

1.1 Introduction to Algorithms, Programming, and Compilers

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Explain what an algorithm is and why its steps must be precise.
- Represent an everyday algorithm in numbered written steps and as a flowchart.
- Explain why the order of steps matters and predict what happens when steps are reordered.
- Describe how code goes from a text editor or IDE, through the compiler, to a running program.
- Tell syntax errors, logic errors, and run-time errors (exceptions) apart, and say when and how each one is found.

What to remember from earlier

This is the first topic of the course, so you do not need any earlier Java knowledge. You only need everyday experience of following instructions, such as a recipe, directions to a friend's house, or the steps to log in to a website. Keep one of those in mind as you read; this chapter keeps coming back to the idea of following instructions exactly.

1.1.1 What Is an Algorithm?

Imagine texting a friend directions to your house. If you write "come over," your friend is stuck. If you write "take the bus to Oak Street, walk two blocks north, and it's the blue door on the left," your friend can get there without asking you anything. The second message works because it is a list of clear steps that anyone can follow.

Computers need exactly that kind of message, and then some. A friend can guess what you meant when a step is fuzzy. A computer cannot guess. It does exactly what it is told, step by step, even when the result is silly. In this chapter, we will call this kind of reader a **literal follower**: someone who carries out every instruction exactly as written and fills in nothing. (It is our name for the idea, not an official term, but it describes a computer perfectly.)

So before anyone writes a line of Java, they need a plan written as clear, ordered steps. That plan has a name. An **algorithm** is a set of steps, done in order, that gets a job done or solves a problem. A recipe is an algorithm. So are the instructions for assembling a bookshelf, and so is the routine you follow to get ready for school.

KEY IDEA

An algorithm is a set of ordered steps that gets a job done or solves a problem. Its steps must be clear enough that a literal follower could carry them out without guessing.

Why do the steps have to be so precise? Because the person or machine following them is not allowed to fill gaps. "Add some salt" works for a cook who knows what "some" means. A literal follower does not know, so it might add none, or the whole box.

WORKED EXAMPLE Making a vague algorithm precise

Here is an algorithm for brushing your teeth, written the way most people would say it:

1. Put toothpaste on the brush.
2. Brush your teeth.
3. Rinse.

Read it as a literal follower would, one step at a time, and ask what is unclear:

Step	Question a literal follower would ask	Precise rewrite
1	How much toothpaste?	Put a pea-sized amount of toothpaste on the brush.
2	Which teeth? For how long?	Brush all of your teeth for 2 minutes.
3	Rinse what? With what?	Spit, then rinse your mouth and the brush with water.

Nothing about the task changed. The rewrite answers every question before it can be asked, so a literal follower could finish the task correctly.

Now change one thing: the reader. If the same three short steps were given to a seven-year-old, a parent might still need to explain them. If they were given to a robot, the robot would fail at step 1. The more literal the follower, the more precise the algorithm has to be. A computer is the most literal follower there is, which is why programming starts with precise algorithms.

WATCH OUT

"The computer will know what I mean." It will not. Students often leave out a step because it feels obvious, such as "open the toothpaste" or "turn on the water." A literal follower skips anything that is not written down, so a missing step means the task fails at that point. When you check an algorithm, read each step as if you knew nothing except what is written.

PAUSE AND TRY

Question 1 Multiple choice

Which of these is the best example of an algorithm?

- A. "Be careful on your way home."
- B. "Walk to the corner, turn left on Pine Street, and stop at the third house."
- C. "Home is about ten minutes away."
- D. "My house is the nicest one on the street."

Question 2 Short answer

The algorithm "Make toast" has two steps: (1) Put bread in the toaster. (2) Take out the toast. Name one step a literal follower would need that is missing, and explain why it matters.

1.1.2 Representing Algorithms: Words and Diagrams

You now know that an algorithm is a list of precise steps. But a list in your head is not useful to anyone else. To share an algorithm, or to check it, you have to write it down. How should you write it?

The two most common ways are words and pictures. **Written language** means writing each step as a sentence, usually as a numbered list so the order is obvious. A **diagram** shows the steps as a picture. The kind of diagram used most for algorithms is a **flowchart**: each step goes in a box, and arrows show which step comes next.

KEY IDEA

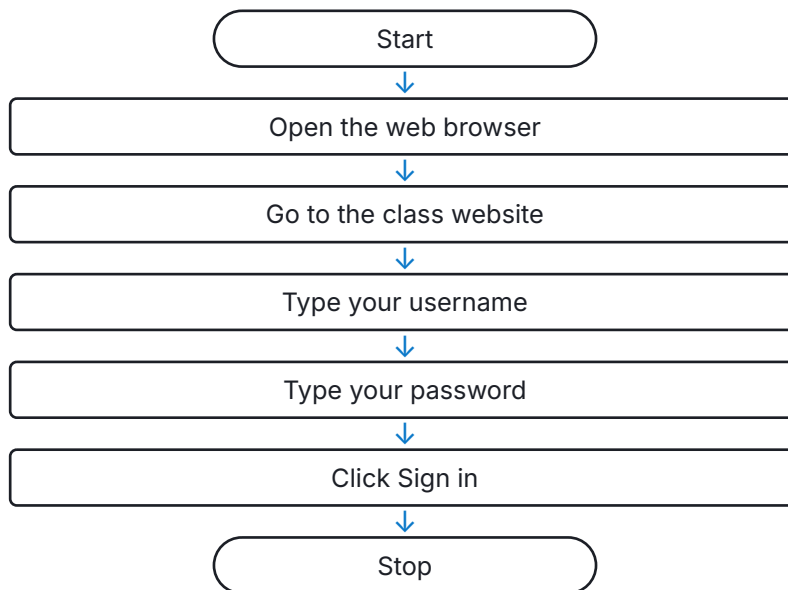
The same algorithm can be represented in written language (numbered steps) or as a diagram (such as a flowchart). Changing the representation does not change the algorithm, as long as the steps and their order stay the same.

WORKED EXAMPLE One algorithm, two representations

Here is an algorithm for logging in to a class website, written as numbered steps:

1. Open the web browser.
2. Go to the class website.
3. Type your username.
4. Type your password.
5. Click **Sign in**.

Here is the same algorithm as a flowchart. Each box holds one step, and each arrow points to the step that comes next. The oval shapes mark where the algorithm starts and stops.



The login algorithm as a flowchart.

Compare them step by step: box 1 matches step 1, box 2 matches step 2, and so on. Both have the same five steps in the same order, so they represent the **same algorithm**. The list is quicker to write. The flowchart makes the order easy to see at a glance, because you just follow the arrows.

Now vary one thing. Suppose someone draws the flowchart again but the arrow from "Type your username" points straight to "Click Sign in," skipping the password box. The boxes are all still there, but the arrows describe a different order, and the password is never typed. That is a different algorithm, and a broken one. In a flowchart, the arrows *are* the order.

WATCH OUT

"A diagram and a list are always different algorithms." Students sometimes think that changing from words to a picture creates a new algorithm. It does not. What defines the algorithm is its steps and their order. To decide whether two representations match, compare them step by step and check that the order is the same; ignore whether one uses sentences and the other uses boxes.

PAUSE AND TRY

Question 3 Tracing

Follow the arrows of the login flowchart above, starting at **Start**. Which step comes immediately after "Go to the class website," and how many steps (boxes, not counting Start and Stop) are completed before "Click Sign in"?

Question 4 Multiple choice

A flowchart shows these boxes connected by arrows in this order: *Fill the kettle* → *Turn on the kettle* → *Wait for the water to boil* → *Pour the water into a cup*. Which written algorithm represents the **same** algorithm?

- A. Turn on the kettle → Fill the kettle → Wait for the water to boil → Pour the water into a cup
- B. Fill the kettle → Turn on the kettle → Wait for the water to boil → Pour the water into a cup
- C. Fill the kettle → Wait for the water to boil → Turn on the kettle → Pour the water into a cup
- D. Fill the kettle → Turn on the kettle → Pour the water into a cup

1.1.3 Sequencing: Order Matters

In the last question, two of the wrong answers had exactly the right steps. They failed only because the steps were in the wrong order. That is worth a closer look, because it is one of the most important ideas in all of programming.

Why does order deserve a name of its own? Because getting the steps right is not enough; they must happen in the right order.

In plain words: you cannot frost a cake before you bake it. The steps are completed one after another, and each step happens in the situation the earlier steps left behind. **Sequencing** is the order in which the steps of a process are completed. In a sequence, steps are completed **one at a time**: the follower finishes one step completely before starting the next.

KEY IDEA

Sequencing defines the order in which steps are completed, and steps are completed one at a time. The same steps in a different order can produce a different result, or no useful result at all.

WORKED EXAMPLE Same steps, different order

Here is an algorithm for putting on shoes:

1. Put on your socks.
2. Put on your shoes.
3. Tie your laces.

Follow it one step at a time, and keep track of the situation after each step:

Step done	Situation afterward
1. Put on your socks.	Socks on, shoes off
2. Put on your shoes.	Socks on, shoes on (untied)
3. Tie your laces.	Socks on, shoes on and tied. Done!

Now swap steps 1 and 2 and follow it again:

Step done	Situation afterward
1. Put on your shoes.	Shoes on bare feet
2. Put on your socks.	Socks go over the shoes (or cannot go on)
3. Tie your laces.	Laces tied under the socks. Not what we wanted

Both versions use the same three steps. Only the order changed, and the result changed with it. Each step acts on the situation left by the steps before it, so a step that is correct in one position can be wrong in another.

Change one thing again: swap steps 2 and 3 in the original. You get "put on socks, tie laces, put on shoes." Step 2 now asks you to tie laces on shoes you are not wearing, so it cannot be completed at all. Some reorderings produce a wrong result; others make a step impossible.

Not every swap matters, though. If an algorithm says "1. Put a fork on the table. 2. Put a spoon on the table," swapping the two gives the same table. Order matters when a later step depends on what an earlier step did. Part of checking an algorithm is noticing which steps depend on which.

WATCH OUT

"If all the right steps are there, the algorithm is correct." Students often check only that nothing is missing. But the follower does the steps one at a time, in the order written, so a correct list of steps in the wrong order is still a wrong algorithm. Always check the order too: for each step, ask whether everything it needs was already done by an earlier step.

PAUSE AND TRY**Question 5** Multiple choice

An algorithm for mailing a letter is: 1. Seal the envelope. 2. Put the letter in the envelope. 3. Put a stamp on the envelope. 4. Drop it in the mailbox. What is wrong with it?

- A. A step is missing.
- B. It has too many steps.
- C. Steps 1 and 2 are in the wrong order.
- D. Nothing; the steps can be done in any order.

Question 6 Short answer

In the algorithm "1. Take out a bowl. 2. Take out a spoon. 3. Pour cereal into the bowl. 4. Pour milk into the bowl," which two steps could be swapped without changing the result? Explain why that swap is safe but swapping steps 1 and 3 is not.

1.1.4 From Algorithm to Program

So far every algorithm has been written for a person. But the goal of this course is to have a computer carry out algorithms. A computer cannot read "put on your socks," and it cannot read an English sentence at all. How do we write an algorithm so that a computer can follow it?

Building on 1.1.1 and 1.1.3: a computer is the ultimate literal follower, and it does steps one at a time, in order. So we need a way of writing steps that is completely precise and has a fixed order.

In plain words: we translate the algorithm into a special language that has strict rules, so that every instruction means exactly one thing. That language is a **programming language**, and this course uses the programming language **Java**. An algorithm written in a programming language is a **program**, and the text of the program that a person writes and reads is called **source code**, or just **code**.

Where code is written

Source code is plain text, so it **can be written in any text editor**, even a basic notes app. In practice, programmers almost always use an **integrated development environment**, or **IDE**. An IDE is one application that lets you **type your code, check and translate it, and start it running**, without switching between programs. It colors different parts of the code so they are easier to read, points out some mistakes as you type, and has a button to run your program. You will learn what "compile" means in 1.1.5.

KEY IDEA

Code can be written in any text editor, but programmers usually use an IDE because it puts the tools to write, compile, and run code in one place.

WORKED EXAMPLE Your first look at a Java program

Here is a complete Java program. You are not expected to write programs like this yet; for now, the goal is to read it and connect it to the idea of an algorithm.

Greeting.java

```

1 public class Greeting
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Good morning!");
6         System.out.println("Welcome to AP CSA.");
7     }
8 }

```

Most of the lines are a frame that every program in this course has. The lines that matter right now are lines 5 and 6:

Part of the code	What it means for now
<code>public class Greeting</code>	The program's name is <code>Greeting</code> . The file is saved as <code>Greeting.java</code> .
<code>public static void main(String[] args)</code>	The program starts running here. The words <code>public</code> and <code>static</code> are explained in a later topic; read past them for now.
<code>{ and }</code>	Curly braces group lines together, like the start and end of a section.
<code>System.out.println("...");</code>	Displays the text between the double quotes, then moves to a new line. Displaying output is taught fully in Topic 1.3.
<code>;</code>	Every statement ends with a semicolon, the way a sentence ends with a period.

Each line inside `main` is one **statement**: one instruction for the computer. The statements run one at a time, from top to bottom, exactly like the steps of an algorithm. So line 5 runs first, then line 6, and the program displays:

```

Good morning!
Welcome to AP CSA.

```

Now change one thing: the order of the statements. This version has the same greeting lines plus one more, with the first two swapped:

GreetingReordered.java

```

1 public class GreetingReordered
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Welcome to AP CSA.");
6         System.out.println("Good morning!");
7         System.out.println("Let us begin.");
8     }
9 }

```

Because statements run in sequence, the output follows the new order exactly:

```
Welcome to AP CSA.  
Good morning!  
Let us begin.
```

This is the same sequencing idea from 1.1.3, now in Java. The computer does not rearrange your statements into an order that makes more sense; it runs them in exactly the order you wrote them.

WATCH OUT

"The program reads like English, so the computer understands English." Code is designed to be readable by people, so it contains English words like `print`. But the computer only accepts code that follows Java's rules exactly. `System.out.println` works; "please display Good morning" does not. In 1.1.5 and 1.1.6 you will see what happens when code does not follow the rules.

PAUSE AND TRY

Question 7 Tracing

What does this program display? Write the output exactly, line by line.

LunchRoutine.java

```
1 public class LunchRoutine  
2 {  
3     public static void main(String[] args)  
4     {  
5         System.out.println("Wash your hands.");  
6         System.out.println("Unpack your lunch.");  
7         System.out.println("Eat.");  
8         System.out.println("Unpack your lunch.");  
9         System.out.println("Clean up.");  
10    }  
11 }
```

Question 8 Multiple choice

Which statement about writing code is true?

- A. Java code can only be written inside an IDE.
- B. Any text editor can hold code, but an IDE also lets you compile and run it in the same place.
- C. An IDE is a programming language, like Java.
- D. A text editor and an IDE both run programs automatically as you type.

1.1.5 The Compiler: Checking Before Running

There is a gap between the code you type and what the computer can actually do. The computer's processor does not understand words like `System.out.println`. Something has to check your Java and translate it into a form the computer can run. And because the computer is a literal follower, it

would be good if something caught certain mistakes **before** the program starts running and doing the wrong thing.

In plain words: before your program runs, a checker reads the whole program, makes sure it follows the language's rules, and, if it does, translates it into instructions the computer can carry out. If the checker finds a broken rule, it refuses to translate and tells you what is wrong. That checker is called the **compiler**. Running the compiler on your code is called **compiling**. Carrying out the translated instructions is called **running** (or **executing**) the program.

So every Java program goes through two separate stages, always in this order:

Stage	What happens	Who does it
Compile	The compiler checks the code for some errors, then translates it into a form the computer can run.	The compiler
Run	The computer carries out the translated statements one at a time, in order.	The computer

Here is the rule that matters most: **errors the compiler detects must be fixed before the program can run**. If the compiler finds even one, there is no translated program, so there is nothing to run. Notice the word *some* in the table: the compiler catches some kinds of mistakes, not all of them. You will meet the kinds it misses in 1.1.7 and 1.1.8.

KEY IDEA

A compiler checks code for some errors before the program runs. Errors the compiler detects must be fixed before the program can be run at all.

WORKED EXAMPLE A compiler error stops everything

Here is the greeting program again. Change one thing: the semicolon at the end of line 5 is missing.

MissingSemicolon.java

```
1 public class MissingSemicolon
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Good morning!")
6         System.out.println("Welcome to AP CSA.");
7     }
8 }
```

When you try to run this in an IDE, the compiler checks the code first. It reaches the end of line 5, expects a semicolon to end the statement, and does not find one. It reports:

```
MissingSemicolon.java:5: error: ';' expected
    System.out.println("Good morning!")
                        ^
1 error
```

Then it stops. Because there was an error, the code is never translated, so the program **never runs**. Nothing at all is displayed: not "Good morning!", and not "Welcome to AP CSA." either, even though line 6 is written correctly.

Add the semicolon back and the compiler finds no errors, translates the code, and the program runs and displays both lines, as in 1.1.4.

WATCH OUT

"The first line is fine, so it will print before the error." This is the most common misunderstanding about compiling. Students picture the computer running line 5, reaching the error, and stopping, with "Good morning!" already displayed. But compiling happens **before** running and covers the **whole** program. One compiler error anywhere means no part of the program runs. If you see a compiler error, expect an empty output and fix the error first.

PAUSE AND TRY

Question 9 Multiple choice

A program has five statements. The compiler reports an error on statement 4. What happens when the programmer tries to run it?

- A. Statements 1, 2, and 3 run, then the program stops.
- B. All five statements run, and the error is shown at the end.
- C. The program does not run at all until the error is fixed.
- D. The compiler skips statement 4 and runs the other four.

Question 10 Short answer

Put these in the order they happen, and explain why the order cannot change: *the program displays its output, the programmer types the code, the compiler checks the code.*

1.1.6 Syntax Errors

You have seen the compiler refuse to run a program with a missing semicolon. That kind of mistake is so common that it has its own name, and learning to read the compiler's messages about it will save you a lot of time.

In plain words: every language has grammar rules. In English, "the dog ran" follows the rules and "dog the ran" does not. Java's rules are much stricter than English's, and the computer allows no exceptions. The rules for how code must be written are called the language's **syntax**. A **syntax error** is code that breaks the language's rules, like a sentence with broken grammar. **Syntax errors are detected by the compiler**, which is why a syntax error stops a program from running.

Here are some common syntax errors and the message the compiler gives for each:

Mistake	Example	Compiler message (main part)
Missing semicolon	<code>System.out.println("Hi")</code>	<code> ';' expected</code>

Missing closing quote	<code>System.out.println("Hi);</code>	unclosed string literal
Wrong capitalization	<code>system.out.println("Hi");</code>	package system does not exist
Misspelled word	<code>System.out.printn("Hi");</code>	cannot find symbol

Java is **case-sensitive**: uppercase and lowercase letters are different, so `system` and `System` are two different words to Java.

KEY IDEA

A syntax error breaks the rules of the programming language. The compiler detects it, and the program cannot run until it is fixed.

WORKED EXAMPLE Reading a compiler message

This program has one syntax error:

UnclosedQuote.java

```
1 public class UnclosedQuote
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Good morning!);
6         System.out.println("Welcome to AP CSA.");
7     }
8 }
```

The compiler reports:

```
UnclosedQuote.java:5: error: unclosed string literal
    System.out.println("Good morning!);
                        ^
```

Read the message in three parts:

1. **Where:** `UnclosedQuote.java:5` means the file `UnclosedQuote.java`, line 5.
2. **What:** `unclosed string literal` means some text in quotes was opened but never closed. (Text inside double quotes is called a string literal; you will study it in Topic 1.3.)
3. **Exactly where:** the caret `^` points to the opening quote before `Good`.

Now look at line 5. There is an opening `"` before `Good`, but the closing `"` before the `)` is missing. Java kept reading, looking for the end of the text, and never found it on that line. The fix is `System.out.println("Good morning!");`.

Change one thing: put two mistakes in the program instead of one.

TwoMistakes.java

```
1 public class TwoMistakes
2 {
3     public static void main(String[] args)
4     {
5         system.out.println("Good morning!");
6         System.out.printn("Welcome to AP CSA.");
7     }
8 }
```

```
TwoMistakes.java:5: error: package system does not exist
TwoMistakes.java:6: error: cannot find symbol
2 errors
```

Line 5 uses a lowercase `s`. Java is case-sensitive, so `system` is not the same word as `System`, and Java does not know what `system` is. Line 6 misspells `println` as `printn`, and "cannot find symbol" means Java looked for something with that name and found nothing. The compiler listed both errors this time, and the program will not run until **both** are fixed. (The real messages include more lines showing the code and a caret; only the first line of each is shown here.)

WATCH OUT

Fixing only the first error. When the compiler lists several errors, students sometimes fix the first one and expect the program to run. It will not run until every compiler error is fixed. Work through the list from the top, recompiling as you go, because one mistake sometimes causes several messages, and fixing it can make the others disappear.

PAUSE AND TRY**Question 11** Debugging

This program does not compile. Find the error, write the corrected line, and explain what the compiler would report.

PracticeQuote.java

```
1 public class PracticeQuote
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Practice makes progress.");
6         System.out.println("Mistakes help you learn.");
7     }
8 }
```

Question 12 Multiple choice

Which line contains a syntax error?

- A. `System.out.println("Go Falcons!");`
- B. `System.out.println("Go Eagles");`
- C. `System.out.println("Go Hawks!");`

```
D. System.out.println("Go Lions!!");
```

1.1.7 Logic Errors and Testing

The compiler is helpful, but it only checks whether code follows Java's rules. It has no idea what you *wanted* the program to do. So what happens when the code follows every rule but does the wrong thing?

Building on 1.1.3 and 1.1.5: the compiler checks for some errors, and statements run in sequence, one at a time, in the order written.

In plain words: you can write a perfectly grammatical sentence that gives the wrong directions.

"Turn right at the light" is good English, but if you should have turned left, your friend still gets lost.

A **logic error** is a mistake in the plan or the code that makes the program do the wrong thing, or something you did not expect. The program compiles and runs to the end, but its result is wrong. The compiler cannot detect logic errors, because the code follows every rule.

So how are logic errors found? By **testing**: running the program with specific data and checking whether it produces the **expected outcome**. You decide in advance what the correct result should be, run the program, and compare. If the actual result differs from the expected result, there is a logic error.

KEY IDEA

A logic error makes a program behave incorrectly even though it compiles and runs. The compiler cannot find it; testing with specific data and comparing to the expected outcome can.

WORKED EXAMPLE A program that compiles but is wrong

This program is supposed to display the steps for making a peanut butter sandwich in order:

SandwichSteps.java

```
1 public class SandwichSteps
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("1. Put the top slice of bread on the sandwich.");
6         System.out.println("2. Spread peanut butter on one slice.");
7         System.out.println("3. Take out two slices of bread.");
8     }
9 }
```

It compiles with no errors, because every statement follows Java's rules. It runs and displays:

```
1. Put the top slice of bread on the sandwich.
2. Spread peanut butter on one slice.
3. Take out two slices of bread.
```

Now test it. Before running it, write down the **expected outcome**: the bread must be taken out first, then spread, then topped. Compare the expected order to the actual output: the actual output starts with putting the top slice on a sandwich that does not exist yet. The two do not match, so there is a logic error. The statements are in the wrong sequence.

The fix is to reorder the statements:

SandwichStepsFixed.java

```
1 public class SandwichStepsFixed
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("1. Take out two slices of bread.");
6         System.out.println("2. Spread peanut butter on one slice.");
7         System.out.println("3. Put the top slice of bread on the sandwich.");
8     }
9 }
```

1. Take out two slices of bread.
2. Spread peanut butter on one slice.
3. Put the top slice of bread on the sandwich.

Now the actual output matches the expected outcome, so this test passes.

Look back at the lunch program from Question 7. It compiled and ran, displaying "Unpack your lunch." after "Eat." If the expected outcome was a sensible lunch routine, that second "Unpack your lunch." is a logic error, and only comparing the output with what you expected reveals it.

WATCH OUT

"It compiled, so it works." No error message does not mean the program is correct. It only means there are no errors the compiler can detect. Students often stop as soon as the compiler is happy. Always decide the expected outcome first, then run the program and compare. Testing is the only way to catch logic errors.

PAUSE AND TRY

Question 13 Multiple choice

A program compiles, runs to the end, and displays the steps of a recipe in the wrong order. What kind of error is this, and how would it be found?

- A. A syntax error, found by the compiler
- B. A logic error, found by testing and comparing to the expected outcome
- C. A syntax error, found by testing
- D. A logic error, found by the compiler

Question 14 Short answer

An algorithm is supposed to double a number. A student writes: "1. Get the number. 2. Add 2 to it." The student tests it with the number 2, gets 4, and decides it works. Give one better

test, with its expected outcome and actual result, and explain why the test with 2 hid the logic error.

1.1.8 Run-Time Errors and Exceptions

There is one more way for a program to go wrong. Some code follows every rule, so it compiles, and it might even be the right plan, but when it actually runs, it asks the computer to do something impossible. Think of a recipe that says "divide the cookies evenly among the guests" on a day when no guests show up. The instruction was fine on paper; it fails only at the moment it is carried out.

Building on 1.1.5 and 1.1.3: compiling happens before running, the compiler checks for some errors, and statements run one at a time, in order.

In plain words: the program starts running normally, reaches a step it cannot possibly complete, and stops right there, without finishing the rest.

A **run-time error** is a problem that shows up **while the program is running**, not before. It usually makes the program **terminate abnormally**: it stops before reaching its end. An **exception** is one kind of run-time error: something goes wrong that the compiler **did not catch**, and Java **interrupts the normal flow** of the program, so the statements after it in the sequence do not run. When this happens, programmers say the program **threw an exception**.

KEY IDEA

A syntax error is caught before running, by the compiler. A run-time error (such as an exception) happens while the program is running and stops it early. A logic error lets the program run to the end, but with wrong results.

To see a run-time error, we need code that compiles but asks for something impossible. Dividing by zero is impossible in math, and in Java the symbol `/` means divide, as on a calculator. (Arithmetic in Java is taught fully in Topic 1.3; here you only need to know that `10 / 0` asks Java to divide 10 by 0.) The compiler checks that `10 / 0` is written by the rules, and it is, so the compiler does not complain.

WORKED EXAMPLE An exception in the middle of a program

Snack.java

```
1 public class Snack
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Sharing 10 crackers among 0 friends...");
6         System.out.println(10 / 0);
7         System.out.println("Enjoy your snack!");
8     }
9 }
```

Follow the program the way the computer does: compile first, then run one statement at a time.

Moment	What happens	Console shows
Compile	The compiler checks all the code. Every statement follows the rules, so compiling succeeds.	(nothing)
Run line 5	Displays the text.	Sharing 10 crackers among 0 friends...
Run line 6	Java tries to divide 10 by 0. That is impossible, so Java throws an exception, and the normal flow is interrupted.	Exception in thread "main" java.lang.ArithmeticException: / by zero
Line 7	Never runs. The program has terminated abnormally.	(nothing)

The complete console output is:

```
Sharing 10 crackers among 0 friends...
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Snack.main(Snack.java:6)
```

Read the exception message the same way you read compiler messages. `ArithmeticException` is the name of the kind of exception: a math problem. `/ by zero` describes what went wrong. `Snack.java:6` gives the line where it happened.

Notice two differences from a syntax error. First, the first line of output **did** appear, because the program really was running when the problem happened. Second, the message starts with `Exception` instead of `error`. And "Enjoy your snack!" never appears, because the exception interrupted the sequence before reaching it.

Now change one thing: move the impossible step to the top.

`SnackFirst.java`

```
1 public class SnackFirst
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(10 / 0);
6         System.out.println("Sharing 10 crackers among 0 friends...");
7         System.out.println("Enjoy your snack!");
8     }
9 }
```

This version also compiles. But the division is now the first statement, so the exception happens before anything is displayed. The console shows only:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at SnackFirst.main(SnackFirst.java:5)
```

Both versions compile, and both terminate abnormally. What changes is how much of the sequence finishes before the interruption, and that depends only on where the impossible step sits in the order.

Here are all three kinds of errors side by side:

Question	Syntax error	Run-time error (exception)	Logic error
Does it compile?	No	Yes	Yes
Does it start running?	No	Yes	Yes
How does it end?	Never runs	Stops early (terminates abnormally)	Runs to the end with a wrong result
Who or what finds it?	The compiler	The program crashes with a message	Testing against the expected outcome

WATCH OUT

"An exception is just a syntax error that shows up later." They are different. A syntax error breaks Java's rules, so the compiler catches it and nothing runs. Code that throws an exception follows every rule, so it compiles and starts running; the problem appears only when the impossible step is reached. Ask two questions to tell them apart: *Did it compile?* and *Does the message say `error:` (from the compiler) or start with `Exception` (from the running program)?* Do not rely on whether output appeared first: as `SnackFirst.java` shows, an exception can happen before anything is displayed.

PAUSE AND TRY**Question 15** Tracing

Trace `Snack.java` from the worked example, but imagine line 6 is moved to the very end, after the "Enjoy your snack!" statement. Does the program compile? Write the complete console output.

Question 16 Multiple choice

A program displays two lines of output and then shows a message starting with `Exception in thread "main"`. Which statement is true?

- A. The compiler found a syntax error.
- B. The program compiled, started running, and was interrupted by a run-time error.
- C. The program finished normally, but had a logic error.
- D. The program never started running.

Putting it together

Jordan is writing a program that displays a short study plan. Here is Jordan's first attempt, followed by everything that happens as Jordan fixes it. Read each stage and connect it to the subsection that explains it.

The algorithm (1.1.1 to 1.1.3). Jordan's plan, in written language: 1. Review notes. 2. Do practice problems. 3. Check answers. Each step is clear, and the order matters: checking answers before doing the problems would make no sense.

Attempt 1: a syntax error (1.1.5, 1.1.6). Jordan types:

```
1 System.out.println("1. Review notes.");
2 System.out.println("2. Do practice problems.");
3 System.out.println("3. Check answers.");
```

The first statement is missing its semicolon. The compiler reports `';' expected` and stops. **Nothing is displayed**, not even the correct lines, because the program never runs.

Attempt 2: a logic error (1.1.7). Jordan adds the semicolon but, while editing, swaps two statements:

```
1 System.out.println("1. Review notes.");
2 System.out.println("3. Check answers.");
3 System.out.println("2. Do practice problems.");
```

It compiles and runs to the end, displaying the lines in that order. Jordan's expected outcome was steps 1, 2, 3. The actual output does not match, so testing reveals a logic error in the sequence. Swapping the last two statements back fixes it.

Attempt 3: a run-time error (1.1.8). Jordan adds a line intended to show minutes per problem, dividing 60 minutes by the number of problems, but types `0` as a placeholder:

```
1 System.out.println("1. Review notes.");
2 System.out.println("2. Do practice problems.");
3 System.out.println(60 / 0);
4 System.out.println("3. Check answers.");
```

It compiles, because the code follows every rule. When it runs, the first two lines are displayed, then `60 / 0` throws an `ArithmeticException` and the program terminates abnormally. "3. Check answers." is never displayed.

Attempt	Compiles?	What is displayed	Kind of error	Found by
1	No	Nothing	Syntax	The compiler
2	Yes	All three lines, wrong order	Logic	Testing
3	Yes	First two lines, then an exception message	Run-time	Running it

Three different mistakes, found at three different moments: before running, after running, and during running.

Vocabulary

Term	Definition	Tiny example
algorithm	A set of ordered steps that gets a job done or solves a problem	A recipe
literal follower	This chapter's name for a reader (like a computer) that does exactly what is written and fills in nothing	Skips "turn on the toaster" if it is not written

written language	Representing an algorithm as sentences, usually a numbered list	"1. Fill the kettle. 2. Turn it on."
diagram / flowchart	Representing an algorithm as a picture; a flowchart uses boxes for steps and arrows for order	Start → Fill kettle → Turn on → Stop
sequencing	The order in which steps are completed, one at a time	Socks before shoes
programming language	A language with strict rules for writing instructions a computer can follow	Java
program / source code	An algorithm written in a programming language; the text a person writes	Greeting.java
statement	One instruction in a program	<code>System.out.println("Hi");</code>
text editor	Any application for writing plain text, including code	A notes app
integrated development environment (IDE)	An application with tools to write, compile, and run code	The editor your class uses for Java
compiler	Checks code for some errors and translates it so the computer can run it	Reports <code>';'</code> expected
compiling / running	Checking and translating the code / carrying out the translated statements	Compile first, then run
syntax	The rules for how code must be written	Statements end with <code>;</code>
syntax error	A mistake where the language's rules are not followed; detected by the compiler	A missing semicolon
case-sensitive	Uppercase and lowercase letters count as different	System is not system
logic error	A mistake that makes a program behave incorrectly or unexpectedly; found by testing	Steps displayed in the wrong order
testing / expected outcome	Running a program with specific data and comparing the result to what should happen	Expect 10 for doubling 5
run-time error	A mistake that occurs while the program is running; usually terminates it abnormally	Dividing by zero
exception	A run-time error the compiler did not catch; it stops the program's normal flow	<code>ArithmeticException: / by zero</code>

Chapter summary

- An **algorithm** is a set of ordered steps that gets a job done. Its steps must be precise, because a computer is a literal follower, and they can be written as numbered steps or drawn as a flowchart.
- **Sequencing** matters: steps are completed one at a time, in order, so the same steps in a different order can give a different result. Java statements run the same way, top to bottom.
- Code can be written in any **text editor**; an **IDE** lets you type, compile, and run it in one place. The **compiler** checks code before it runs, and any error it finds must be fixed before anything runs.
- A **syntax error** breaks Java's rules and is caught by the compiler. A **logic error** runs to the end with a wrong result and is found by **testing** against the **expected outcome**.
- A **run-time error** happens while the program runs and stops it early; an **exception** is a run-time error the compiler did not catch.

Review questions

Question 17 Multiple choice

A student draws this flowchart for borrowing a library book: Start → Scan your student ID → Hand the book to the librarian → Find the book on the shelf → Stop. What is the best description of the problem?

- A. Flowcharts cannot represent algorithms; it must be rewritten as a numbered list.
- B. One of the steps is too vague for a literal follower to carry out.
- C. The steps are in the wrong order: the book must be found before it can be handed over.
- D. Nothing is wrong; these steps can be done in any order.

Question 18 Tracing

What does this program display?

Morning.java

```
1 public class Morning
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Wake up.");
6         System.out.println("Eat breakfast.");
7         System.out.println("Brush teeth.");
8         System.out.println("Eat breakfast.");
9     }
10 }
```

Question 19 Debugging

This program should display two lines, but it does not compile. Find **both** errors and write the corrected lines.

Cheer.java

```
1 public class Cheer
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Go team!")
6         System.out.println("We can do it!");
7     }
8 }
```

Question 20 Multiple choice

Which error is the compiler able to detect?

- A. Steps displayed in the wrong order
- B. A missing semicolon
- C. Dividing by zero while the program runs
- D. A test result that does not match the expected outcome

Question 21 Short answer

Explain the difference between a logic error and a run-time error. In your answer, say whether each one lets the program run to the end.

Question 22 Tracing

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

Countdown.java

```
1 public class Countdown
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("3");
6         System.out.println("2");
7         System.out.println(1 / 0);
8         System.out.println("Liftoff!");
9     }
10 }
```

Question 23 Short answer

An algorithm is supposed to convert hours to minutes by multiplying the number of hours by 60. A student writes: "1. Get the number of hours. 2. Add 60 to it." The student tests it with 1 hour, gets 61, and thinks "close enough." (a) Give two tests, with specific data and expected outcomes, that clearly reveal the logic error. (b) Explain why the test with 1 hour almost hid the error.

Reflection

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

☐ I can explain what an algorithm is and why its steps must be precise. Reread: 1.1.1 What Is an Algorithm?

- ☐ I can show the same algorithm as numbered steps and as a flowchart. Reread: 1.1.2 Representing Algorithms: Words and Diagrams
- ☐ I can explain why changing the order of steps can change the result. Reread: 1.1.3 Sequencing: Order Matters
- ☐ I can explain what a text editor and an IDE are used for. Reread: 1.1.4 From Algorithm to Program
- ☐ I can explain why a compiler error means no part of the program runs. Reread: 1.1.5 The Compiler: Checking Before Running
- ☐ I can read a compiler message and find the syntax error it describes. Reread: 1.1.6 Syntax Errors
- ☐ I can explain how testing with expected outcomes finds logic errors. Reread: 1.1.7 Logic Errors and Testing
- ☐ I can tell a run-time error (exception) apart from syntax and logic errors. Reread: 1.1.8 Run-Time Errors and Exceptions

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. An algorithm is a step-by-step process for completing a task. Choice B gives ordered steps that someone could follow to reach the house.

- **A** is wrong: it is advice, not a set of steps. Nobody could follow it to complete a task.
- **C** is wrong: it is a fact about distance. It describes the trip but gives no steps.
- **D** is wrong: it is an opinion. A literal follower cannot act on "nicest."

If you chose differently, reread *1.1.1 What Is an Algorithm?*.

Question 2

Model answer: The algorithm never says to turn the toaster on (or push the lever down). A literal follower would put the bread in and immediately take it out, still untoasted, because nothing told it to start toasting.

A complete answer must:

- name a specific missing step (turning the toaster on, choosing a setting, or waiting until the toast is done are all good), and
- explain what goes wrong if a literal follower skips it.

If your answer only said "it is too short," reread *1.1.1 What Is an Algorithm?*: the problem is a missing step, not length.

Question 3

"Type your username" comes immediately after "Go to the class website." **Four** steps are completed before "Click Sign in": open the browser, go to the website, type the username, and type the password.

To trace a flowchart, put your finger on Start and move along one arrow at a time, naming each box as you reach it. If your count was different, you may have counted Start or Stop as steps; they mark the beginning and end but are not actions.

Reread *1.1.2 Representing Algorithms: Words and Diagrams* if your answer differed.

Question 4

B. It has the same four steps in the same order as the flowchart's arrows. A list and a flowchart represent the same algorithm when their steps and order match.

- **A** is wrong: the first two steps are swapped, so the kettle is turned on while empty. Same steps, different order, different algorithm.
- **C** is wrong: it waits for the water to boil before the kettle is on, so the water never boils. The order changed.
- **D** is wrong: it leaves out "Wait for the water to boil." A missing step makes a different algorithm.

Reread *1.1.2 Representing Algorithms: Words and Diagrams* if you chose differently.

Question 5

C. Sequencing means steps are done one at a time in the order written. Sealing the envelope first means the letter cannot be put inside afterward, because step 2 depends on the envelope still being open.

- **A** is wrong: every needed step is present. The problem is order, not a missing step.
- **B** is wrong: each step is necessary, and the number of steps is not the issue.
- **D** is wrong: putting the letter in depends on the envelope being open, so these two steps cannot be done in any order.

Reread 1.1.3 *Sequencing: Order Matters* if you chose differently.

Question 6

Model answer: Steps 1 and 2 (or steps 2 and 3) can be swapped safely, because taking out the spoon does not depend on the bowl, and the bowl does not depend on the spoon. Swapping steps 1 and 3 is not safe: step 3 pours cereal into the bowl, so it depends on step 1 having already taken the bowl out. Done first, step 3 would pour cereal onto the counter.

A complete answer must:

- name a pair of steps that do not depend on each other, and
- explain that step 3 depends on step 1, so step 1 must come first.

Reread 1.1.3 *Sequencing: Order Matters* if your reasoning differed.

Question 7

The statements run one at a time, top to bottom, and each displays one line:

```
Wash your hands.  
Unpack your lunch.  
Eat.  
Unpack your lunch.  
Clean up.
```

Notice that "Unpack your lunch." appears **twice**, because the statement appears twice. The computer does not notice that unpacking again after eating is silly; it is a literal follower and runs every statement in order. (Whether this program does what its author *wanted* is a different question, which 1.1.7 answers.)

Reread 1.1.4 *From Algorithm to Program* if your output differed.

Question 8

B. Source code is plain text, so any text editor can hold it. Programmers use an IDE because typing, compiling, and running all happen in one application.

- **A** is wrong: code is plain text, so any text editor works. An IDE is convenient, not required.
- **C** is wrong: an IDE is an application for working with code. Java is the language the code is written in.
- **D** is wrong: a plain text editor cannot run programs at all, and an IDE runs a program when you tell it to, not as you type.

Reread 1.1.4 *From Algorithm to Program* if you chose differently.

Question 9

C. Errors the compiler detects must be fixed before the program can run. The compiler checks the whole program before anything runs, so one error means nothing runs.

- **A** is wrong: this pictures the computer running statements until it hits the error. Compiling happens before running, so statements 1 to 3 never run either.
- **B** is wrong: with a compiler error, the program is never translated, so no statement runs.
- **D** is wrong: the compiler never skips or fixes statements for you; it reports the error and stops.

Reread 1.1.5 *The Compiler: Checking Before Running* if you chose differently.

Question 10

Model answer: (1) The programmer types the code. (2) The compiler checks the code. (3) The program displays its output. The compiler can only check code that already exists, and the program can only run after the compiler has checked and translated it without errors, so each step depends on the one before.

A complete answer must:

- give the order type → compile → run, and
- explain that running depends on a successful compile (and compiling depends on the code being written).

Reread 1.1.5 *The Compiler: Checking Before Running* if your order differed.

Question 11

Error: line 6 misspells `println` as `printline`.

Corrected line:

```
1 System.out.println("Mistakes help you learn.");
```

The compiler reports `cannot find symbol` for line 6, because Java looked for something named `printline` and found nothing by that name. It is a syntax error, so the program does not run at all; even line 5's message is not displayed until line 6 is fixed.

Reread 1.1.6 *Syntax Errors* if you missed the error.

Question 12

C. It is missing the semicolon at the end of the statement, which breaks Java's rules. The compiler would report `';' expected`.

- **A** is wrong: it follows every rule.
- **B** is wrong: leaving out the `!` changes the text that is displayed, but text inside the quotes can be anything, so it is not a syntax error.
- **D** is wrong: two exclamation points are just part of the text inside the quotes. The rules care about the semicolon, quotes, and parentheses, not the words inside the quotes.

Reread 1.1.6 *Syntax Errors* if you chose differently.

Question 13

B. The code follows Java's rules (it compiled) and runs to the end, but behaves incorrectly. That is a logic error, and logic errors are found by testing the program with specific data and comparing to the expected outcome.

- **A** is wrong: a syntax error would stop the program from compiling, but this one compiled.
- **C** is wrong: it is not a syntax error, because the code compiled. Testing finds logic errors.
- **D** is wrong: the compiler only checks rules, not what you meant, so it cannot detect logic errors.

Reread *1.1.7 Logic Errors and Testing* if you chose differently.

Question 14

Model answer: Test with 5. The expected outcome is 10 (5 doubled), but the algorithm gives 7 (5 + 2), so the test reveals the logic error. The test with 2 hid it because 2 doubled and 2 plus 2 both equal 4; that one input happens to give the right answer by coincidence.

A complete answer must:

- give a test value other than 2, with both the expected outcome (double) and the actual result (plus 2), showing they differ, and
- explain that $2 + 2 = 2 \times 2$, so the test with 2 could not tell the right algorithm from the wrong one.

Testing with more than one value, including some that are different from each other, makes logic errors much harder to miss. Reread *1.1.7 Logic Errors and Testing* if your reasoning differed.

Question 15

Yes, it compiles, because every statement still follows Java's rules; moving a statement does not break any rule.

Here is the moved version:

SnackLast.java

```
1 public class SnackLast
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Sharing 10 crackers among 0 friends...");
6         System.out.println("Enjoy your snack!");
7         System.out.println(10 / 0);
8     }
9 }
```

When it runs, the two text statements run first, then the division throws an exception:

```
Sharing 10 crackers among 0 friends...
Enjoy your snack!
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

(followed by `at SnackLast.main(SnackLast.java:7)`, naming the new line). The program still terminates abnormally, but this time every normal statement finished first, because the impossible step is last in the sequence.

Reread *1.1.8 Run-Time Errors and Exceptions* if your output differed.

Question 16

B. Output appeared, so the program compiled and was running. The `Exception` message means an exception, a type of run-time error, interrupted it and it terminated abnormally.

- **A** is wrong: a syntax error stops the program before it runs, so no output would appear, and the message would say `error`, not `Exception`.
- **C** is wrong: a logic error lets the program finish normally with wrong results; it does not produce an exception message.
- **D** is wrong: two lines of output appeared, so it definitely started running.

Reread *1.1.8 Run-Time Errors and Exceptions* if you chose differently.

Question 17

C. Steps are completed one at a time, in the order the arrows show. "Hand the book to the librarian" depends on already having the book, so "Find the book on the shelf" must come first. Same steps, wrong sequence.

- **A** is wrong: a flowchart is a valid way to represent an algorithm. Rewriting it as a list would not fix the order.
- **B** is wrong: each step names one specific action. The problem is when the steps happen, not how clearly they are written.
- **D** is wrong: handing over the book depends on having found it, so these steps cannot be done in any order.

Reread *1.1.3 Sequencing: Order Matters* and *1.1.2 Representing Algorithms: Words and Diagrams* if you chose differently.

Question 18

The statements run one at a time, top to bottom:

```
Wake up.  
Eat breakfast.  
Brush teeth.  
Eat breakfast.
```

"Eat breakfast." appears twice because its statement appears twice. The computer does not decide that eating breakfast twice is odd; it is a literal follower and runs every statement in order. If the expected outcome was a routine with one breakfast, this is a logic error that testing would reveal.

Reread *1.1.4 From Algorithm to Program* and *1.1.3 Sequencing: Order Matters* if your output differed.

Question 19

Error 1: line 5 is missing its semicolon. The compiler reports `';' expected`.

Error 2: line 6 is missing the closing double quote before `)`. The compiler reports `unclosed string literal`.

Corrected lines:

```
1 System.out.println("Go team!");  
2 System.out.println("We can do it!");
```

Both are syntax errors, so the program displays nothing until both are fixed. Fixing only one still leaves a compiler error.

Reread *1.1.6 Syntax Errors* if you missed either error.

Question 20

B. A missing semicolon breaks Java's rules, so it is a syntax error, and syntax errors are detected by the compiler.

- **A** is wrong: steps in the wrong order follow the rules, so it is a logic error, found by testing.
- **C** is wrong: dividing by zero follows the rules and fails only while running, so it is a run-time error (an exception) that the compiler did not detect.
- **D** is wrong: comparing to the expected outcome is testing, which finds logic errors. The compiler knows nothing about expected outcomes.

Reread 1.1.6 *Syntax Errors* and the comparison table in 1.1.8 *Run-Time Errors and Exceptions* if you chose differently.

Question 21

Model answer: Both kinds of programs compile. A logic error lets the program run all the way to the end, but the result is wrong, so it is found by testing against the expected outcome. A run-time error happens while the program is running and typically terminates it abnormally, so the program does not reach its end; statements after the error never run.

A complete answer must say:

- both compile;
- a logic error runs to the end with an incorrect result and is found by testing;
- a run-time error stops the program early (terminates abnormally).

Reread 1.1.7 *Logic Errors and Testing* and 1.1.8 *Run-Time Errors and Exceptions* if your explanation differed.

Question 22

```
3
2
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

The first two statements run and display 3 and 2. The third tries to divide 1 by 0, which throws an exception and interrupts the normal flow, so the program terminates abnormally and **Liftoff!** is never displayed. It compiled because every statement follows Java's rules; the problem appears only when the division runs.

Reread 1.1.8 *Run-Time Errors and Exceptions* if your output differed.

Question 23

Model answer:

(a) Test with 2 hours: expected 120 minutes, actual 62. Test with 10 hours: expected 600 minutes, actual 70. Both results are far from the expected outcomes, so they clearly reveal the logic error: the algorithm adds 60 instead of multiplying by 60.

(b) For 1 hour, adding 60 gives 61 and multiplying by 60 gives 60. The two results are so close that the wrong answer looked almost right. Inputs where the right and wrong algorithms give nearly the same answer can hide a logic error.

A complete answer must include, for each test, the input, the expected outcome (hours × 60), and the actual result (hours + 60), and explain that 1 hour produces nearly the same result either way.

Reread *1.1.7 Logic Errors and Testing* if your reasoning differed.