followed by parentheses, ending with a semicolon: `printReminder();`.

You have been calling methods since your first program. `System.out.println("Hi");` is a call to a method named `println` that someone else wrote. And `main` is a method too: it is the one Java calls for you automatically when the program starts.

KEY IDEA

A method is a named block of code. Writing it down does not run it; its statements run only when the method is called.

WORKED EXAMPLE A method that is never called

This program contains two methods: `main` and a new one named `printReminder`. Read the whole file before predicting the output.

`NeverCalled.java`

```
1 public class NeverCalled
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Homeroom starts at 8:00.");
6         System.out.println("Have a good day!");
7     }
8
9     public static void printReminder()
10    {
11        System.out.println("Bring your library book.");
12    }
13 }
```

Here is how to read it:

Lines	What they are
3–7	The <code>main</code> method. Java calls it when the program starts.
9	The first line of the <code>printReminder</code> method. It gives the method its name.
10–12	The block (between the braces) that belongs to <code>printReminder</code> . This is the code the name stands for.

Every method in this chapter's examples starts with `public static`. Those words are explained in a later topic; read past them for now. The word `void` is explained in 1.9.7.

When the program runs, Java calls `main`, and `main` runs lines 5 and 6:

```
Homeroom starts at 8:00.
Have a good day!
```

"Bring your library book." **never appears**. Line 11 is a perfectly good statement, but it sits inside `printReminder`, and nothing ever calls `printReminder`. A method's code waits until it is called.

Now change one thing: add two calls to `printReminder` inside `main`.

ReminderTwice.java

```
1 public class ReminderTwice
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Homeroom starts at 8:00.");
6         printReminder();
7         System.out.println("Lunch is at 11:30.");
8         printReminder();
9     }
10
11     public static void printReminder()
12     {
13         System.out.println("Bring your library book.");
14     }
15 }
```

```
Homeroom starts at 8:00.
Bring your library book.
Lunch is at 11:30.
Bring your library book.
```

Each call on lines 6 and 8 runs the method's block once, so the reminder appears twice even though its `println` statement is written only once. If the reminder's wording changes, you edit line 13 and both calls pick up the change. Because `printReminder` is in the same class as `main`, the call is just its name and parentheses; Topic 1.10 explains the full rules for calling methods.

WATCH OUT

Leaving off the parentheses. Students sometimes write just the method's name, thinking the name alone means "run it."

MissingParens.java

```
1 public class MissingParens
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Homeroom starts at 8:00.");
6         printReminder;
7     }
8
9     public static void printReminder()
10    {
11        System.out.println("Bring your library book.");
12    }
13 }
```

Java reports `MissingParens.java:6: error: not a statement`. Without parentheses, `printReminder` is not a call; it is just a name, and a name by itself is not an instruction Java can carry out. The parentheses are what make it a call. Fix it by writing `printReminder();`.

PAUSE AND TRY**Question 1** Multiple choice

Which statement about methods is true?

- A. A method's statements run as soon as Java reaches the method in the file.
- B. A method's statements run only when the method is called.
- C. A method can be called only once in a program.
- D. A block of code is any line that ends in a semicolon.

Question 2 Tracing

What does this program print?

Practice.java

```
1 public class Practice
2 {
3     public static void main(String[] args)
4     {
5         printWarmUp();
6         System.out.println("Run three laps.");
7         printWarmUp();
8     }
9
10    public static void printWarmUp()
11    {
12        System.out.println("Stretch for five minutes.");
13    }
14
15    public static void printCoolDown()
16    {
17        System.out.println("Walk one lap slowly.");
18    }
19 }
```

1.9.2 Procedural Abstraction

How does `System.out.println` actually get letters onto your screen? You have no idea, and you have used it in every program since Topic 1.1. That is not a gap in your knowledge; it is the whole point. You only need to know **what** it does (display a value and move to a new line), not **how** it does it.

In plain words: a method lets the person who uses it ignore the details of how it works. A driver uses a steering wheel without knowing how the steering system is built; turning the wheel left turns the car left, and that is all the driver needs. The name for this idea is **procedural abstraction**: using a method by knowing what it does, even if you do not know how it was written. ("Abstraction" means hiding details so you can think about the bigger picture.)

So where do you learn *what* a method does? From its documentation. In Topic 1.7 you saw that an API's documentation describes the behaviors a class offers, and in Topic 1.8 you saw documentation comments that state preconditions and postconditions. That documentation is your

whole view of the method. Here is a simplified version of the documentation for the `Math` class's square-root method:

```
1 /**
2  * Returns the positive square root of a.
3  * Precondition: a is not negative.
4  */
5 public static double sqrt(double a)
```

The comment tells you what goes in (a number that is not negative), and what comes out (its square root). The code that computes square roots is nowhere in sight, and you do not need it.

KEY IDEA

Procedural abstraction means you can use a method by knowing what it does, from its name and documentation, without knowing how it was written.

WORKED EXAMPLE Using a method you have never seen inside

The documentation says `sqrt` gives back the square root of the number in its parentheses. A call to it is written `Math.sqrt(...)`; the `Math.` part tells Java which class the method belongs to, and Topic 1.10 explains it fully.

```
1 System.out.println(Math.sqrt(144.0));
2 System.out.println(Math.sqrt(2.0));
```

```
12.0
1.4142135623730951
```

Step by step:

1. Line 1 calls `sqrt` with 144.0. The method does its work out of sight and produces 12.0, because $12 \times 12 = 144$. `println` displays it.
2. Line 2 calls `sqrt` with 2.0. There is no whole number whose square is 2, so the result is a long decimal. You did not need to know how Java finds that decimal, only that it is the square root.

Now change the situation: use the same method for a new problem. A ladder leans against a wall, its top 3 meters up the wall, its base 4 meters from the wall. From geometry, the ladder's length is the square root of $(3 \times 3 + 4 \times 4)$.

```
1 double wallHeight = 3.0;
2 double distanceFromWall = 4.0;
3 double ladderLength = Math.sqrt(wallHeight * wallHeight + distanceFromWall *
4 distanceFromWall);
4 System.out.println(ladderLength);
```

```
5.0
```

Java first evaluates the expression inside the parentheses, $9.0 + 16.0 = 25.0$, and then `sqrt` gives back 5.0. The method was written long before anyone thought about your ladder, but because you

know what it does, you can use it anywhere you need a square root. That reuse is the payoff of procedural abstraction: one well-documented method serves many programs, and nobody has to write square-root code twice.

WATCH OUT

Guessing instead of reading the documentation. Procedural abstraction lets you skip the *how*, not the *what*. The method `Math.pow(a, b)` computes *a* raised to the power *b*. A student who knows that 2 to the power 3 is 8 predicts this prints 8:

```
1 System.out.println(Math.pow(2, 3));
```

It prints `8.0`. The documentation's header is `public static double pow(double a, double b)`, and the `double` in front of `pow` means the method always gives back a `double` (1.9.7 explains this). The student guessed from the method's name instead of reading what the documentation promises. Before using a method, read its documentation for what goes in and what comes out.

PAUSE AND TRY

Question 3 Multiple choice

Which situation is the best example of procedural abstraction?

- A. Reading the code inside `Math.sqrt` line by line before deciding whether to use it
- B. Using `Math.sqrt` after reading its documentation, without ever seeing its code
- C. Copying the steps for computing a square root into your program each time you need one
- D. Writing a comment above every line of `main`

Question 4 Short answer

A classmate says, "I can't use `Math.sqrt` safely, because I don't know how it calculates square roots." Explain why they are wrong, and name what they *do* need to know.

1.9.3 What Happens When a Method Is Called

In Topic 1.1 you learned that statements run one at a time, top to bottom. But a method's statements are not where the call is; they are somewhere else in the file. So when Java reaches a call, what happens to "top to bottom"? And once the method finishes, where does the program pick up again?

In plain words: a method call is like a detour. You are reading a recipe, and step 3 says "make the sauce (see page 40)." You flip to page 40, do every step there, and then come back to step 4 of the original recipe, the step right after the one that sent you away. You do not start the recipe over, and you do not skip ahead.

The order in which statements actually run is called the **flow of control**. A method call **interrupts** the normal top-to-bottom order: Java pauses where it is, runs the statements in the method, and when the method's last statement has run, control **returns** to the point immediately after the call. (A method can also hand back control early with a `return` statement; you will see that in 1.9.7.)

KEY IDEA

A method call interrupts the normal order. Java runs every statement in the method first, then returns to the point immediately after the call and continues from there.

WORKED EXAMPLE Following the detour

Put your finger on line 5 and follow the flow of control.

Announcements.java

```

1  public class Announcements
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Good morning, Hoppers!");
6          printLunchMenu();
7          System.out.println("Have a great day!");
8      }
9
10     public static void printLunchMenu()
11     {
12         System.out.println("Lunch: tacos");
13         System.out.println("Side: apple slices");
14     }
15 }
```

Step	Line running	What happens	Printed
1	5	First statement of main	Good morning, Hoppers!
2	6	Call to printLunchMenu. main pauses here; control jumps to line 12	
3	12	First statement of printLunchMenu	Lunch: tacos
4	13	Last statement of printLunchMenu	Side: apple slices
5	7	The method is done, so control returns to the line right after the call	Have a great day!

The complete output:

```

Good morning, Hoppers!
Lunch: tacos
Side: apple slices
Have a great day!
```

The method's two lines appear **between** main's lines, at the exact spot where the call was. Line 7 waits until the whole method is finished.

Now change one thing: let a method call another method. Here `main` calls `printSchedule`, and `printSchedule` itself calls `printLunchMenu`.

NestedCalls.java

```
1 public class NestedCalls
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Start of announcements");
6         printSchedule();
7         System.out.println("End of announcements");
8     }
9
10    public static void printSchedule()
11    {
12        System.out.println("Period 1: Math");
13        printLunchMenu();
14        System.out.println("Period 5: Art");
15    }
16
17    public static void printLunchMenu()
18    {
19        System.out.println("Lunch: tacos");
20    }
21 }
```

```
Start of announcements
Period 1: Math
Lunch: tacos
Period 5: Art
End of announcements
```

The same rule applies at every level. Line 6 sends control into `printSchedule`, which prints line 12. Line 13 sends control into `printLunchMenu`, which prints line 19. When `printLunchMenu` finishes, control returns to the point right after **its** call, which is line 14 inside `printSchedule`, not `main`. Only when `printSchedule` finishes does control return to `main` at line 7. Each method returns to whoever called it.

WATCH OUT

"Methods run in the order they are written in the file." Students often trace a file from the top line to the bottom line. Here a method is written *above* `main`:

DefinedFirst.java

```
1 public class DefinedFirst
2 {
3     public static void printBell()
4     {
5         System.out.println("The bell rings.");
6     }
7
8     public static void main(String[] args)
9     {
10        System.out.println("First period");
11        printBell();
12        System.out.println("Second period");
13    }
14 }
```

A top-to-bottom reading predicts "The bell rings." first. The real output is:

```
First period
The bell rings.
Second period
```

The program always starts in `main`, and a method runs only when the flow of control reaches a call to it. Where the method is written in the file does not matter. To trace a program, start at the first line of `main` and follow the calls.

PAUSE AND TRY**Question 5** Tracing

What does this program print? Keep track of where control returns after each method finishes.

FieldTrip.java

```
1 public class FieldTrip
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Board the bus.");
6         visitMuseum();
7         eatLunch();
8         System.out.println("Ride home.");
9     }
10
11     public static void visitMuseum()
12     {
13         System.out.println("See the dinosaurs.");
14         eatLunch();
15         System.out.println("Visit the gift shop.");
16     }
17
18     public static void eatLunch()
19     {
20         System.out.println("Eat lunch.");
21     }
22 }
```

Question 6 Multiple choice

A method is called on line 9 of `main`. After the method's last statement runs, which statement runs next?

- A. The first statement of `main`
- B. The statement immediately after the call on line 9
- C. The next method written below it in the file
- D. Nothing; the program ends when a method finishes

1.9.4 Parameters and Arguments

`printReminder` prints the same thing every time. That is fine for a reminder, but most useful methods need to work with different values each time. `Math.sqrt` would be pointless if it could only find the square root of one number. A ticket booth needs to print the total for *any* price and *any* number of tickets. How does a method get the values it should work on?

In plain words: the method announces, in its first line, the values it needs and gives each one a name and a type. Whoever calls the method supplies actual values for those names. Think of a form with blanks labeled "Price" and "Number of tickets": the form (the method) defines the blanks, and each customer (each call) fills them in.

Reading a method header

The first line of a method is its **method header**. Here is the header of a method that prints the cost of some movie tickets:

```
1 public static void printTotal(double ticketPrice, int tickets)
```

Piece	What it is
<code>public static</code>	Explained in a later topic; read past them for now.
<code>void</code>	The return type : what kind of value the method gives back. <code>void</code> means "nothing." (1.9.7 explains this.)
<code>printTotal</code>	The method's name.
<code>(double ticketPrice, int tickets)</code>	The parameter list : the values the method needs, in order. It always has parentheses, even when empty, as in <code>printReminder()</code> .
<code>double ticketPrice</code>	One parameter : a variable declared in the header, with a type (<code>double</code>) and a name (<code>ticketPrice</code>). The method's body can use it like any other variable.
<code>int tickets</code>	A second parameter, of type <code>int</code> . Parameters are separated by commas.

A **parameter** is a variable declared in the header of a method. It can be used inside the method's body, and it is how values get into the method. (Java has one other kind of header with parameters, belonging to a constructor; you will meet constructors in Topic 1.13.)

An **argument** is a value passed into a method when the method is called. In the call `printTotal(12.5, 3)`, the arguments are 12.5 and 3. When the call happens, each parameter starts out holding the value of the matching argument: the first parameter gets the first argument, the second gets the second, and so on.

KEY IDEA

Parameters are the named variables in a method's header; arguments are the actual values in a call. When the method is called, the first parameter gets the first argument, the second gets the second, and so on.

WORKED EXAMPLE Filling in the parameters

TicketTotal.java

```
1 public class TicketTotal
2 {
3     public static void main(String[] args)
4     {
5         printTotal(12.5, 3);
6         printTotal(8.0, 2);
7     }
8
9     public static void printTotal(double ticketPrice, int tickets)
10    {
11        System.out.println(ticketPrice * tickets);
12    }
13 }
```

`printTotal` is called twice. Each call gives its parameters new starting values:

Call (line)	ticketPrice	tickets	Printed
printTotal(12.5, 3) (5)	12.5	3	37.5 (12.5 × 3)
printTotal(8.0, 2) (6)	8.0	2	16.0 (8.0 × 2)

On line 5, control jumps into `printTotal`, `ticketPrice` starts as 12.5, and `tickets` starts as 3. Line 11 multiplies them; a `double` times an `int` gives a `double` (Topic 1.3), so it prints 37.5. The method finishes, control returns to line 6, and the second call starts the method fresh with 8.0 and 2. The method's code never changed; only the values passed in did.

Now change what the arguments look like. An argument does not have to be a literal number. It can be a variable or a whole expression, and its name has nothing to do with the parameter's name.

`TicketVariables.java`

```

1 public class TicketVariables
2 {
3     public static void main(String[] args)
4     {
5         double matinee = 9.5;
6         int groupSize = 4;
7         printTotal(matinee, groupSize);
8         printTotal(matinee, groupSize + 1);
9     }
10
11     public static void printTotal(double ticketPrice, int tickets)
12     {
13         System.out.println(ticketPrice * tickets);
14     }
15 }
```

```

38.0
47.5
```

On line 7, Java looks up the values of `matinee` (9.5) and `groupSize` (4) and passes those values; inside the method they are known as `ticketPrice` and `tickets`. On line 8, Java first evaluates `groupSize + 1` to get 5, and passes 5. So $9.5 \times 4 = 38.0$ and $9.5 \times 5 = 47.5$. What gets passed is always a **value**; matching is done by position, never by name.

WATCH OUT

Using a parameter outside its method. A parameter belongs to its method's block. Students sometimes expect `tickets` to still exist in `main` after the call:

ParameterOutside.java

```
1 public class ParameterOutside
2 {
3     public static void main(String[] args)
4     {
5         printTotal(12.5, 3);
6         System.out.println(tickets);
7     }
8
9     public static void printTotal(double ticketPrice, int tickets)
10    {
11        System.out.println(ticketPrice * tickets);
12    }
13 }
```

Java reports ParameterOutside.java:6: error: cannot find symbol with symbol: variable tickets. The parameter tickets can be used only inside printTotal's body. main never declared a variable named tickets, so to main that name means nothing. If main needs the value, main must keep it in its own variable.

PAUSE AND TRY

Question 7 Multiple choice

Consider this method header.

```
1 public static void printArea(double length, double width)
```

Which statement is true?

- A. printArea is a parameter, and length and width are arguments.
- B. length and width are parameters; the values written in a call such as printArea(4.0, 2.5) are arguments.
- C. The header has one parameter, double length, double width.
- D. void is the name of the method.

Question 8 Tracing

What does this program print? Give the starting value of each parameter.

ProduceCost.java

```

1  public class ProduceCost
2  {
3      public static void main(String[] args)
4      {
5          int boxes = 3;
6          printCost(2.0, boxes + 1);
7      }
8
9      public static void printCost(double pricePerPound, int pounds)
10     {
11         System.out.println(pricePerPound * pounds);
12     }
13 }

```

1.9.5 Method Signatures

When you write `Math.sqrt(49.0)`, how does Java know which method you mean, and whether your arguments are acceptable? It cannot read your mind or the method's documentation comment. It needs a short, exact description of each method that it can compare against your call.

That description is the **method signature**: the method's name together with the ordered list of its parameter types. For a method with no parameters, the signature is the name with an empty parameter list. To write a signature, start from the header and keep only the name and the parameter types, in order:

Header (from the documentation)	Signature
<code>public static double sqrt(double a)</code>	<code>sqrt(double)</code>
<code>public static double pow(double a, double b)</code>	<code>pow(double, double)</code>
<code>public static void printTotal(double ticketPrice, int tickets)</code>	<code>printTotal(double, int)</code>
<code>int length()</code> (from the <code>String</code> documentation)	<code>length()</code>
<code>int indexOf(String str)</code> (from the <code>String</code> documentation)	<code>indexOf(String)</code>

Notice what the signature leaves out: the return type, the parameter names, and `public static`. Why? Because a call does not show any of them. In `printTotal(12.5, 3)` you can see the name and the types of the values (a `double`, then an `int`), and nothing else. The signature holds exactly what Java can check against a call.

Arguments must match the signature

For a call to compile, its arguments must be **compatible in number and order** with the parameter types in the signature:

- **Number**: one argument for each parameter, no more and no fewer.
- **Order**: the first argument must fit the first parameter's type, the second argument the second type, and so on.

- **Compatible type:** each argument must be a value that can be stored in that parameter. An `int` argument may go into a `double` parameter, because Java widens it automatically (Topic 1.5). A `double` argument may not go into an `int` parameter, because that could lose the decimal part.

KEY IDEA

A method signature is the method's name plus the ordered list of its parameter types, such as `pow(double, double)`. A call's arguments must match it in number, order, and type.

WORKED EXAMPLE Reading the documentation for `pow`

The documentation for `pow` says: "Returns the value of the first argument raised to the power of the second argument." Its signature is `pow(double, double)`. So the order of the arguments carries meaning: the first is the base, the second is the exponent.

```
1 System.out.println(Math.pow(2.0, 5.0));
2 System.out.println(Math.pow(5.0, 2.0));
```

```
32.0
25.0
```

Both calls match the signature: two arguments, both `double`. But line 1 means 2 to the 5th power ($2 \times 2 \times 2 \times 2 \times 2 = 32$), and line 2 means 5 to the 2nd power ($5 \times 5 = 25$). Same method, same two numbers, different order, different answer. Java matches arguments to parameters by position, so the documentation's order is the order you must use.

Now change the argument's type. `sqrt`'s parameter is a `double`, but here the argument is the `int` value 49:

```
1 System.out.println(Math.sqrt(49));
```

```
7.0
```

This compiles and runs. The `int` 49 is compatible with the `double` parameter, so Java widens it to 49.0 before the method starts. The rule is "compatible with" the parameter type, not "exactly the same as."

Here is what happens when a call does not match the signature. Each of these fails to compile:

Call	Problem	Main part of the compiler message
<code>Math.sqrt(9.0, 16.0)</code>	Wrong number: two arguments, one parameter	method sqrt in class Math cannot be applied to given types;
<code>Math.sqrt("49")</code>	Wrong type: text cannot be stored in a double	incompatible types: String cannot be converted to double

For a wrong number of arguments, the compiler also lists `required: double` and `found: double, double`: it is comparing the call to the signature, exactly as you would.

WATCH OUT

Putting arguments in the wrong order. Students remember *which* values a method needs but not their order. The signature is `printTotal(double, int)`, but this call passes the ticket count first:

TicketSwapped.java

```
1 public class TicketSwapped
2 {
3     public static void main(String[] args)
4     {
5         printTotal(3, 12.5);
6     }
7
8     public static void printTotal(double ticketPrice, int tickets)
9     {
10        System.out.println(ticketPrice * tickets);
11    }
12 }
```

Java reports

TicketSwapped.java:5: error: incompatible types: possible lossy conversion from double to int

Matching by position, the first argument 3 goes to ticketPrice (an int into a double is fine), but the second argument 12.5 goes to tickets, an int. Storing 12.5 in an int would lose the .5, so Java refuses. Fix it by following the signature's order: `printTotal(12.5, 3);`.

Be careful: if both parameters had the same type, swapping the arguments would compile and silently give the wrong answer, as `pow` showed. Always check the documentation for the order.

PAUSE AND TRY**Question 9** Short answer

Write the signature for each header from the `String` and `Math` documentation.

1. `String substring(int beginIndex, int endIndex)`
2. `public static int abs(int a)`
3. `int length()`

Question 10 Debugging

This program should print the cost of 2 pounds of apples at 1.5 dollars per pound, but it does not compile. Explain the error and write the corrected call.

ProduceSwap.java

```
1 public class ProduceSwap
2 {
3     public static void main(String[] args)
4     {
5         printCost(2, 1.5);
6     }
7
8     public static void printCost(double pricePerPound, int pounds)
9     {
10        System.out.println(pricePerPound * pounds);
11    }
12 }
```

1.9.6 Call by Value

Here is a reasonable thing to try: write a method that adds 10 bonus points to a score, pass it your score variable, and expect your variable to go up by 10. Does it? To answer that, you need to know exactly what Java hands to a method when you pass a variable as an argument: the variable itself, or something else?

Building on 1.9.4: a parameter starts out holding the value of its argument, and an argument can be a variable.

In plain words: when you pass a variable, Java does not give the method your variable. It reads the value stored in your variable and gives the method a **copy** of that value. It is like photocopying your homework for a friend: if your friend scribbles on the copy, your original is untouched.

Java passes arguments using **call by value**: each parameter is initialized with a copy of its argument's value. The parameter is a separate variable. Changing it inside the method changes only the copy.

KEY IDEA

With call by value, a parameter gets a copy of the argument's value. Changing the parameter inside the method does not change the caller's variable.

WORKED EXAMPLE The bonus that did not stick

Predict what line 8 prints before you read on.

BonusPoints.java

```

1 public class BonusPoints
2 {
3     public static void main(String[] args)
4     {
5         int myScore = 80;
6         addBonus(myScore);
7         System.out.print("Back in main: ");
8         System.out.println(myScore);
9     }
10
11    public static void addBonus(int score)
12    {
13        score += 10;
14        System.out.print("Inside addBonus: ");
15        System.out.println(score);
16    }
17 }

```

There are two separate variables here: `myScore`, which belongs to `main`, and `score`, which belongs to `addBonus`. Follow both:

Line	What happens	myScore (in main)	score (in addBonus)
5	<code>myScore</code> is declared and set to 80	80	(does not exist yet)
6	Call: a copy of 80 goes into the new parameter <code>score</code>	80	80
13	<code>score += 10</code> changes the copy	80	90
14–15	Prints <code>Inside addBonus: 90</code>	80	90
7–8	Method finished; back in <code>main</code> , prints <code>Back in main: 80</code>	80	(gone)

```

Inside addBonus: 90
Back in main: 80

```

Line 13 changed `score`, the copy. `myScore` was never touched, so it still holds 80. When the method finished, its parameter went away along with the 90.

Now change one thing: give `main`'s variable the **same name** as the parameter. Surely now they are the same variable?

SameName.java

```
1 public class SameName
2 {
3     public static void main(String[] args)
4     {
5         int score = 80;
6         addBonus(score);
7         System.out.print("Back in main: ");
8         System.out.println(score);
9     }
10
11    public static void addBonus(int score)
12    {
13        score += 10;
14        System.out.print("Inside addBonus: ");
15        System.out.println(score);
16    }
17 }
```

```
Inside addBonus: 90
Back in main: 80
```

The output is identical. There are still two variables that happen to share a name: one declared on line 5 in `main`'s block, and one declared in `addBonus`'s header on line 11. Each method sees only its own. The call on line 6 copies the value 80 from one to the other, and that is the only connection between them.

WATCH OUT

"The method changed its parameter, so my variable changed too." This is the most common mistake with methods. Students picture the argument variable itself traveling into the method. It does not; only a copy of its value does. In `BonusPoints.java`, expecting `Back in main: 90` is this mistake. If you need a method to produce a new value for the caller, the method has to **give a value back**, and the caller has to store it. That is the subject of 1.9.7.

PAUSE AND TRY**Question 11** Tracing

What does this program print? Track `main`'s `laps` and the parameter `laps` separately.

Laps.java

```
1 public class Laps
2 {
3     public static void main(String[] args)
4     {
5         int laps = 5;
6         train(laps);
7         laps += 1;
8         System.out.println(laps);
9     }
10
11     public static void train(int laps)
12     {
13         laps *= 2;
14         System.out.print("Trained: ");
15         System.out.println(laps);
16     }
17 }
```

Question 12 Multiple choice

Which statement best describes how Java passes an `int` variable as an argument?

- A. The method receives the caller's variable itself, so any change to the parameter changes the caller's variable.
- B. The method receives a copy of the variable's value; the parameter is a separate variable.
- C. The method receives the variable's name and looks up its value each time the parameter is used.
- D. The caller's variable is reset to 0 after the method finishes.

1.9.7 Return Values

Some methods do something you can see, like printing. Others work out an answer that the caller needs to keep using: `Math.sqrt` computes a square root, and the program usually needs that number for something else. As 1.9.6 showed, a method cannot sneak an answer back by changing its parameter. So how does an answer get from the method back to the caller?

In plain words: some methods are like a doorbell: you press it, something happens, and you get nothing to hold. Others are like a calculator: you put numbers in and get a number back, and what you do with that number is up to you.

The **return type** in a method's header (the word just before the name) says which kind the method is.

- A **void method** has the return type `void`. It does not give back any value. `printTotal`, `printReminder`, and `System.out.println` are void methods. Because there is no value, a call to a void method is used **only as a statement on its own**, never as part of an expression.
- A **non-void method** has any other return type, such as `int`, `double`, or `String`. It gives back a value, called its **return value**, of exactly that type. `sqrt` and `pow` return a `double`.

Inside a non-void method, a **return statement** such as `return hours * 60;` does two things: it sends the value back to the caller, and it ends the method, sending control back to the point of the call (1.9.3). Java then uses the returned value **in place of the call**, as if the call had been replaced by the value.

To use a return value, the caller must either **store it in a variable** (of a compatible type) or **use it as part of an expression**. Otherwise the value is simply thrown away.

KEY IDEA

A void method gives nothing back, so its call stands alone as a statement. A non-void method returns a value of its return type, and the caller must store that value or use it in an expression.

WORKED EXAMPLE Storing a return value and using one in an expression

`abs` is another `Math` method; its header is `public static int abs(int a)`, and it returns the **absolute value** of `a`: its distance from zero, so the number without its minus sign.

```
1 double side = Math.sqrt(81.0);
2 System.out.println(side);
3 System.out.println(Math.abs(-15) + 5);
```

```
9.0
20
```

Watch each call get replaced by its value:

Line	Statement as written	After the call returns	Result
1	<code>double side = Math.sqrt(81.0);</code>	<code>double side = 9.0;</code>	Stored: side holds 9.0
2	<code>System.out.println(side);</code>	(no call to replace)	Prints 9.0
3	<code>System.out.println(Math.abs(System.out.println(15 + 5));</code>	<code>5);</code>	Used in an expression: prints 20

Line 1 stores the return value. Line 3 never stores it; the value 15 is used right away in the addition. Both are correct ways to use a return value.

Now look at a non-void method from the inside, with its `return` statement, and watch the flow of control. `minutesIn` is a method written for this example.

MinutesReturn.java

```
1 public class MinutesReturn
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Planning practice...");
6         int total = minutesIn(2) + 15;
7         System.out.println(total);
8     }
9
10    public static int minutesIn(int hours)
11    {
12        System.out.println("Converting hours to minutes");
13        return hours * 60;
14    }
15 }
```

```
Planning practice...
Converting hours to minutes
135
```

1. Line 5 prints the first line.
2. Line 6 starts evaluating `minutesIn(2) + 15`. Before it can add, it needs the value of `minutesIn(2)`, so control jumps into the method with `hours = 2`.
3. Line 12 prints "Converting hours to minutes".
4. Line 13 computes $2 \times 60 = 120$ and returns it. The method ends right there.
5. Control returns to the middle of line 6, exactly where the call was. The call is replaced by 120, so Java finishes $120 + 15 = 135$ and stores it in `total`.
6. Line 7 prints 135.

The header promised an `int` (`public static int minutesIn`), and the returned value, 120, is an `int`. You are not expected to write methods like this yet; Unit 3 teaches that. For now, you need to read them.

WATCH OUT

Using a void method's call as a value. A void method gives nothing back, so there is nothing to store:

VoidInExpression.java

```
1 public class VoidInExpression
2 {
3     public static void main(String[] args)
4     {
5         double total = printTotal(12.5, 3);
6     }
7
8     public static void printTotal(double ticketPrice, int tickets)
9     {
10        System.out.println(ticketPrice * tickets);
11    }
12 }
```

Java reports

VoidInExpression.java:5: error: incompatible types: void cannot be converted to double.
printTotal prints the total but does not return it. Call it as a statement on its own:
printTotal(12.5, 3);.

Calling a non-void method and ignoring the result. The opposite mistake compiles, which makes it sneakier:

```
1 int temperature = -8;  
2 Math.abs(temperature);  
3 System.out.println(temperature);
```

This prints -8. `Math.abs(temperature)` did return 8, but nothing stored or used it, so the 8 was thrown away. And because of call by value (1.9.6), `abs` could not change `temperature` itself. The fix is to store the return value:

```
1 int temperature = -8;  
2 int degreesBelowZero = Math.abs(temperature);  
3 System.out.println(degreesBelowZero);
```

This prints 8.

PAUSE AND TRY

Question 13 Tracing

What does this program print?

PizzaOrder.java

```
1 public class PizzaOrder  
2 {  
3     public static void main(String[] args)  
4     {  
5         int slices = slicesFor(3) + 1;  
6         System.out.println(slices);  
7     }  
8  
9     public static int slicesFor(int people)  
10    {  
11        System.out.print("Ordering for ");  
12        System.out.println(people);  
13        return people * 2;  
14    }  
15 }
```

Question 14 Multiple choice

A program contains the method with header

`public static void printTotal(double ticketPrice, int tickets)`. Which statement does **not** compile?

A. `double root = Math.sqrt(64.0);`

- B. `printTotal(9.5, 2);`
- C. `System.out.println(Math.sqrt(64.0) * 2);`
- D. `System.out.println(printTotal(9.5, 2));`

1.9.8 Overloaded Methods

You know that `Math.abs(-15)` returns 15, an `int`. But you can also find the absolute value of `-7.5`, and the answer, 7.5, is a `double`. One method cannot have a parameter that is both an `int` and a `double`, and it cannot return two different types. So how can `abs` handle both?

The answer: there are two methods named `abs`. The `Math` documentation lists both:

```
1 public static int abs(int a)
2 public static double abs(double a)
```

Their signatures are `abs(int)` and `abs(double)`. The names match, but the parameter types differ, so the signatures are different. Methods are **overloaded** when there are several methods with the same name but different signatures.

When you call an overloaded method, Java compares your arguments with each signature and runs the one whose parameter list matches. This is why 1.9.5 defined a signature as name plus parameter types: the name alone is not enough to pick a method, but the name plus the types of the arguments is.

You have used an overloaded method since Topic 1.1 without knowing it. `System.out.println` has many versions, including `println(int)`, `println(double)`, `println(String)`, and `println()` with no parameters, which is why it accepts whatever you give it.

KEY IDEA

Overloaded methods share a name but have different signatures. Java picks the version whose parameter types match the call's arguments.

WORKED EXAMPLE Which abs runs?

```
1 System.out.println(Math.abs(-7));
2 System.out.println(Math.abs(-7.5));
3 System.out.println(Math.abs(-7.0));
```

```
7
7.5
7.0
```

Line	Argument (type)	Version chosen	Returns
1	-7 (int)	<code>abs(int)</code>	7 (an int)
2	-7.5 (double)	<code>abs(double)</code>	7.5

3	-7.0 (double)	abs(double)	7.0 (a double)
---	---------------	-------------	----------------

Lines 1 and 3 have the same number value, but `-7` is an `int` literal and `-7.0` is a `double` literal, so Java picks different versions. The `.0` in the output shows that line 3's result came back as a `double`.

Overloaded methods can also differ in **how many** parameters they have. The `String` documentation lists two `substring` methods, which give back part of a piece of text:

```
1 String substring(int beginIndex)
2 String substring(int beginIndex, int endIndex)
```

Their signatures are `substring(int)` and `substring(int, int)`. A call with one `int` argument runs the first; a call with two runs the second. (A call to a `String` method is written after a piece of text, like `word.substring(2, 5)`; Topic 1.14 teaches those calls, and Topic 1.15 teaches what `substring` does in detail.) Here is the same idea in a program you can trace, with two methods written for this example:

Receipt.java

```
1 public class Receipt
2 {
3     public static void main(String[] args)
4     {
5         printTotal(4.25);
6         printTotal(4.25, 2);
7     }
8
9     public static void printTotal(double price)
10    {
11        System.out.println(price);
12    }
13
14    public static void printTotal(double price, int quantity)
15    {
16        System.out.println(price * quantity);
17    }
18 }
```

```
4.25
8.5
```

Line 5 has one `double` argument, matching `printTotal(double)` on line 9. Line 6 has a `double` and an `int`, matching `printTotal(double, int)` on line 14, which prints $4.25 \times 2 = 8.5$.

WATCH OUT

Trying to overload by return type or parameter name. A signature contains only the name and the parameter types. Two methods that differ only in return type, or only in parameter names, have the same signature, and Java cannot tell them apart:

SameSignature.java

```
1 public class SameSignature
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(half(9));
6     }
7
8     public static int half(int number)
9     {
10        return number / 2;
11    }
12
13    public static double half(int value)
14    {
15        return value / 2.0;
16    }
17 }
```

SameSignature.java:13: error: method half(int) is already defined in class SameSignature

Both methods have the signature `half(int)`. In the call `half(9)`, nothing tells Java which one you want, because a call shows neither the return type nor the parameter names. To overload, change the number, types, or order of the parameters.

PAUSE AND TRY

Question 15 Multiple choice

A program calls `substring` on a piece of text with the arguments `(2, 5)`. Which header from the `String` documentation describes the method that runs?

- A. `String substring(int beginIndex)`
- B. `String substring(int beginIndex, int endIndex)`
- C. `int indexOf(String str)`
- D. `int length()`

Question 16 Multiple choice

Which pair of method headers can both appear in the same class?

- A. `public static void show(int a)` and `public static void show(int b)`
- B. `public static int show(int a)` and `public static double show(int a)`
- C. `public static void show(int a)` and `public static void show(double a)`
- D. `public static void show(int a, double b)` and `public static void show(int x, double y)`

Putting it together

Here is a program that uses every idea from this chapter. A teacher adds a 5-point curve to a quiz score, prints the original and the curved score, and then works out how far the curved score is from a goal of 90.

QuizReport.java

```

1 public class QuizReport
2 {
3     public static void main(String[] args)
4     {
5         int quiz = 78;
6         int curved = addCurve(quiz, 5);
7         printScore(quiz);
8         printScore(curved);
9         int gap = Math.abs(curved - 90);
10        System.out.print("Points from goal: ");
11        System.out.println(gap);
12    }
13
14    public static int addCurve(int score, int points)
15    {
16        score += points;
17        return score;
18    }
19
20    public static void printScore(int score)
21    {
22        System.out.print("Score: ");
23        System.out.println(score);
24    }
25 }

```

First, read the headers like documentation:

Header	Signature	Kind
public static int addCurve(int score, int points)	addCurve(int, int)	Non-void, returns an int
public static void printScore(int score)	printScore(int)	Void
public static int abs(int a) (in Math)	abs(int)	Non-void, returns an int

Now trace the flow of control. `addCurve` and `printScore` each have their own parameter named `score`; those are two separate variables, and neither one is `quiz`.

Line(s)	What happens	quiz	curved
5	quiz set to 78	78	
6 → 16 → 17	Call <code>addCurve(78, 5)</code> : copies go into <code>score</code> (78) and <code>points</code> (5). <code>score += points</code> makes the copy 83, and 83 is returned	78	
6	Back in line 6, the call is replaced by 83, which is stored	78	83
7 → 22–23	Call <code>printScore(78)</code> , a void method used as a statement. Prints Score: 78	78	83
8 → 22–23	Call <code>printScore(83)</code> . Prints Score: 83	78	83
9	<code>curved - 90</code> is -7; <code>Math.abs(-7)</code> returns 7, which is stored in <code>gap</code>	78	83

10-11	Prints Points from goal: 7	78	83
-------	----------------------------	----	----

```
Score: 78
Score: 83
Points from goal: 7
```

What each idea contributed:

- **Methods and flow of control (1.9.1, 1.9.3):** `addCurve` and `printScore` run only when called, and after each one, control returns to the line it was called from.
- **Procedural abstraction (1.9.2):** line 9 uses `Math.abs` knowing only what it does.
- **Parameters, arguments, and signatures (1.9.4, 1.9.5):** each call passes values that match its signature in number, order, and type.
- **Call by value (1.9.6):** `addCurve` changed its parameter `score` to 83, yet `quiz` still printed 78.
- **Return values (1.9.7):** the curved score reached `main` only because `addCurve` returned it and line 6 stored it. `printScore` is void, so it is called as a statement.
- **Overloading (1.9.8):** `abs(int)` is chosen because `curved - 90` is an `int`.

Common mistakes

The Watch Out boxes explain each mistake. Use this table to look up a compiler message you see while working with methods.

Compiler message (main part)	Usual cause	Fix
not a statement	Method name written without parentheses	Add <code>()</code> , with any arguments inside
method ... cannot be applied to given types;	Wrong number of arguments (or arguments that match no version)	Compare the call to the signature
incompatible types: possible lossy conversion from double to int	A double argument passed to an int parameter, often from swapped order	Pass arguments in the signature's order
incompatible types: void cannot be converted to ...	A void method's call stored in a variable	Call the void method as a statement on its own
'void' type not allowed here	A void method's call used inside another expression or call	Call the void method as a statement on its own
cannot find symbol (variable)	A parameter used outside its own method	Keep the value in a variable of the caller
method ... is already defined in class ...	Two methods with the same signature	Change the number, types, or order of the parameters

One mistake has no message at all: calling a non-void method and ignoring its return value. The program compiles and runs, but the answer is lost. Store it or use it in an expression.

Vocabulary

Term	Definition	Example
block	Any section of code enclosed in braces	The body of <code>main</code>
method	A named block of code that runs only when it is called	<code>printReminder</code>
call (a method)	Tell Java to run a method's code now, by writing its name and parentheses	<code>printReminder();</code>
procedural abstraction	Using a method by knowing what it does, even without knowing how it was written	Using <code>Math.sqrt</code> from its documentation
flow of control	The order in which statements actually run	<code>main</code> → method → back to <code>main</code>
method header	The first line of a method: return type, name, and parameter list	<code>public static double sqrt(double a)</code>
return type	The type of value a method gives back, written before its name; <code>void</code> means none	<code>double</code> in the <code>sqrt</code> header
parameter list	The parentheses in a header, holding the method's parameters in order	<code>(double ticketPrice, int tickets)</code>
parameter	A variable declared in a method's header that the method's body can use	<code>ticketPrice</code>
argument	A value passed into a method when it is called	<code>12.5</code> in <code>printTotal(12.5, 3)</code>
method signature	The method's name and the ordered list of its parameter types	<code>pow(double, double)</code>
call by value	Passing arguments by giving each parameter a copy of the argument's value	<code>addBonus</code> cannot change <code>myScore</code>
void method	A method that returns no value; called only as a statement on its own	<code>System.out.println</code>
non-void method	A method that returns a value of its return type	<code>Math.abs</code>
return value	The value a non-void method gives back to its caller	<code>15</code> from <code>Math.abs(-15)</code>
return statement	A statement that ends a method and sends a value back to the caller	<code>return hours * 60;</code>
absolute value	A number's distance from zero	<code>Math.abs(-8)</code> returns <code>8</code>
overloaded	Describes several methods with the same name but different signatures	<code>abs(int)</code> and <code>abs(double)</code>

Chapter summary

- A **method** is a named block of code that runs only when it is called. Thanks to **procedural abstraction**, you can use a method from its documentation, knowing what it does without knowing how it was written.
- A method call interrupts the normal top-to-bottom order: Java runs the method's statements, then returns to the point immediately after the call (or to the middle of the expression the call is in).
- A **method header** lists the return type, the name, and the **parameters**. A **method signature** is just the name and the ordered parameter types, and a call's **arguments** must match it in number, order, and type.
- Java uses **call by value**: each parameter gets a copy of its argument, so changing a parameter never changes the caller's variable.
- A **void method** returns nothing and is called as a statement on its own. A **non-void method** returns a value of its return type, which the caller must store or use in an expression.
- Methods are **overloaded** when they share a name but have different signatures; Java picks the version whose parameter types match the arguments.

Review questions

Question 17 Multiple choice

Here is part of the `Math` documentation:

- `int abs(int a)`: returns the absolute value of `a`
- `double pow(double a, double b)`: returns `a` raised to the power `b`
- `double sqrt(double a)`: returns the square root of `a`; precondition: `a` is not negative

Two players scored 72 and 85. Which expression gives how far apart the scores are as a positive number, without the programmer having to know which score is larger?

- A. `Math.abs(72) - 85`
- B. `Math.abs(72 - 85)`
- C. `Math.sqrt(72 - 85)`
- D. `Math.pow(72 - 85, 2)`

Question 18 Tracing

What does this program print? Watch the flow of control, the copies, and the returned value.

Bakery.java

```
1 public class Bakery
2 {
3     public static void main(String[] args)
4     {
5         int muffins = 4;
6         int boxes = boxesNeeded(muffins);
7         System.out.println(muffins);
8         System.out.println(boxes);
9         printLabel(boxes);
10    }
11
12    public static void printThanks()
13    {
14        System.out.println("Thank you!");
15    }
16
17    public static int boxesNeeded(int count)
18    {
19        count += 2;
20        System.out.println("Packing");
21        return count / 3;
22    }
23
24    public static void printLabel(int boxes)
25    {
26        System.out.print("Boxes: ");
27        System.out.println(boxes);
28    }
29 }
```

Question 19 Debugging

This program should print the cost of 4 muffins, 6.0. It compiles, but it prints 0.0. Find the error and write the corrected line.

MuffinCost.java

```
1 public class MuffinCost
2 {
3     public static void main(String[] args)
4     {
5         double total = 0.0;
6         costFor(4);
7         System.out.println(total);
8     }
9
10    public static double costFor(int muffins)
11    {
12        return muffins * 1.5;
13    }
14 }
```

Question 20 Short answer

A class contains both of these methods:

```
1 public static void printTotal(double price, int quantity)
2 public static void printTotal(int quantity, double price)
```

(a) Is this allowed? Explain using signatures. (b) Which method runs for the call `printTotal(3, 9.5)`, and why?

Question 21 Multiple choice

What are the values of `a` and `b` after this code runs?

TripleIt.java

```
1 public class TripleIt
2 {
3     public static void main(String[] args)
4     {
5         int a = 5;
6         int b = triple(a);
7         System.out.println(a);
8         System.out.println(b);
9     }
10
11     public static int triple(int n)
12     {
13         n *= 3;
14         return n;
15     }
16 }
```

- A. `a` is 15, `b` is 15
- B. `a` is 5, `b` is 15
- C. `a` is 5, `b` is 5
- D. `a` is 15, `b` is 5

Question 22 Multiple choice

A class contains a method with this header:

```
1 public static double average(int first, int second)
```

Which statement compiles?

- A. `double avg = average(80, 92);`
- B. `int avg = average(80, 92);`
- C. `double avg = average(80);`
- D. `double avg = average(80.5, 92);`

Question 23 Short answer

Many programs call `minutesIn(2)` from 1.9.7. A teammate wants to make two changes. (a) Rewrite the body so it returns `hours * 30 * 2` instead of `hours * 60`. (b) Change the header to `public static int minutesIn(int hours, int extraMinutes)`. For each change, say whether the existing calls `minutesIn(2)` still compile and give the same result, and explain why using procedural abstraction and signatures.

Reflection

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

- ☐ I can explain what a method and a block are, and why a method that is never called never runs. Reread: 1.9.1 Methods Are Named Blocks of Code
- ☐ I can explain how documentation lets me use a method without knowing how it was written. Reread: 1.9.2 Procedural Abstraction
- ☐ I can trace a program into a method and back to the point right after the call. Reread: 1.9.3 What Happens When a Method Is Called
- ☐ I can name the return type, name, and parameters in a header and match arguments to parameters. Reread: 1.9.4 Parameters and Arguments
- ☐ I can write a method signature and decide whether a call matches it in number, order, and type. Reread: 1.9.5 Method Signatures
- ☐ I can explain why changing a parameter does not change the caller's variable. Reread: 1.9.6 Call by Value
- ☐ I can tell void from non-void methods and use a return value by storing it or using it in an expression. Reread: 1.9.7 Return Values
- ☐ I can decide whether two methods are overloaded and predict which version a call runs. Reread: 1.9.8 Overloaded Methods

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. A method is a named block of code that runs only when it is called. Writing the method in the file just defines it.

- **A** is wrong: this is the mistake of reading a file top to bottom as if every line runs. `NeverCalled.java` contains a method that is never called, and its statement never runs.
- **C** is wrong: a method can be called as many times as you like; `ReminderTwice.java` calls `printReminder` twice. Reuse is a big reason methods exist.
- **D** is wrong: a block is a section of code enclosed in braces, not a single statement.

Reread *1.9.1 Methods Are Named Blocks of Code* if you chose differently.

Question 2

```
Stretch for five minutes.  
Run three laps.  
Stretch for five minutes.
```

`main` calls `printWarmUp` (line 5), prints its own line (line 6), then calls `printWarmUp` again (line 7). Each call runs the method's block once. `printCoolDown` is defined on lines 15–18 but never called, so "Walk one lap slowly." is never printed, even though it is the last method in the file.

Reread *1.9.1 Methods Are Named Blocks of Code* if your output differed.

Question 3

B. Procedural abstraction means using a method by knowing what it does (here, from its documentation) even though you do not know how it was written.

- **A** is wrong: it is the opposite of abstraction. Studying the inside is exactly what abstraction lets you skip.
- **C** is wrong: copying the steps means you are dealing with the *how* every time. A method lets you write it once and just call it.
- **D** is wrong: comments explain code but have nothing to do with using a method without seeing its code.

Reread *1.9.2 Procedural Abstraction* if you chose differently.

Question 4

Model answer: Procedural abstraction means you can use a method by knowing what it does, not how it was written. My classmate needs to know what the method does and how to use it: it gives back the square root of the value in its parentheses, the value must not be negative (its precondition), and the result is a `double`. All of that is in the documentation. How it calculates the answer is hidden on purpose.

A complete answer must:

- say that you only need to know *what* the method does, not *how*, and

- name the documentation (what goes in, what comes out, any precondition) as the source of that knowledge.

Reread 1.9.2 *Procedural Abstraction* if your answer differed.

Question 5

```
Board the bus.
See the dinosaurs.
Eat lunch.
Visit the gift shop.
Eat lunch.
Ride home.
```

Line running	Why	Printed
5	First statement of <code>main</code>	Board the bus.
6 → 13	<code>main</code> calls <code>visitMuseum</code>	See the dinosaurs.
14 → 20	<code>visitMuseum</code> calls <code>eatLunch</code>	Eat lunch.
15	<code>eatLunch</code> is done; control returns into <code>visitMuseum</code> , after line 14	Visit the gift shop.
7 → 20	<code>visitMuseum</code> is done; back in <code>main</code> , line 7 calls <code>eatLunch</code>	Eat lunch.
8	<code>eatLunch</code> is done; back in <code>main</code>	Ride home.

"Eat lunch." appears twice because `eatLunch` is called twice, once from inside `visitMuseum` and once from `main`.

Reread 1.9.3 *What Happens When a Method Is Called* if your output differed.

Question 6

B. Once a method finishes its last statement, Java goes back to the caller and picks up with the statement right after the call.

- **A** is wrong: the program does not start over. `main` was only paused at the call.
- **C** is wrong: the order of methods in the file does not control what runs. Only calls do.
- **D** is wrong: only the method is finished. The caller (`main`) still has statements left to run.

Reread 1.9.3 *What Happens When a Method Is Called* if you chose differently.

Question 7

B. Parameters are the variables declared in the header (`length` and `width`, both of type `double`). Arguments are the values supplied in a call.

- **A** is wrong: `printArea` is the method's name, and `length` and `width` are declared in the header, which makes them parameters, not arguments.
- **C** is wrong: each type-and-name pair is a separate parameter, and commas separate them, so there are two.
- **D** is wrong: `void` is the return type. The name is the word just before the parentheses, `printArea`.

Reread 1.9.4 *Parameters and Arguments* if you chose differently.

Question 8

8.0

On line 6, Java evaluates the arguments first: `2.0` is a literal, and `boxes + 1` is `3 + 1 = 4`. So `pricePerPound` starts as `2.0` and `pounds` starts as `4`. Line 11 prints `2.0 × 4 = 8.0`, a `double` because one operand is a `double`.

If you answered `6.0`, you passed `boxes` (`3`) instead of evaluating the expression `boxes + 1`. Reread *1.9.4 Parameters and Arguments* if your answer differed.

Question 9

1. `substring(int, int)`
2. `abs(int)`
3. `length()`

A signature is the method name followed by the parameter types in order, in parentheses. Leave out the return type (`String`, `int`), the parameter names (`beginIndex`, `endIndex`, `a`), and `public static`. A method with no parameters keeps its empty parentheses.

Reread *1.9.5 Method Signatures* if your signatures included anything else.

Question 10

Error: the arguments on line 5 are in the wrong order. The signature is `printCost(double, int)`, so the first argument is matched to `pricePerPound` and the second to `pounds`. The second argument, `1.5`, would have to be stored in the `int` parameter `pounds`, which would lose the decimal part. Java reports `incompatible types: possible lossy conversion from double to int`, and nothing runs.

Corrected call (line 5):

`ProduceFixed.java`

```
1 public class ProduceFixed
2 {
3     public static void main(String[] args)
4     {
5         printCost(1.5, 2);
6     }
7
8     public static void printCost(double pricePerPound, int pounds)
9     {
10        System.out.println(pricePerPound * pounds);
11    }
12 }
```

Now `1.5` goes to `pricePerPound`, `2` goes to `pounds`, and the program prints `3.0`.

Reread *1.9.5 Method Signatures* if you did not spot the order problem.

Question 11

Trained: 10
6

Line	laps in main	laps in train	Printed
5	5		
6	5	5 (a copy)	
13	5	10	
14-15	5	10	Trained: 10
7	6	(gone)	
8	6		6

The method doubled its own copy to 10. Back in `main`, `laps` was still 5, and line 7 added 1 to get 6. If you answered 11, you assumed the doubling changed `main`'s variable. The shared name does not make them the same variable.

Reread 1.9.6 *Call by Value* if your output differed.

Question 12

B. Java uses call by value: each parameter is initialized with a copy of its argument's value.

- **A** is wrong: this is the "same variable" misconception. `BonusPoints.java` shows the caller's variable unchanged after the method changes its parameter.
- **C** is wrong: names play no part in passing; arguments are matched by position, and only a value is passed. `TicketVariables.java` passes `groupSize` into a parameter named `tickets`.
- **D** is wrong: the caller's variable is untouched by the call. It keeps whatever value it had.

Reread 1.9.6 *Call by Value* if you chose differently.

Question 13

```
Ordering for 3
7
```

Line 5 needs the value of `slicesFor(3)` before it can add 1, so control jumps into `slicesFor` with `people = 3`. Lines 11–12 print `Ordering for 3`. Line 13 returns $3 \times 2 = 6$, and control returns to line 5, where the call is replaced by 6. Then $6 + 1 = 7$ is stored in `slices`, and line 6 prints 7.

If you answered 6, you dropped the `+ 1`: the return value is used in an expression, not just stored. If you printed "Ordering for 3" after 7, remember that the method's statements run during the call, before line 6.

Reread 1.9.7 *Return Values* if your output differed.

Question 14

D. `printTotal` is a void method, so it has no value to give to `println`. A void method's call must stand alone as a statement. Java reports 'void' type not allowed here.

- **A** is wrong: `sqrt` is non-void and returns a `double`, which is stored in a `double` variable. That compiles.
- **B** is wrong: this is the correct way to call a void method, as a statement on its own.
- **C** is wrong: the `double` returned by `sqrt` is used in an expression (times 2), which is allowed for a non-void method.

Reread 1.9.7 *Return Values* if you chose differently.

Question 15

B. Java chooses among overloaded methods by matching the arguments to each signature. Two `int` arguments match `substring(int, int)`.

- **A** is wrong: it has the right name, but its signature `substring(int)` takes one argument, not two.
- **C** is wrong: the name is different, and it takes one `String`, not two `int` values.
- **D** is wrong: the name is different, and it takes no arguments.

Reread 1.9.8 *Overloaded Methods* if you chose differently.

Question 16

C. The signatures are `show(int)` and `show(double)`. Same name, different parameter types, so the methods are overloaded, and a call such as `show(4)` or `show(4.0)` picks one clearly.

- **A** is wrong: both signatures are `show(int)`. Parameter names are not part of the signature, so Java reports that the method is already defined.
- **B** is wrong: both signatures are `show(int)`. The return type is not part of the signature either.
- **D** is wrong: both signatures are `show(int, double)`. Only the names changed.

Reread 1.9.8 *Overloaded Methods* if you chose differently.

Question 17

B. `72 - 85` is evaluated first, giving `-13`, and `abs` returns its distance from zero, `13`. Subtracting in the other order would give `13` too, so the programmer does not need to know which score is larger. This is procedural abstraction at work: the documentation alone tells you `abs` is the right tool.

- **A** is wrong: the argument is only `72`, so `abs` returns `72`, and then `85` is subtracted, giving `-13`. The subtraction has to be inside the parentheses to be part of the argument.
- **C** is wrong: the argument is `-13`, which breaks `sqrt`'s precondition. And a square root is not the distance between two numbers anyway.
- **D** is wrong: it returns `169.0`, the square of the difference, not the difference.

Reread 1.9.2 *Procedural Abstraction* and 1.9.7 *Return Values* if you chose differently.

Question 18

```
Packing
4
2
Boxes: 2
```

Line(s)	What happens	muffins	boxes (in main)
5	muffins set to 4	4	
6 → 19–21	Call <code>boxesNeeded(4)</code> : <code>count</code> gets a copy, <code>4</code> . <code>count += 2</code> makes it 6. Prints <code>Packing</code> . Returns <code>6 / 3 = 2</code>	4	
6	The call is replaced by <code>2</code> , which is stored in <code>boxes</code>	4	2
7	Prints <code>4</code> : <code>muffins</code> is unchanged, because only the copy <code>count</code> changed	4	2
8	Prints <code>2</code>	4	2
9 → 26–27	Calls the void method <code>printLabel</code> with <code>2</code> , which prints <code>Boxes: 2</code>	4	2

`printThanks` is never called, so "Thank you!" never prints, even though it is written before the other two methods. `Packing` prints first because the method's statements run during the call on line 6, before line 7.

Reread 1.9.3 *What Happens When a Method Is Called*, 1.9.6 *Call by Value*, or 1.9.7 *Return Values* for whichever line you got wrong.

Question 19

Error: line 6 calls the non-void method `costFor`, which returns $4 \times 1.5 = 6.0$, but the return value is neither stored nor used, so it is thrown away. `total` keeps its original 0.0. This is a logic error, so the compiler gives no message.

Corrected line 6:

MuffinCostFixed.java

```
1 public class MuffinCostFixed
2 {
3     public static void main(String[] args)
4     {
5         double total = 0.0;
6         total = costFor(4);
7         System.out.println(total);
8     }
9
10    public static double costFor(int muffins)
11    {
12        return muffins * 1.5;
13    }
14 }
```

Now the call is replaced by 6.0, which is stored in `total`, and line 7 prints 6.0. (Writing `double total = costFor(4);` on line 5 and deleting line 6 also works.)

Reread 1.9.7 *Return Values* if you missed the error.

Question 20

Model answer:

(a) Yes. The signatures are `printTotal(double, int)` and `printTotal(int, double)`. The parameter types are in a different order, so the signatures are different, and the methods are overloaded.

(b) The second one, `printTotal(int, double)`. This test program shows it by printing a different label from each version:

OrderOverload.java

```
1 public class OrderOverload
2 {
3     public static void main(String[] args)
4     {
5         printTotal(3, 9.5);
6     }
7
8     public static void printTotal(double price, int quantity)
9     {
10        System.out.println("Version 1");
11    }
12
13    public static void printTotal(int quantity, double price)
14    {
15        System.out.println("Version 2");
16    }
17 }
```

It prints `Version 2`. Java matches by position: the arguments are an `int` (3) then a `double` (9.5). The first version would need its second argument, 9.5, to go into an `int` parameter, which is not allowed. The second version fits: 3 into `int`, 9.5 into `double`.

A complete answer must:

- write both signatures and say that a different order of parameter types makes them different, and
- explain that 9.5 cannot go into an `int` parameter, so only the (`int`, `double`) version matches.

Reread *1.9.5 Method Signatures* and *1.9.8 Overloaded Methods* if your reasoning differed.

Question 21

B. The program prints 5 and then 15. On line 6, `n` gets a copy of `a`'s value, 5. Line 13 triples the copy to 15, and line 14 returns 15, which is stored in `b`. Because of call by value, `a` is untouched and stays 5.

- **A** is wrong: it assumes changing `n` also changed `a`. Only the copy changed.
- **C** is wrong: it assumes the tripling was lost. It changed `n`, and `n`'s value was returned and stored in `b`.
- **D** is wrong: it mixes up which variable got which value. `a` was never assigned again after line 5.

Reread *1.9.6 Call by Value* and *1.9.7 Return Values* if you chose differently.

Question 22

A. Two `int` arguments match the signature `average(int, int)`, and the returned `double` is stored in a `double` variable. (It stores 86.0.)

- **B** is wrong: the method returns a `double`, and a `double` cannot be stored in an `int` variable without a cast. Java reports possible lossy conversion from `double` to `int`.
- **C** is wrong: the signature needs two arguments and the call has one. Java reports that the method cannot be applied to given types.
- **D** is wrong: the first argument, 80.5, is a `double`, and the first parameter is an `int`. Java reports possible lossy conversion from `double` to `int`.

Reread *1.9.5 Method Signatures* and *1.9.7 Return Values* if you chose differently.

Question 23

Model answer:

(a) The calls still compile and give the same result. Callers depend only on *what* the method does (turn hours into minutes) and on its signature, `minutesIn(int)`. Neither changed. The *how* inside the body changed, and procedural abstraction means callers never depended on the how. Since $30 \times 2 = 60$, the answer is the same.

(b) The calls no longer compile. The signature becomes `minutesIn(int, int)`, so a call with one argument does not match it in number, and Java reports that the method cannot be applied to given types. Every call would need a second argument.

A complete answer must:

- say that changing only the body leaves callers unaffected because they rely on what the method does, not how, and
- say that changing the parameter list changes the signature, so calls with the old number of arguments no longer match.

Reread 1.9.2 *Procedural Abstraction* and 1.9.5 *Method Signatures* if your reasoning differed.