

4.6 Using Text Files

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

By the end you will be able to

- Explain why a program stores data in a file and how that data differs from values held in variables.
- Connect a program to a text file with the File and Scanner classes, including the imports and the throws IOException clause Java requires.
- Read values from a file with nextInt, nextDouble, nextBoolean, next, and nextLine, and predict what each call returns.
- Use a while loop with hasNext to read a file whose length you do not know, and close the file when you are done.
- Break a line of text into an array of pieces with split.

What to remember from earlier

- A Scanner reads text one token at a time, and each call moves it forward. ([Topic 1.14: Calling Instance Methods, 1.14.4 Calling Methods on a Scanner](#))
- A class that is not in java.lang needs an import statement. ([Topic 1.7: Application Program Interface \(API\) and Libraries, 1.7.2 Packages and import](#))
- A while loop repeats its body as long as its condition is true. ([Topic 2.7: while Loops, 2.7.1 Repeating Code with a while Loop](#))
- An array holds a fixed number of values, and arr.length is how many. ([Topic 4.3: Array Creation and Access](#))
- A loop can visit every element of an array to find a total or a largest value. ([Topic 4.5: Implementing Array Algorithms](#))

4.6.1 Files Keep Data Between Runs

A weather club records the high temperature every day. So far, every program you have written got its data from values typed into the code, such as `int[] temps = {72, 68, 75};`, or from a `Scanner` built on a string. Both have the same problem: to use a new week of temperatures, someone has to edit the program and run it again. And when the program ends, every variable and every array disappears. Nothing the program held survives to the next run.

Real programs keep their data somewhere else: in a **file**. A file is storage for data that **persists**, meaning it stays saved when no program is running, even after the computer is turned off. A program can open the file while it runs and pull the data out. The club can add next week's temperatures to the file without touching the program at all, and the same program will read whatever the file holds that day.

Think of the program as a recipe and the file as the pantry. The recipe says what to do with the ingredients; the pantry holds the ingredients between meals. You can restock the pantry without rewriting the recipe.

This chapter uses **text files**: files that hold ordinary characters you could read in any text editor, arranged in lines. Here is the club's file, named `temps.txt`. It is shown exactly as it is stored:

```
72 68 75
80 77
```

To Java, this is not five numbers yet. It is a sequence of characters: a 7, a 2, a space, a 6, and so on, with a line break after 75 and another after 77. A `Scanner` turns those characters into values the same way it did for strings in Topic 1.14: it splits the text into **tokens**, chunks separated by **whitespace**. Whitespace means spaces, tabs, and line breaks; all three count as separators, and several in a row count as one separator.

KEY IDEA

A file stores data that persists when no program is running. A program reads the data while it runs, so the data can change without changing the program.

WORKED EXAMPLE Lines and tokens in a file

Count the lines and tokens in `temps.txt`, shown above.

- **Lines:** the file has two lines. The first is `72 68 75`; the second is `80 77`.
- **Tokens:** a `Scanner` sees `72`, `68`, `75`, `80`, `77`. That is five tokens. The line break after 75 separates 75 from 80 exactly the way a space would.

Now compare a second file that holds the same tokens with different spacing:

```
72
68    75

80 77
```

This file has four lines, counting the empty third line, but it still has the same five tokens in the same order. A program that reads tokens gets the same five values from either file. A program that reads whole lines (in 4.6.5) would see the difference.

WATCH OUT

Thinking a variable "remembers" from one run to the next. Students sometimes expect that if a program set `highScore` to 900 last time, it will start with 900 next time. It will not. Variables and arrays live in the computer's memory only while the program runs; each run starts fresh from the code. Data that must survive between runs has to be stored outside the program, in a file. (This course reads files; programs that write files exist too, but they are not part of the AP course.)

PAUSE AND TRY

Question 1 Multiple choice

The weather club's program reads `temps.txt`. Which statement is true?

- A. Once the program has run, the temperatures are stored inside the program's variables until the next time it runs.
- B. The club can add new temperatures to `temps.txt`, and the next run of the unchanged program will read them.
- C. The temperatures must be typed into an array initializer before the program can use them.
- D. The file disappears when the program ends, just as the program's variables do.

Question 2 Short answer

How many lines and how many tokens does this file have? List the tokens.

```
Lee 12
  Kim 9 true
Ana
```

4.6.2 Opening a File with File and Scanner

You already know how to pull tokens out of text with a `Scanner`. The only new step is pointing the `Scanner` at a file instead of at a string or the keyboard. Java uses two classes for this.

- The `File` class represents a file by its name. The constructor `File(String pathname)` takes the file's name, called its **pathname**, as a `String`. A plain name such as `"temps.txt"` means "the file with this name in the folder the program runs from."
- The `Scanner` class has a constructor `Scanner(File f)` that takes a `File` object and connects the scanner to that file for reading.

Together, these two steps **open** the file: they make its contents available for the program to read. After that, `nextInt`, `next`, and the other methods work exactly as they did on a string.

Both `File` and the exception class `IOException` (explained in 4.6.3) live in the `java.io` package, the part of the Java library for input and output. `Scanner` lives in `java.util`. None of them are in `java.lang`, so, as in Topic 1.7, each needs an import statement at the top of the file.

KEY IDEA

To read a file, create a `File` object from the file's name and pass it to the `Scanner` constructor. Import `java.io.File`, `java.io.IOException`, and `java.util.Scanner`.

WORKED EXAMPLE Reading the first two temperatures

This program reads the first two values from `temps.txt` (shown in 4.6.1). Look for the three imports, the `File` object, and the `Scanner` built from it.

FirstFileRead.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class FirstFileRead
6 {
7     public static void main(String[] args) throws IOException
8     {
9         File dataFile = new File("temps.txt");
10        Scanner input = new Scanner(dataFile);
11        int monday = input.nextInt();
12        int tuesday = input.nextInt();
13        System.out.println("Monday: " + monday);
14        System.out.println("Tuesday: " + tuesday);
15        input.close();
16    }
17 }
```

Line(s)	What it does
1-3	Make File, IOException, and Scanner available. Without them, the compiler does not recognize those names.
7	throws IOException at the end of the header is required whenever a method opens a file. 4.6.3 explains why.
9	Creates a File object that names temps.txt. dataFile holds a reference to it.
10	Creates a Scanner connected to that file. The scanner's place starts before the first token, 72.
11	nextInt() reads the token 72 and returns the int 72. The scanner moves past it.
12	nextInt() reads the next token, 68.
13-14	Print the two values.
15	close() tells Java the program is finished with the file. 4.6.7 explains this step.

```
Monday: 72
Tuesday: 68
```

The File variable on line 9 is used only once, to build the scanner. Most programs skip the variable and create the File object right inside the Scanner constructor call. This version does that and reads one more value:

OneLineOpen.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class OneLineOpen
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        int first = input.nextInt();
11        int second = input.nextInt();
12        int third = input.nextInt();
13        System.out.println("Sum of first three: " + (first + second + third));
14        input.close();
15    }
16 }
```

Sum of first three: 215

Line 9 evaluates `new File("temps.txt")` first, producing a reference to a new `File` object, and passes that reference straight to `new Scanner(...)`. The result is the same open file as before. Lines 10 to 12 read 72, 68, and 75, and $72 + 68 + 75$ is 215. The rest of this chapter uses this one-line form.

Earlier topics built scanners on `System.in` to read the keyboard. Reading keyboard input is not part of the AP exam; reading files is. That is why this chapter reads files; only the `split` examples in 4.6.8 work on plain strings.

WATCH OUT

Giving the Scanner the file name instead of a File. It is easy to write `new Scanner("temps.txt")`, leaving out `new File(...)`. That compiles, because `Scanner` also has a constructor that takes a `String`. But it scans the string itself, the nine characters `temps.txt`, not the file with that name:

ScannerOnName.java

```
1 import java.util.Scanner;
2
3 public class ScannerOnName
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("temps.txt");
8         String first = input.next();
9         System.out.println("First token: " + first);
10        input.close();
11    }
12 }
```

First token: temps.txt

The only token is the file name. Had line 8 called `nextInt()`, the program would have crashed with an `InputMismatchException`, because `temps.txt` is not a whole number. The fix is to wrap the name in a `File` object: `new Scanner(new File("temps.txt"))`, with the `java.io` imports and `throws IOException` added.

Forgetting `import java.io.File;`. This program has the other two imports but not that one:

`MissingFileImport.java`

```
1 import java.io.IOException;
2 import java.util.Scanner;
3
4 public class MissingFileImport
5 {
6     public static void main(String[] args) throws IOException
7     {
8         Scanner input = new Scanner(new File("temps.txt"));
9         System.out.println(input.nextInt());
10        input.close();
11    }
12 }
```

It does not compile. Java reports something like this (wording can vary between Java versions):

```
MissingFileImport.java:8: error: cannot find symbol
    symbol:   class File
```

Line 8 is the first line that uses the name `File`, and without the import the compiler has no idea what class that is. Adding `import java.io.File;` at the top fixes it.

PAUSE AND TRY

Question 3 Tracing

`temps.txt` holds the five values shown in 4.6.1. Write the exact output.

PauseTempsDiff.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class PauseTempsDiff
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner reader = new Scanner(new File("temps.txt"));
10        int a = reader.nextInt();
11        int b = reader.nextInt();
12        int c = reader.nextInt();
13        int d = reader.nextInt();
14        System.out.println(d - a);
15        System.out.println(b + c);
16        reader.close();
17    }
18 }
```

Question 4 Multiple choice

A program has all three imports and `throws IOException` on `main`. Which statement creates a `Scanner` that reads the file `prices.txt`?

- A. `Scanner input = new Scanner("prices.txt");`
- B. `Scanner input = new Scanner(new File("prices.txt"));`
- C. `Scanner input = new Scanner(File("prices.txt"));`
- D. `File input = new File(new Scanner("prices.txt"));`

Question 5 Short answer

`String` and `Math` never need an import, but a program that reads a file needs

`import java.io.File;`, `import java.io.IOException;`, and `import java.util.Scanner;`. Explain why, and say what the compiler reports if the `File` import is missing.

4.6.3 When a File Cannot Be Opened

Reading a string can never fail to start: the string is right there in the code. A file is different. The name might be misspelled, the file might have been moved or deleted, or it might never have been created. When the program tries to open a file that is not there, it cannot continue normally.

Java will not let you ignore that possibility. When code uses the `File` class to open a file, Java **requires** the method to say what should happen if the file cannot be opened. In this course, you do that by writing `throws IOException` at the end of the opening method's header. An `IOException` is the kind of exception Java uses for input and output problems, such as a missing file. Writing `throws IOException` in the header announces "this method might run into an input or output problem, and if it does, it will not handle it; it will pass it along." For `main`, passing it along means the program stops.

Why does Java insist? For most mistakes, such as dividing by zero, Java trusts you to write correct code. A missing file is not a mistake in the code; it depends on what is on the disk when the program runs, so even correct code can hit it. Java makes you acknowledge that risk where everyone reading the method header can see it.

KEY IDEA

A method that opens a file must declare `throws IOException` in its header. If the file cannot be opened while the program runs, the program stops with an exception.

WORKED EXAMPLE Leaving out throws IOException

This program is `FirstFileRead` with `throws IOException` removed from line 7:

`NoThrows.java`

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class NoThrows
6 {
7     public static void main(String[] args)
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        System.out.println(input.nextInt());
11        input.close();
12    }
13 }
```

It does not compile. Java reports something like this (wording can vary):

```
NoThrows.java:9: error: unreported exception FileNotFoundException; must be caught or
declared to be thrown
```

Two things in that message are worth decoding. `FileNotFoundException` is the specific kind of `IOException` that the `Scanner` constructor might throw when the file is missing. "Must be caught or declared to be thrown" lists Java's two options. "Declared to be thrown" is what `throws IOException` does. ("Caught" refers to a way of handling the exception inside the method, which is not part of this course.) Putting `throws IOException` back on line 7 fixes it, because every `FileNotFoundException` is a kind of `IOException`.

Now suppose the code is correct but the file is not there. This program has a typo: it asks for `temp.txt`, but the file is named `temps.txt`.

MissingFile.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class MissingFile
6 {
7     public static void main(String[] args) throws IOException
8     {
9         System.out.println("Opening the file...");
10        Scanner input = new Scanner(new File("temp.txt"));
11        System.out.println("First value: " + input.nextInt());
12        input.close();
13    }
14 }
```

The program compiles, because the compiler does not look at what files exist. When it runs, line 9 prints its message, then line 10 tries to open the file and fails:

```
Opening the file...
Exception in thread "main" java.io.FileNotFoundException: temp.txt (No such file or
directory)
```

(The part in parentheses varies between computers and Java versions.) Line 11 never runs. Because `main` declared `throws IOException`, the exception is passed out of `main`, and the program **terminates**. The fix is to use the file's real name, `temps.txt`.

Notice where the crash happens: at the `Scanner` constructor, which is where Java actually tries to open the file. Creating a `File` object only records a name, so `new File("temp.txt")` by itself succeeds even when no such file exists.

Methods that call file-reading methods

The rule applies to every method that opens a file, not just `main`. Here a helper method opens the file, so its header has `throws IOException`. `main` calls it, so `main` needs it as well:

HelperRead.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class HelperRead
6 {
7     public static int firstValue(String fileName) throws IOException
8     {
9         Scanner input = new Scanner(new File(fileName));
10        int value = input.nextInt();
11        input.close();
12        return value;
13    }
14
15    public static void main(String[] args) throws IOException
16    {
17        System.out.println("First temperature: " + firstValue("temps.txt"));
18    }
19 }
```

First temperature: 72

Take away `throws IOException` from line 15 and the program no longer compiles:

HelperMainNoThrows.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class HelperMainNoThrows
6 {
7     public static int firstValue(String fileName) throws IOException
8     {
9         Scanner input = new Scanner(new File(fileName));
10        int value = input.nextInt();
11        input.close();
12        return value;
13    }
14
15    public static void main(String[] args)
16    {
17        System.out.println("First temperature: " + firstValue("temps.txt"));
18    }
19 }
```

HelperMainNoThrows.java:17: error: unreported exception IOException; must be caught or declared to be thrown

`firstValue`'s header warns that calling it might produce an `IOException`. Line 17 calls it, so `main` faces the same risk and must declare it too. The warning travels up to every caller.

PAUSE AND TRY**Question 6** Multiple choice

The folder this program runs from has no file named `scores.txt`. What happens?

PauseMissingScores.java

```

1  import java.io.File;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  public class PauseMissingScores
6  {
7      public static void main(String[] args) throws IOException
8      {
9          System.out.println("Start");
10         File scores = new File("scores.txt");
11         System.out.println("File object made");
12         Scanner input = new Scanner(scores);
13         System.out.println("Reading");
14         input.close();
15     }
16 }

```

- A. It does not compile, because `scores.txt` does not exist.
- B. It prints Start, then stops with a `FileNotFoundException` on line 10.
- C. It prints Start and File object made, then stops with a `FileNotFoundException` on line 12.
- D. It prints all three messages, because `close` handles the missing file.

Question 7 Short answer

In `HelperRead`, `main` never creates a `File` or a `Scanner` itself. Explain why `main` still needs `throws IOException` in its header.

4.6.4 Reading Tokens from a File

Files usually mix kinds of data. A game might save each player as a name, a level, an accuracy percentage, and whether the player is online, all in one line. To use those values, the program has to read each token as the right type: the name as text, the level as a whole number, and so on.

You have met most of the reading methods already. Here is the full set of token readers on the AP Java Quick Reference:

Method	What it returns	Fails with <code>InputMismatchException</code> when the next token...
<code>int nextInt()</code>	The next token as an <code>int</code>	is not a whole number, or is too big to fit in an <code>int</code>
<code>double nextDouble()</code>	The next token as a <code>double</code>	is not a number
<code>boolean nextBoolean()</code>	The next token as a <code>boolean</code>	is not true or false

String next()	The next token as a String, whatever it is	never; any token is valid text
---------------	--	--------------------------------

Each of these skips any whitespace, reads one token, and moves the scanner past it. They never return the whitespace itself. `nextBoolean` is the only new one: it turns the token `true` into the boolean value `true` and `false` into `false`.

KEY IDEA

Each token-reading call skips whitespace and returns the next token as the type its name promises. The calls must match the order and types of the tokens in the file.

WORKED EXAMPLE Reading a saved player

`player.txt` holds one line:

```
Maya 17 88.5 true
```

`ReadPlayer.java`

```

1  import java.io.File;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  public class ReadPlayer
6  {
7      public static void main(String[] args) throws IOException
8      {
9          Scanner input = new Scanner(new File("player.txt"));
10         String name = input.next();
11         int level = input.nextInt();
12         double accuracy = input.nextDouble();
13         boolean online = input.nextBoolean();
14         System.out.println("Name: " + name);
15         System.out.println("Level: " + level);
16         System.out.println("Accuracy: " + accuracy);
17         System.out.println("Online: " + online);
18         input.close();
19     }
20 }
```

The `|` marks where the scanner is after each call.

Line	Call	Variable gets	Rest of input
10	<code>input.next()</code>	<code>name</code> ← "Maya"	17 88.5 true
11	<code>input.nextInt()</code>	<code>level</code> ← 17	88.5 true
12	<code>input.nextDouble()</code>	<code>accuracy</code> ← 88.5	true
13	<code>input.nextBoolean()</code>	<code>online</code> ← true	(nothing left)

```
Name: Maya
Level: 17
Accuracy: 88.5
Online: true
```

Each call matched its token: text, whole number, decimal number, `true`. `online` holds the `boolean` value `true`, not the four letters, so the program could use it directly as an `if` condition.

Now change only the spacing. `player-spread.txt` holds the same four tokens scattered over four lines, with extra spaces and an empty line:

```
Maya
  17

88.5      true
```

`ReadPlayerSpread` is `ReadPlayer` with only the file name changed:

`ReadPlayerSpread.java`

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class ReadPlayerSpread
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("player-spread.txt"));
10        String name = input.next();
11        int level = input.nextInt();
12        double accuracy = input.nextDouble();
13        boolean online = input.nextBoolean();
14        System.out.println("Name: " + name);
15        System.out.println("Level: " + level);
16        System.out.println("Accuracy: " + accuracy);
17        System.out.println("Online: " + online);
18        input.close();
19    }
20 }
```

```
Name: Maya
Level: 17
Accuracy: 88.5
Online: true
```

The output is identical. Every token reader skips all whitespace, including line breaks and blank lines, before its token. For token reading, only the **order** of the tokens matters.

One more variation: `nextDouble` accepts a whole number. Reading the first token of `temps.txt` with `nextDouble` returns `72.0`, a `double`:

WholeAsDouble.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class WholeAsDouble
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        double first = input.nextDouble();
11        System.out.println(first);
12        input.close();
13    }
14 }
```

72.0

The reverse is not true. `nextInt` will not accept 88.5; it does not round or drop the decimal part.

WATCH OUT

Reading the tokens out of order. This program tries to read the level first:

WrongOrder.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class WrongOrder
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("player.txt"));
10        int level = input.nextInt();
11        String name = input.next();
12        System.out.println(name + " is level " + level);
13        input.close();
14    }
15 }
```

Exception in thread "main" java.util.InputMismatchException

The next token is Maya, and `nextInt` cannot turn a name into a whole number, so line 10 throws an **InputMismatchException** and nothing prints. Students write this because they think of `nextInt` as "find the number," but it only ever looks at the *next* token. The fix is to read in file order: `next()` for the name first, then `nextInt()`.

The same exception appears when the token has the right look but the wrong range or spelling. A population of 3000000000 is too big for an `int` (the largest is about 2.1 billion), and `nextBoolean` rejects `yes` because only `true` and `false` count:

BooleanYes.java

```

1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class BooleanYes
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("answer.txt"));
10        boolean agreed = input.nextBoolean();
11        System.out.println(agreed);
12        input.close();
13    }
14 }

```

With `answer.txt` holding `yes`, line 10 throws an `InputMismatchException`. A file meant to be read with `nextBoolean` must store the words `true` and `false`.

PAUSE AND TRY

Question 8 Tracing

`player.txt` holds `Maya 17 88.5 true`, as in the worked example. Write the exact output.

PauseTokens.java

```

1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class PauseTokens
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("player.txt"));
10        String who = input.next();
11        int level = input.nextInt();
12        double accuracy = input.nextDouble();
13        boolean online = input.nextBoolean();
14        if (online)
15        {
16            System.out.println(who + " is online");
17        }
18        System.out.println(accuracy + level);
19        input.close();
20    }
21 }

```

Question 9 Multiple choice

`city.txt` holds `Springfield 30000000000`. What happens when this program runs?

TooBigInt.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class TooBigInt
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("city.txt"));
10        String city = input.next();
11        int population = input.nextInt();
12        System.out.println(city + ": " + population);
13        input.close();
14    }
15 }
```

- A. It prints Springfield: 3000000000.
- B. It prints Springfield: 2147483647, the largest int.
- C. It throws an InputMismatchException on line 11, because the token is too big for an int.
- D. It throws an InputMismatchException on line 10, because next cannot read a city name.

4.6.5 Reading Whole Lines

roster.txt lists club members, one full name per line:

```
Ada Lovelace
Grace Hopper
Alan Turing
```

Each name contains a space. Token readers treat that space as a separator, so `next()` would split every name in two. When each line of a file is one record, you want the whole line at once.

That is what `nextLine()` does, as you saw in Topic 1.4: it returns everything from the scanner's current place up to the end of the line, spaces included, as a `String`. The line break itself is not part of the result; the scanner moves past it to the start of the next line.

KEY IDEA

`nextLine` returns the rest of the current line, spaces included, without the line break. Use it when each line of a file is one piece of data.

WORKED EXAMPLE Reading full names

ReadRoster.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class ReadRoster
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("roster.txt"));
10        String first = input.nextLine();
11        String second = input.nextLine();
12        String third = input.nextLine();
13        System.out.println("1: " + first);
14        System.out.println("2: " + second);
15        System.out.println("3: " + third);
16        input.close();
17    }
18 }
```

Line	Call	Variable gets	Scanner moves to start of
10	input.nextLine()	first ← "Ada Lovelace"	line 2
11	input.nextLine()	second ← "Grace Hopper"	line 3
12	input.nextLine()	third ← "Alan Turing"	the end of the file

```
1: Ada Lovelace
2: Grace Hopper
3: Alan Turing
```

Now change one thing: read the same file with `next()` instead of `nextLine()`. Nothing else in the program changes.

ReadRosterNext.java

```

1  import java.io.File;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  public class ReadRosterNext
6  {
7      public static void main(String[] args) throws IOException
8      {
9          Scanner input = new Scanner(new File("roster.txt"));
10         String first = input.next();
11         String second = input.next();
12         String third = input.next();
13         System.out.println("1: " + first);
14         System.out.println("2: " + second);
15         System.out.println("3: " + third);
16         input.close();
17     }
18 }

```

```

1: Ada
2: Lovelace
3: Grace

```

`next()` stops at any whitespace, so the space inside Ada Lovelace ends the first token. Three calls get through only a name and a half. Side by side:

Method	Stops at	Spaces in the result?	First call on roster.txt
<code>next()</code>	any whitespace	never	"Ada"
<code>nextLine()</code>	the end of the line only	yes	"Ada Lovelace"

WATCH OUT

Mixing `nextLine` with the token readers on one file. The two kinds of methods handle the line break differently. A token reader such as `nextInt()` stops right after its token and leaves the rest of the line, including the line break, unread. If `nextLine()` is called next, it returns whatever is left of that line, which is often nothing: the empty string `""`. Programs that mix them need extra code to adjust for this, and that kind of code is not on the AP exam.

The safe habit for this course: in a given file, read with **either** `nextLine()` **or** the token readers, not both. If a file has full names, read lines. If it is all numbers and single words, read tokens.

PAUSE AND TRY**Question 10** Tracing

`books.txt` holds three lines:

```
Dune, Frank Herbert, 1965
Kindred, Octavia Butler, 1979
Hatchet, Gary Paulsen, 1986
```

Write the exact output.

PauseBookLine.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class PauseBookLine
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("books.txt"));
10        String line = input.nextLine();
11        line = input.nextLine();
12        System.out.println(line.length());
13        System.out.println(line.substring(0, 7));
14        input.close();
15    }
16 }
```

Question 11 Short answer

A file holds one street address per line, such as 42 Maple Street. A student reads each address with three calls: `nextInt()`, `next()`, `next()`. Explain why `nextLine()` is the better choice for this file.

4.6.6 Reading Until the File Runs Out

Every program so far knew exactly how many values the file held. But the point of a file is that its contents can change: the weather club adds a temperature every day. A program that makes exactly five `nextInt()` calls will miss the sixth day, and if a file has only four values, a fifth call will crash. The program needs to ask, before each read, "is there anything left?"

`hasNext()` answers that question. It returns `true` if the scanner has another token to read and `false` if only whitespace (or nothing) is left. It does not read anything or move the scanner; it only looks ahead.

That makes it a natural condition for a `while` loop (Topic 2.7): keep reading while there is something to read.

```
1 while (input.hasNext())
2 {
3     // read one value and use it
4 }
```

Each pass checks for more data, reads one item, and processes it. When the last item has been read, `hasNext()` returns `false` and the loop ends, whether the file held five values or five hundred.

KEY IDEA

A while loop with `hasNext()` as its condition reads a file of any length: it keeps reading while another item remains and stops when the file runs out.

WORKED EXAMPLE Total and average of every temperature

The program does not say how many values `temps.txt` holds. It counts them as it reads.

SumTemps.java

```

1  import java.io.File;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  public class SumTemps
6  {
7      public static void main(String[] args) throws IOException
8      {
9          Scanner input = new Scanner(new File("temps.txt"));
10         int total = 0;
11         int count = 0;
12         while (input.hasNext())
13         {
14             int temp = input.nextInt();
15             total += temp;
16             count++;
17         }
18         input.close();
19         System.out.println("Readings: " + count);
20         System.out.println("Total: " + total);
21         System.out.println("Average: " + (double) total / count);
22     }
23 }
```

Check on line 12	Line 14 reads	total	count
true	72	72	1
true	68	140	2
true	75	215	3
true	80	295	4
true	77	372	5
false	(loop ends)	372	5

After 77 only a line break remains, which is whitespace, so `hasNext()` returns `false`. Line 18 closes the file, and line 21 casts `total` to `double` (Topic 1.5) so the division keeps its decimal part: $372 \div 5 = 74.4$.

```
Readings: 5
Total: 372
Average: 74.4
```

If the club adds three more temperatures to the file, the same program reads eight values with no change to the code.

The same loop shape works with `nextLine()`. This program numbers every line of `roster.txt`:

NumberedRoster.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class NumberedRoster
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("roster.txt"));
10        int lineNumber = 1;
11        while (input.hasNext())
12        {
13            String line = input.nextLine();
14            System.out.println(lineNumber + ". " + line);
15            lineNumber++;
16        }
17        input.close();
18    }
19 }
```

```
1. Ada Lovelace
2. Grace Hopper
3. Alan Turing
```

Each pass reads one whole line. After `Alan Turing` nothing but a line break remains, so `hasNext()` is `false` and the loop stops.

WATCH OUT

Reading a fixed number of times. This loop assumes the file has six temperatures, but `temps.txt` has five:

FixedCountLoop.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class FixedCountLoop
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        for (int day = 1; day <= 6; day++)
11        {
12            System.out.println("Day " + day + ": " + input.nextInt());
13        }
14        input.close();
15    }
16 }
```

```
Day 1: 72
Day 2: 68
Day 3: 75
Day 4: 80
Day 5: 77
Exception in thread "main" java.util.NoSuchElementException
```

On the sixth pass there is no token at all, so `nextInt()` throws a **NoSuchElementException**, the exception for reading past the end of the input. (It is a close relative of `InputMismatchException`, which means a token exists but has the wrong type.) The fix is to let the file decide: replace the `for` header with `while (input.hasNext())`.

Forgetting to read inside the loop. This loop is meant to count the words in a file:

```
1 int words = 0;
2 while (input.hasNext())
3 {
4     words++;
5 }
```

It never ends. `hasNext()` only looks; it does not move the scanner. Nothing in the body reads a token, so the first token is still waiting every time, `hasNext()` stays `true`, and the loop runs forever. Every pass of a `hasNext()` loop must read something, such as `input.next();`, to move toward the end of the file.

PAUSE AND TRY

Question 12 Tracing

`temps.txt` holds 72 68 75 on one line and 80 77 on the next. Write the exact output.

PauseAboveCount.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class PauseAboveCount
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        int warmDays = 0;
11        int last = 0;
12        while (input.hasNext())
13        {
14            last = input.nextInt();
15            if (last > 74)
16            {
17                warmDays++;
18            }
19        }
20        input.close();
21        System.out.println(warmDays + " " + last);
22    }
23 }
```

Question 13 Debugging

This program is supposed to print every temperature in `temps.txt` that is above 74. Instead it prints 80 and then crashes with a `NoSuchElementException`. Identify the error and write a corrected loop body.

WarmPrintWrong.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class WarmPrintWrong
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        while (input.hasNext())
11        {
12            if (input.nextInt() > 74)
13            {
14                System.out.println(input.nextInt());
15            }
16        }
17        input.close();
18    }
19 }
```

4.6.7 Closing the File

While a program has a file open, the computer's operating system keeps track of it, and an open file can stay tied up until the program ends. A program that opens many files, or runs for a long time, should hand each file back as soon as it is finished with it. It is like returning a library book when you are done instead of keeping it until you move away.

The method `close()` does this. Calling `input.close()` closes the scanner and the file it was reading. After that call, the scanner cannot be used again.

Where does `close()` go? After the last read, which in a `hasNext()` loop means **after** the loop, as in `SumTemps` line 18. Closing is good practice in every program that reads a file, even short ones.

KEY IDEA

Call `close()` on the Scanner when the program is finished reading the file, after the last read. A closed Scanner cannot read any more.

WORKED EXAMPLE Using the scanner after closing it

This program closes the file too early, then tries to keep reading:

`ReadAfterClose.java`

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class ReadAfterClose
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("temps.txt"));
10        int first = input.nextInt();
11        input.close();
12        System.out.println("First: " + first);
13        int second = input.nextInt();
14        System.out.println("Second: " + second);
15    }
16 }
```

First: 72

Exception in thread "main" java.lang.IllegalStateException: Scanner closed

Line 10 reads 72. Line 11 closes the scanner. Line 12 still works, because it only prints a variable that already holds 72; it does not touch the scanner. Line 13 asks the closed scanner for another value, and Java throws an `IllegalStateException` with the message `Scanner closed`. The fix is to move `input.close()`; below the last read, after line 13.

The second most common placement mistake is putting `close()` inside a loop. Then the first pass closes the file, and the loop's next `hasNext()` check fails. You will debug that program in the review questions.

WATCH OUT

Leaving out `close()` because "it still works." A short program without `close()` often runs correctly, because the file is released anyway when the program ends. That is why students skip it. But the habit matters: in larger programs a forgotten file stays tied up for as long as the program runs. Treat opening and closing as a pair: every `new Scanner(new File(...))` gets a matching `close()` after its last read.

PAUSE AND TRY**Question 14** Multiple choice

`SumTemps` and `NumberedRoster` (in 4.6.6) both read a file with a `hasNext()` loop. Where should `input.close();` go in such a program?

- A. Right after the `Scanner` is created, so the file is not left open by accident
- B. As the last statement inside the loop body, so the file is closed after each item
- C. Immediately after the loop ends, once the last item has been read
- D. Nowhere; Java closes every file automatically, so `close()` is never needed

4.6.8 Splitting a Line into Pieces

Many data files store one record per line, with the fields separated by a punctuation mark. `books.txt` uses commas:

```
Dune,Frank Herbert,1965
Kindred,Octavia Butler,1979
Hatchet,Gary Paulsen,1986
```

`nextLine()` can read "Dune,Frank Herbert,1965" as one string, but the program wants the title, the author, and the year separately. Token readers will not help: they split at spaces, which would cut Frank Herbert in half and leave the commas attached. You could find each comma with `indexOf` and cut with `substring` (Topic 1.15), but that is a lot of index arithmetic for every line.

The `String` method `split` does this in one call. Its header is `String[] split(String del)`. The parameter is the **delimiter**, the string that separates the pieces. `split` returns a new array of `Strings` (Topics 4.3 to 4.5) holding the pieces between the delimiters, in order. The delimiters themselves are not included in any piece, and the original string is unchanged, as always for strings.

KEY IDEA

`line.split(",")` returns a `String` array of the pieces of line between the commas, in order, with the commas removed.

WORKED EXAMPLE Splitting one book record

```

1 String line = "Dune,Frank Herbert,1965";
2 String[] parts = line.split(",");
3 System.out.println(parts.length);
4 System.out.println(parts[0]);
5 System.out.println(parts[1]);
6 System.out.println(parts[2]);

```

The string has two commas, so it splits into three pieces:

index:	0	1	2
piece:	"Dune"	"Frank Herbert"	"1965"

`parts.length` is 3. `parts[1]` keeps its space, because the delimiter is a comma, not a space. `parts[2]` is the String "1965", not an `int`: `split` always returns text. (Topic 4.7 shows how to turn the text "1965" into the number 1965. Until then, keep numbers you need for arithmetic in files you read with `nextInt` or `nextDouble`.)

```

3
Dune
Frank Herbert
1965

```

Change only the delimiter and the pieces change completely:

```

1 String line = "Dune,Frank Herbert,1965";
2 String[] bySpace = line.split(" ");
3 System.out.println(bySpace.length);
4 System.out.println(bySpace[0]);
5 System.out.println(bySpace[1]);
6 String[] bySemicolon = line.split(";");
7 System.out.println(bySemicolon.length);
8 System.out.println(bySemicolon[0]);

```

```

2
Dune,Frank
Herbert,1965
1
Dune,Frank Herbert,1965

```

Split on " ", the line has one space, so there are two pieces, and the commas stay inside them. Split on ";", which never appears, there is nothing to split around, so the array has one element: the whole string. `split` always returns an array with at least one element.

Two more rules show up in real data:

- **The delimiter is matched exactly.** Splitting "Ada, 92" on ", " leaves the space at the start of the second piece. Splitting on ", " (comma and space) removes both.
- **Two delimiters in a row make an empty piece.** In "red,,blue,green", nothing sits between the two commas, so that piece is the empty string "".

```
1 String entry = "Ada, 92";
2 String[] pieces = entry.split(",");
3 System.out.println "[" + pieces[1] + "]";
4 String[] fixed = entry.split(" ");
5 System.out.println "[" + fixed[1] + "]";
```

```
[ 92]
[92]
```

The brackets are printed around each piece so you can see the space. The first piece holds three characters, a space and 92; the second holds just 92.

Now put `split` to work on a file. Read each line with `nextLine()` in a `hasNext()` loop, then split it:

BookList.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class BookList
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("books.txt"));
10        while (input.hasNext())
11        {
12            String line = input.nextLine();
13            String[] parts = line.split(",");
14            String title = parts[0];
15            String year = parts[2];
16            System.out.println(title + " (" + year + ")");
17        }
18        input.close();
19    }
20 }
```

```
Dune (1965)
Kindred (1979)
Hatchet (1986)
```

Each pass: line 12 reads one record, line 13 splits it into a three-element array, and lines 14 and 15 pick out the title at index 0 and the year at index 2. A new array is built on every pass, sized to fit that line's pieces.

WATCH OUT

Using a delimiter with special characters. The delimiter is not quite an ordinary string: it is written in a pattern language in which some punctuation marks, such as `.`, `*`, and `|`, have special meanings, so splitting on them does not do what you expect. That pattern language is not part of the AP course. Stick to plain delimiters such as a comma, a space, a colon, a semicolon, or a hyphen, and they behave exactly as described here.

PAUSE AND TRY**Question 15** Tracing

Write the exact output.

```
1 String colors = "red,,blue,green";
2 String[] parts = colors.split(",");
3 System.out.println(parts.length);
4 System.out.println "[" + parts[1] + ""];
5 System.out.println(parts[parts.length - 1]);
```

Question 16 Multiple choice

A file line holds `Ana Diaz, Lincoln High, 11`. Which call puts `"Lincoln High"` at index 1 of the result?

- A. `line.split(" ")`
- B. `line.split(",")`
- C. `line.split(", ")`
- D. `line.split("Lincoln")`

Putting it together

Two programs combine the chapter's tools with arrays from Topics 4.3 to 4.5.

Reading numbers into an array

`grades.txt` stores how many grades follow, then the grades:

```
5
88 92 75 100 81
```

Storing the count first is a common file layout: it tells the program how big an array to create before reading.

`GradeReport.java`

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class GradeReport
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("grades.txt"));
10        int count = input.nextInt();
11        int[] grades = new int[count];
12        for (int i = 0; i < grades.length; i++)
13        {
14            grades[i] = input.nextInt();
15        }
16        input.close();
```

```

17
18     int total = 0;
19     int highest = grades[0];
20     for (int grade : grades)
21     {
22         total += grade;
23         if (grade > highest)
24         {
25             highest = grade;
26         }
27     }
28     System.out.println("Grades read: " + grades.length);
29     System.out.println("Average: " + (double) total / grades.length);
30     System.out.println("Highest: " + highest);
31 }
32 }

```

Line(s)	What happens	State
9	Open grades.txt	scanner before 5
10	nextInt() reads the count	count = 5
11	Create an array with room for exactly 5 ints, all 0	{0, 0, 0, 0, 0}
12-15	Five passes; each nextInt() reads the next grade into grades[i], crossing the line break after 5	{88, 92, 75, 100, 81}
16	Close the file; everything needed is now in the array	
18-27	Enhanced for loop (Topic 4.4) totals the grades and tracks the highest	total = 436, highest = 100
29	436 ÷ 5 as a double	87.2

```

Grades read: 5
Average: 87.2
Highest: 100

```

The file can be closed as early as line 16 because the data now lives in the array, and the analysis loop never touches the scanner. A `for` loop is fine for the reading here, because the file itself said how many grades there are. Without that count, a `hasNext()` loop would be needed, and since an array's size is fixed when it is created, the program would have no way to know how big to make the array in advance.

Splitting records and filtering

signups.txt stores each student's name and chosen club:

```

Ada Lovelace,chess
Grace Hopper,robotics
Alan Turing,chess
Katherine Johnson,math
Mark Dean,chess

```

The program lists everyone who chose chess and counts them.

ChessSignups.java

```

1  import java.io.File;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  public class ChessSignups
6  {
7      public static void main(String[] args) throws IOException
8      {
9          Scanner input = new Scanner(new File("signups.txt"));
10         int chessCount = 0;
11         while (input.hasNext())
12         {
13             String line = input.nextLine();
14             String[] parts = line.split(",");
15             if (parts[1].equals("chess"))
16             {
17                 System.out.println(parts[0]);
18                 chessCount++;
19             }
20         }
21         input.close();
22         System.out.println(chessCount + " signed up for chess");
23     }
24 }

```

Pass	parts[0]	parts[1]	equals("chess")?	chessCount
1	"Ada Lovelace"	"chess"	yes, print	1
2	"Grace Hopper"	"robotics"	no	1
3	"Alan Turing"	"chess"	yes, print	2
4	"Katherine Johnson"	"math"	no	2
5	"Mark Dean"	"chess"	yes, print	3

```

Ada Lovelace
Alan Turing
Mark Dean
3 signed up for chess

```

The names have spaces, so the program reads lines (`nextLine()`), not tokens, and never mixes the two. Line 15 uses `equals`, not `==`, to compare string contents (Topic 2.6). The `close()` on line 21 comes after the loop, once every line has been read.

Common mistakes

Each of these was taught in context above. This table collects what Java reports, so you can recognize the message when it appears.

What Java reports	When	Likely cause and fix
-------------------	------	----------------------

unreported exception FileNotFoundException (or IOException)	Compiling	A method that opens a file, or calls one that does, needs throws IOException
cannot find symbol ... class File	Compiling	Add import java.io.File;
FileNotFoundException	Running, at new Scanner(...)	The file name is wrong or the file is not in the program's folder
InputMismatchException	Running, at a next... call	The next token is the wrong type or out of range; read in file order
NoSuchElementException	Running, at a next... call	Reading past the end of the file; loop with hasNext()
IllegalStateException: Scanner closed	Running, after close()	close() came before the last read; move it after the loop
No error, but the only token is the file name	Running	new Scanner("name.txt") scans new Scanner(new File("name.txt"))

Vocabulary

Term	Definition	Example
file	Storage for data that persists when no program is running	temps.txt
persists	Stays saved after the program ends	Temperatures in a file survive between runs
text file	A file of ordinary characters arranged in lines	72 68 75 on line 1
whitespace	Spaces, tabs, and line breaks, which separate tokens	The line break after 75
File	A java.io class whose objects represent a file by its name	new File("temps.txt")
pathname	The name (and location) of a file, passed to the File constructor	"temps.txt"
open a file	Connect a program to a file so it can read the contents	new Scanner(new File("temps.txt"))
java.io	The Java library package for input and output, containing File and IOException	import java.io.File;
IOException	The exception for input and output problems, such as a file that cannot be opened	throws IOException
throws IOException	A clause in a method header declaring that the method may pass on an IOException	main(String[] args) throws IOException

FileNotFoundException	The kind of <code>IOException</code> thrown when a file to be read does not exist	Opening <code>temp.txt</code> by mistake
nextBoolean()	Reads the next token as a <code>boolean</code> ; it must be <code>true</code> or <code>false</code>	<code>true</code> → <code>true</code>
hasNext()	Returns <code>true</code> if another token remains to be read, without reading it	<code>while (input.hasNext())</code>
NoSuchElementException	The exception thrown when a <code>Scanner</code> is asked to read past the end of its input	A sixth <code>nextInt()</code> on five values
close()	Closes a <code>Scanner</code> and the file it reads; the scanner cannot be used afterward	<code>input.close();</code>
split	A <code>String</code> method that returns an array of the pieces between delimiters	<code>"a,b".split(",")</code> → <code>{"a", "b"}</code>
delimiter	The string that separates pieces in a call to <code>split</code>	<code>", "</code>

Chapter summary

- A file stores data that persists when no program is running. A program reads it while running, so the data can change without changing the code.
- To open a file, import `java.io.File`, `java.io.IOException`, and `java.util.Scanner`, then write `new Scanner(new File("name.txt"))`. Every method that opens a file, and every method that calls one, needs throws `IOException`; if the file cannot be opened, the program terminates with a `FileNotFoundException`.
- `nextInt`, `nextDouble`, `nextBoolean`, and `next` skip whitespace and read one token of their type, throwing an `InputMismatchException` on a mismatch. `nextLine` reads the rest of a line, spaces included. Use one style or the other on a given file.
- A `while (input.hasNext())` loop reads a file of any length; every pass must read something. Call `close()` after the last read.
- `split(del)` returns a `String` array of the pieces between delimiters, with the delimiters removed. Pair it with `nextLine` to read records whose fields are separated by commas or other plain punctuation.

Review questions

Question 17 Multiple choice

`MissingFileImport` (repeated here) does not compile. Which change makes it compile and print 72?

MissingFileImport.java

```
1 import java.io.IOException;
2 import java.util.Scanner;
3
4 public class MissingFileImport
5 {
6     public static void main(String[] args) throws IOException
7     {
8         Scanner input = new Scanner(new File("temps.txt"));
9         System.out.println(input.nextInt());
10        input.close();
11    }
12 }
```

- A. Add `import java.io.File;` above line 1.
- B. Remove `throws IOException` from line 6, since the file exists.
- C. Change line 8 to `Scanner input = new Scanner("temps.txt");`
- D. Add `import java.util.File;` above line 1.

Question 18 Tracing

`words.txt` holds the quick brown fox on line 1 and jumps over the lazy dog on line 2. Write the exact output.

LongWords.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class LongWords
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("words.txt"));
10        int longCount = 0;
11        while (input.hasNext())
12        {
13            String word = input.next();
14            if (word.length() > 4)
15            {
16                System.out.println(word);
17                longCount++;
18            }
19        }
20        input.close();
21        System.out.println("Long words: " + longCount);
22    }
23 }
```

Question 19 Multiple choice

`player.txt` holds `Maya 17 88.5 true`. What happens when this program runs?

RevTokenOrder.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class RevTokenOrder
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("player.txt"));
10        String name = input.next();
11        double first = input.nextDouble();
12        System.out.println(name + " " + first);
13        int second = input.nextInt();
14        System.out.println(second);
15        input.close();
16    }
17 }
```

- A. It throws an `InputMismatchException` on line 11, because 17 is not written as a decimal.
- B. It prints Maya 17.0, then throws an `InputMismatchException` on line 13.
- C. It prints Maya 17.0 and then 88.
- D. It prints Maya 88.5 and then 17.

Question 20 Debugging

This program should welcome every member listed in `roster.txt` (three names, one per line). It prints `Welcome, Ada Lovelace` and then crashes with `java.lang.IllegalStateException: Scanner closed`. Identify the error and correct it.

CloseInsideLoop.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class CloseInsideLoop
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("roster.txt"));
10        while (input.hasNext())
11        {
12            String name = input.nextLine();
13            System.out.println("Welcome, " + name);
14            input.close();
15        }
16    }
17 }
```

Question 21 Tracing

`books.txt` holds the three lines shown in 4.6.8. Write the exact output.

RevAuthors.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class RevAuthors
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("books.txt"));
10        String result = "";
11        while (input.hasNext())
12        {
13            String[] parts = input.nextLine().split(",");
14            String[] names = parts[1].split(" ");
15            result += names[1].substring(0, 1);
16        }
17        input.close();
18        System.out.println(result);
19    }
20 }
```

Question 22 Multiple choice

What does this code print?

```
1 String names = "Lee, Kim, Ana";
2 String[] parts = names.split(",");
3 System.out.println(parts.length);
4 System.out.println "[" + parts[2] + ""];
5 System.out.println(parts[2].length());
```

- A. 3, then [Ana], then 3
- B. 3, then [Ana], then 4
- C. 5, then [Kim], then 3
- D. 2, then an `ArrayIndexOutOfBoundsException`

Question 23 Tracing

grades.txt holds 5 on the first line and 88 92 75 100 81 on the second. Write the exact output.

RevMinScore.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class RevMinScore
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner input = new Scanner(new File("grades.txt"));
10        int[] grades = new int[input.nextInt()];
11        for (int i = 0; i < grades.length; i++)
12        {
13            grades[i] = input.nextInt();
```

```
14     }
15     input.close();
16
17     int lowest = grades[0];
18     int over90 = 0;
19     for (int i = 1; i < grades.length; i++)
20     {
21         if (grades[i] < lowest)
22         {
23             lowest = grades[i];
24         }
25         if (grades[i] > 90)
26         {
27             over90++;
28         }
29     }
30     System.out.println(lowest + " " + over90);
31 }
32 }
```

Question 24 Short answer

The weather club's file gets one new temperature every day. One student's program reads it with `for (int day = 1; day <= 5; day++)` and a `nextInt()` call in the body. Another uses `while (input.hasNext())`. Explain which program keeps working as the file grows and shrinks, and what goes wrong with the other.

Reflection

- ☐ I can explain why data stored in a file persists when the program is not running. Reread: 4.6.1 Files Keep Data Between Runs
- ☐ I can open a file with the `File` and `Scanner` classes and name the imports it needs. Reread: 4.6.2 Opening a File with `File` and `Scanner`
- ☐ I can explain why a method that opens a file needs throws `IOException`, and what happens when the file is missing. Reread: 4.6.3 When a File Cannot Be Opened
- ☐ I can predict what `nextInt`, `nextDouble`, `nextBoolean`, and `next` return, and when they throw an `InputMismatchException`. Reread: 4.6.4 Reading Tokens from a File
- ☐ I can choose between `nextLine` and `next`, and explain why one file should not mix them. Reread: 4.6.5 Reading Whole Lines
- ☐ I can trace a while loop that uses `hasNext` to read every item in a file. Reread: 4.6.6 Reading Until the File Runs Out
- ☐ I can place `close()` correctly and explain what happens if a closed `Scanner` is used. Reread: 4.6.7 Closing the File
- ☐ I can predict the array that `split` returns, including empty pieces and leftover spaces. Reread: 4.6.8 Splitting a Line into Pieces

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. The data lives in the file, which persists between runs. The program reads whatever the file holds each time it runs, so changing the file changes the data without changing the program.

- **A** is wrong: variables exist only while the program is running. When it ends they are gone, and the next run starts over.
- **C** is wrong: an array initializer puts data in the code, which is exactly what reading a file avoids.
- **D** is wrong: a file persists when no program is running. That is the whole reason to use one.

Reread *4.6.1 Files Keep Data Between Runs* if you chose differently.

Question 2

Three lines and six tokens: Lee, 12, Kim, 9, true, Ana.

The extra spaces before `Kim` and between `Kim` and `9` are whitespace, and any amount of whitespace counts as one separator. Line breaks also separate tokens, so `12` and `Kim` are separate tokens even though no space sits between them.

Reread *4.6.1 Files Keep Data Between Runs* if your count differed.

Question 3

```
8
143
```

Line	Call	Variable gets	Tokens left
10	<code>reader.nextInt()</code>	<code>a = 72</code>	68 75 80 77
11	<code>reader.nextInt()</code>	<code>b = 68</code>	75 80 77
12	<code>reader.nextInt()</code>	<code>c = 75</code>	80 77
13	<code>reader.nextInt()</code>	<code>d = 80</code>	77

`d - a` is `80 - 72 = 8`, and `b + c` is `68 + 75 = 143`. The fourth read crosses the line break between `75` and `80` without any special code, because a line break is just whitespace between tokens. The token `77` is never read, which is fine.

Reread *4.6.2 Opening a File with File and Scanner* if your output differed.

Question 4

B. The `File` constructor takes the file's name, and the `Scanner` constructor takes the resulting `File` object.

- **A** is wrong: it compiles, but it scans the characters `prices.txt` themselves, not the file.
- **C** is wrong: without `new`, Java reads `File(...)` as a call to a method named `File`, which does not exist. It does not compile (`error: cannot find symbol`). Creating any object takes `new` (Topic 1.13).
- **D** is wrong: it has the two classes backwards. A `File` is built from a name, and a `Scanner` is built from a `File`, not the other way around. It does not compile.

Reread 4.6.2 *Opening a File with File and Scanner* if you chose differently.

Question 5

Model answer: Classes in the `java.lang` package, such as `String` and `Math`, are available to every program automatically. `File` and `IOException` are in the `java.io` package and `Scanner` is in `java.util`, so the program must import them to use their short names. If `import java.io.File;` is missing, the compiler does not recognize the name `File` and reports `cannot find symbol for class File` at the first line that uses it.

A complete answer names the packages (`java.io` for `File` and `IOException`, `java.util` for `Scanner`), says only `java.lang` is automatic, and names the `cannot find symbol` error.

Reread 4.6.2 *Opening a File with File and Scanner* if your answer differed.

Question 6

C. Line 9 prints `Start`. Line 10 creates a `File` object, which only records the name, so it succeeds even though the file is missing. Line 11 prints `File object made`. Line 12 is where the `Scanner` actually tries to open the file, so that is where the `FileNotFoundException` happens and the program terminates.

- **A** is wrong: the compiler checks the code, not the disk. The code is correct, including `throws IOException`, so it compiles.
- **B** is wrong: creating a `File` object does not open the file. The failure comes at the `Scanner` constructor.
- **D** is wrong: line 13 and line 14 never run. Once the exception happens, `main` ends.

Reread 4.6.3 *When a File Cannot Be Opened* if you chose differently.

Question 7

Model answer: `main` calls `firstValue`, and `firstValue`'s header says it throws `IOException`, meaning it might pass a file problem back to whoever called it. Since `main` is the caller, that problem could arrive in `main`. Java requires `main` to either handle it or declare it, so `main` declares `throws IOException` too. Without it the program does not compile (unreported exception `IOException`).

A complete answer says that `firstValue` declares it may throw `IOException`, and that a method calling it must also declare it.

Reread 4.6.3 *When a File Cannot Be Opened* if your answer differed.

Question 8

```
Maya is online
105.5
```

Lines 10 to 13 read `who = "Maya"`, `level = 17`, `accuracy = 88.5`, and `online = true`, as in the worked example. Line 14's condition is `true`, so line 16 prints `Maya is online`. Line 18 adds a `double` and an `int`: `88.5 + 17` is the `double` `105.5`. (There is no string on either side of `+`, so this is addition, not joining text.)

Reread 4.6.4 *Reading Tokens from a File* if your output differed.

Question 9

C. `nextInt` succeeds only when the next token is a whole number that fits in an `int`. Three billion is larger than the largest `int` (2,147,483,647), so line 11 throws an `InputMismatchException`, and line 12 never runs.

- **A** is wrong: an `int` cannot hold three billion, and `nextInt` refuses the token instead of storing it.
- **B** is wrong: `nextInt` does not clamp a value to the largest `int`. It throws an exception.
- **D** is wrong: `next` returns any token as a `String`; it can never mismatch.

Reread 4.6.4 *Reading Tokens from a File* if you chose differently.

Question 10

```
27
Kindred
```

Line 10 stores the first line in `line`. Line 11 reads the second line and stores it in `line`, replacing the first. So `line` holds "Kindred,Octavia Butler,1979". Count its characters: Kindred (7) + , (1) + Octavia (7) + space (1) + Butler (6) + , (1) + 1979 (4) = 27. The line break is not included. `substring(0, 7)` returns indices 0 through 6, Kindred.

Reread 4.6.5 *Reading Whole Lines* if your output differed.

Question 11

Model answer: Each line is one address, and addresses do not all have three tokens.

12 Oak Hill Road has four, and the student's three calls would leave Road behind to be misread as the start of the next address. An address that starts with a word, such as Apartment 5, would even crash `nextInt()` with an `InputMismatchException`. `nextLine()` returns the whole line however many spaces it has, so each call reads exactly one address.

A complete answer mentions that each line is one record, that the number of tokens per line can vary, and that `nextLine()` keeps the spaces and reads exactly one line.

Reread 4.6.5 *Reading Whole Lines* if your answer differed.

Question 12

```
3 77
```

Pass	last	last > 74?	warmDays
1	72	no	0
2	68	no	0
3	75	yes	1
4	80	yes	2
5	77	yes	3

Then `hasNext()` is `false` and the loop ends. `last` was declared before the loop, so it still holds the final value read, 77, when line 21 prints.

Reread 4.6.6 *Reading Until the File Runs Out* if your output differed.

Question 13

Every call to `nextInt()` reads a new token, and the loop body calls it twice: once in the condition on line 12 and again on line 14. So the value that gets printed is not the value that was tested.

Pass	Line 12 reads	> 74?	Line 14 reads and prints
1	72	no	
2	68	no	
3	75	yes	80
4	77	yes	nothing left: crash

On pass 4, `hasNext()` was `true` because 77 was waiting, but line 12 used it up, so line 14 has nothing to read and throws a `NoSuchElementException`. Checking `hasNext()` once only guarantees **one** more token.

Corrected loop body: read once, store the value, then use the variable.

```
1 int temp = input.nextInt();
2 if (temp > 74)
3 {
4     System.out.println(temp);
5 }
```

Now each pass reads exactly one token, and the program prints 75, 80, and 77.

Reread 4.6.6 *Reading Until the File Runs Out* if your fix differed.

Question 14

C. The scanner must stay open for every read, and the last read happens on the final pass of the loop. Closing right after the loop is the earliest safe point.

- **A** is wrong: the first read would then use a closed scanner and throw an `IllegalStateException`.
- **B** is wrong: the first pass would close the scanner, and the loop's next `hasNext()` check would throw an `IllegalStateException`.
- **D** is wrong: a file is released when the program ends, but a program should close each file as soon as it is done with it rather than relying on that.

Reread 4.6.7 *Closing the File* if you chose differently.

Question 15

```
4
[, ]
green
```

The string has three commas, so it splits into four pieces:

Index	Piece
0	"red"
1	" "
2	"blue"

3	"green"
---	---------

Nothing sits between the first two commas, so `parts[1]` is the empty string, printed as `[]`. The last index is `parts.length - 1`, which is 3, so the last line prints `green`.

Reread *4.6.8 Splitting a Line into Pieces* if your output differed.

Question 16

B. The fields are separated by commas with no spaces, so splitting on `,` gives `"Ana Diaz"`, `"Lincoln High"`, and `"11"`.

- **A** is wrong: splitting on spaces gives `"Ana"`, `"Diaz, Lincoln"`, and `"High, 11"`, so index 1 is `"Diaz, Lincoln"`. This comes from thinking spaces are always the separator, as they are for token readers.
- **C** is wrong: the line never contains a comma followed by a space, so nothing matches and the array holds only the whole line.
- **D** is wrong: the delimiter is removed, so `"Lincoln"` would not appear in any piece. The pieces would be `"Ana Diaz,"` and `" High, 11"`.

Reread *4.6.8 Splitting a Line into Pieces* if you chose differently.

Question 17

A. The error is `cannot find symbol for class File`. `File` lives in `java.io`, so `import java.io.File;` makes it available. Everything else is already correct, so the program then opens `temps.txt` and prints its first value, 72.

- **B** is wrong: `throws IOException` is required whenever a method opens a file, whether or not the file exists. Removing it adds an unreported exception error and does not fix the missing import.
- **C** is wrong: it compiles, but it scans the text `temps.txt`. `nextInt()` then throws an `InputMismatchException`, because `temps.txt` is not a number.
- **D** is wrong: `File` is in `java.io`, not `java.util`. Only `Scanner` comes from `java.util`.

Reread *4.6.2 Opening a File with File and Scanner* and *4.6.3 When a File Cannot Be Opened* if you chose differently.

Question 18

```
quick
brown
jumps
Long words: 3
```

`next()` reads one word per pass; the line break between `fox` and `jumps` is whitespace, so the loop crosses it without noticing.

Word	Length	> 4?	LongCount
the	3	no	0
quick	5	yes	1
brown	5	yes	2
fox	3	no	2
jumps	5	yes	3

over	4	no	3
the	3	no	3
lazy	4	no	3
dog	3	no	3

over and lazy have exactly 4 letters, and $4 > 4$ is false.

Reread *4.6.6 Reading Until the File Runs Out* if your output differed.

Question 19

B. Line 10 reads Maya. Line 11 calls `nextDouble()` on the token 17, which is allowed: a whole number is a valid `double`, so `first` is 17.0. Line 12 prints Maya 17.0. Line 13 calls `nextInt()` on 88.5, which is not a whole number, so it throws an `InputMismatchException`, and line 14 never runs.

- **A** is wrong: `nextDouble` accepts whole numbers and returns them as `doubles`, as with 72.0 in 4.6.4.
- **C** is wrong: `nextInt` does not drop the decimal part. It refuses a token that is not a whole number.
- **D** is wrong: each call reads the *next* token; `nextDouble` does not search ahead for a token with a decimal point.

Reread *4.6.4 Reading Tokens from a File* if you chose differently.

Question 20

`input.close()` on line 14 is inside the loop body, so the first pass reads and welcomes Ada, then closes the scanner. The loop goes back to line 10, and `hasNext()` is called on a closed scanner, which throws an `IllegalStateException`. The crash comes from the loop condition, not from a `nextLine` call.

Corrected code: move the `close()` call below the loop's closing brace.

```
1 while (input.hasNext())
2 {
3     String name = input.nextLine();
4     System.out.println("Welcome, " + name);
5 }
6 input.close();
```

Now all three passes run, and the file is closed once, after the last read:

```
Welcome, Ada Lovelace
Welcome, Grace Hopper
Welcome, Alan Turing
```

Reread *4.6.7 Closing the File* if your fix differed.

Question 21

```
HBP
```

Each pass splits a line twice: first on commas to get the author, then the author on a space to get first and last name.

Pass	parts[1]	names	names[1].substring(0, 1)	result
1	"Frank Herbert"	{"Frank", "Herbert"}	"H"	"H"
2	"Octavia Butler"	{"Octavia", "Butler"}	"B"	"HB"
3	"Gary Paulsen"	{"Gary", "Paulsen"}	"P"	"HBP"

Line 13 calls `split` directly on the string `nextLine()` returns, which works because `nextLine()` produces a `String`. The program builds the initials of the authors' last names.

Reread 4.6.8 *Splitting a Line into Pieces* if your output differed.

Question 22

B. The string has two commas, so `split(",")` returns three pieces: "Lee", " Kim", and " Ana". The delimiter is only the comma, so the space after each comma stays at the start of the next piece. `parts[2]` is " Ana", four characters long.

- **A** is wrong: `split` matches the delimiter exactly and removes only the comma, not the space after it. Splitting on ", " would give [Ana] and 3.
- **C** is wrong: this counts the spaces as delimiters too, as token readers would. `split` uses only the delimiter you pass it.
- **D** is wrong: two commas make three pieces, so index 2 exists.

Reread 4.6.8 *Splitting a Line into Pieces* if you chose differently.

Question 23

75 2

Line 10 reads the count, 5, and uses it directly as the array size. Lines 11 to 14 read the five grades into `grades`, which becomes {88, 92, 75, 100, 81}. Line 15 closes the file.

`lowest` starts at `grades[0]`, 88, and the second loop starts at index 1:

i	grades[i]	< lowest?	> 90?	lowest	over90
1	92	no	yes	88	1
2	75	yes	no	75	1
3	100	no	yes	75	2
4	81	no	no	75	2

Starting at index 1 is safe for `lowest`, because `grades[0]` is already its starting value. It skips `grades[0]` for the `over90` count, which gives the right answer here only because 88 is not over 90.

Reread *Reading numbers into an array* in Putting it together if your output differed.

Question 24

Model answer: The `hasNext()` loop keeps working. It checks before each read whether another token remains, so it reads every value in the file, however many there are, and stops when the file runs out. The `for` loop always makes exactly five reads. Once the file has more than five values, it silently ignores the extra days. If the file ever has fewer than five, a `nextInt()` call runs out of data and throws a `NoSuchElementException`. Since the whole point of storing the data in a file is that it can change without changing the program, the loop should let the file decide how many values to read.

A complete answer says that `hasNext()` adapts to any length, that a fixed count misses extra values, and that it crashes with a `NoSuchElementException` when there are too few.

Reread *4.6.6 Reading Until the File Runs Out* if your answer differed.