Scanner

The third topic in the Java Basics category: `Scanner`'s token-based reading model, the classic `nextInt()` + `nextLine()` trap, custom delimiters via `useDelimiter()`, reading from a `File`, a real performance difference between `Scanner` and `BufferedReader`, and `InputMismatchException` / `NoSuchElementException` exception handling.

Beginner 20 min

Last updated: October 21, 2026
Read online: https://www.learnforgex.com/en/topics/scanner

Scanner

`Scanner` is the third topic in the Java Basics category -- the most common way to read input from the console, parse a file, or break apart (tokenize) a piece of text you already have. It looks like a simple utility class, but it hides subtle behaviors like the `nextInt()` / `nextLine()` mix-up -- a classic trap almost every Java student falls into at least once.

What Is Scanner?

`Scanner`, defined in the `java.util` package, is a class that splits a text source (console input, a `File`, a `String`, or any `InputStream` / `Readable`) into TOKENS and lets you read those tokens by converting them into primitive types (`int`, `double`, `boolean`, etc.) or `String`. By default, tokens are separated by WHITESPACE characters (space, tab, newline), but this behavior can be changed with a custom regex.

Why Does It Exist?

Reading raw bytes/characters one at a time from an `InputStream` or `BufferedReader` and manually parsing them with methods like `Integer.parseInt()` is tedious and error-prone. `Scanner` reduces that parsing work to a single method call (`nextInt()`, `nextDouble()`, etc.) -- especially in simple console applications ("enter a number", "enter your name"), it's the most practical way to read user input.

History

`Scanner` arrived with Java 5 (2004) -- part of the language's largest feature set at the time, alongside generics, enums, varargs, and autoboxing. Before that, reading console input

generally meant wrapping `BufferedReader` + `InputStreamReader` and parsing manually -- `Scanner` reduced that to a much more convenient API. Internally it uses a regex-based matching engine; this gives it flexibility (defining any regex delimiter via `useDelimiter()`) but makes it slower than raw reading (see the "Scanner vs BufferedReader" section).

Basic Usage: Reading Tokens

`Scanner` works with the same API on any text source (a `String`, `System.in`, a `File`). `next()` reads a word, `nextInt()` reads an integer, `nextDouble()` reads a decimal number -- each method CONVERTS the token it reads into the expected type. Methods like `hasNext()` / `hasNextInt()` let you check a token WITHOUT consuming it -- that's how you safely loop over an unknown number of tokens.

```
import java.util.Scanner;

public class ScannerBasicsExample {
    public static void main(String[] args) {
        // Scanner can read from ANY source that implements Readable, or from an
        // InputStream -- here we use a plain String as the source, but the same
        // API works identically for System.in (keyboard input) or a File.
        String input = "Alice 30 5.6 true";
        Scanner scanner = new Scanner(input);

        // By default, Scanner splits tokens on WHITESPACE and parses each token
        // according to the method you call: next() for a word, nextInt() for an
        // int, nextDouble() for a double, nextBoolean() for a boolean.
        String name = scanner.next();
        int age = scanner.nextInt();
        double height = scanner.nextDouble();
        boolean active = scanner.nextBoolean();

        System.out.println("name: " + name);
        System.out.println("age: " + age);
        System.out.println("height: " + height);
        System.out.println("active: " + active);

        scanner.close();

        // hasNext()/hasNextInt()/etc. let you check WITHOUT consuming -- the
        // safe way to loop through tokens of unknown count.
        Scanner tokenScanner = new Scanner("10 20 thirty 40");
        System.out.print("Looping with hasNextInt(): ");
        while (tokenScanner.hasNext()) {
            if (tokenScanner.hasNextInt()) {
                System.out.print(tokenScanner.nextInt() + " ");
            } else {
                System.out.print("[skipping non-int: " + tokenScanner.next() + "] ");
            }
        }
        System.out.println();
        tokenScanner.close();
    }
}
```

The Classic nextInt() + nextLine() Trap

`nextInt()` (or `nextDouble()`, `next()`, etc.) only consumes the number/word ITSELF -- it does NOT consume the newline character (`\n`) right after it, leaving it in the input stream. If you call `nextLine()` right afterward, that call reads up to that leftover `\n` -- meaning it returns an EMPTY string, NOT the next line you expected.

```
import java.io.ByteArrayInputStream;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class ScannerNextIntNextLinePitfallExample {
    public static void main(String[] args) {
        // Simulating typed console input: the user types "25", presses Enter,
        // then types "Alice" and presses Enter. This works identically with
        // `new Scanner(System.in)` reading real keyboard input.
        String simulatedTyping = "25\nAlice\n";
        InputStream fakeConsole = new ByteArrayInputStream(simulatedTyping.getBytes
(StandardCharsets.UTF_8));

        System.out.println("--- THE BUG ---");
        Scanner buggyScanner = new Scanner(fakeConsole);
        int age = buggyScanner.nextInt();
        // SURPRISE: nextInt() only consumes the digits "25", NOT the newline
        // character right after them -- that leftover "\n" is still sitting in
        // the input. The very next nextLine() call reads up to that leftover
        // newline, which means it reads an EMPTY string instead of "Alice".
        String name = buggyScanner.nextLine();
        System.out.println("age: " + age);
        System.out.println("name (BUG -- empty, not \"Alice\"): [" + name + "]");
        buggyScanner.close();

        System.out.println();
        System.out.println("--- THE FIX ---");
        InputStream fixedConsole = new ByteArrayInputStream(simulatedTyping.getBytes
(StandardCharsets.UTF_8));
        Scanner fixedScanner = new Scanner(fixedConsole);
        int age2 = fixedScanner.nextInt();
        fixedScanner.nextLine(); // consume the leftover newline left by nextInt()
        String name2 = fixedScanner.nextLine();
        System.out.println("age: " + age2);
        System.out.println("name (fixed): [" + name2 + "]");
        fixedScanner.close();
    }
}
```

Expected Output

```
--- THE BUG ---
age: 25
name (BUG -- empty, not "Alice"): []

--- THE FIX ---
age: 25
name (fixed): [Alice]
```

This is a classic trap almost every Java student falls into at least once: in a flow like "read an age, then read a name", the `nextLine()` right after `nextInt()` unexpectedly returns an empty string. The fix is simple: consume the leftover newline with an extra `nextLine()` call right after `nextInt()`, BEFORE the real `nextLine()` call.

Custom Delimiters

The default whitespace delimiter can be changed to ANY regex with `useDelimiter(regex)` -- this turns `Scanner` into a simple CSV parser or a tokenizer for custom-formatted text.

```
import java.util.Scanner;

public class ScannerDelimiterExample {
    public static void main(String[] args) {
        // By default the delimiter is whitespace -- useDelimiter() lets you
        // change it to any regex, which turns Scanner into a simple tokenizer
        // for structured text like CSV.
        String csvLine = "apple, banana,  cherry ,date";
        Scanner csvScanner = new Scanner(csvLine);
        csvScanner.useDelimiter("\\s*,\\s*"); // comma, optionally surrounded by spaces

        System.out.print("CSV fields: ");
        while (csvScanner.hasNext()) {
            System.out.print("[" + csvScanner.next() + "] ");
        }
        System.out.println();
        csvScanner.close();

        // A regex delimiter can be much richer than a single character -- here,
        // any run of non-digit characters separates the numbers.
        String messyNumbers = "12abc34##56   78xyz90";
        Scanner numberScanner = new Scanner(messyNumbers);
        numberScanner.useDelimiter("[^0-9]+");

        System.out.print("Extracted numbers: ");
        while (numberScanner.hasNextInt()) {
            System.out.print(numberScanner.nextInt() + " ");
        }
        System.out.println();
        numberScanner.close();
    }
}
```

Expected Output

```
CSV fields: [apple] [banana] [cherry] [date]
Extracted numbers: 12 34 56 78 90
```

> The regex you pass to `useDelimiter()` isn't limited to a single character -- in the example above, `"[^0-9]+"` (one or more non-digit characters) is used to extract only the numbers out of a messy piece of text. This gives far richer parsing power than simple single-character delimiters.

Reading from a File

`Scanner` has a constructor that takes a `File` object directly -- which means reading a file uses the SAME API as reading a `String` / `System.in`. Since `Scanner` holds a file handle underneath, it MUST be closed once you're done -- the safest way to do that is try-with-resources.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class ScannerFileExample {
    public static void main(String[] args) throws IOException {
        // Create a small temp file to read from, so this example is
        // self-contained and reproducible.
        File tempFile = File.createTempFile("scanner-demo", ".txt");
        tempFile.deleteOnExit();
        try (FileWriter writer = new FileWriter(tempFile)) {
            writer.write("Alice,30\n");
            writer.write("Bob,25\n");
            writer.write("Charlie,35\n");
        }

        // Scanner can read directly from a File -- try-with-resources makes
        // sure the underlying file handle is closed even if something throws.
        try (Scanner fileScanner = new Scanner(tempFile)) {
            int lineNumber = 1;
            while (fileScanner.hasNextLine()) {
                String line = fileScanner.nextLine();
                System.out.println("line " + lineNumber + ": " + line);
                lineNumber++;
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }

        // A second pass, this time parsing each line's fields with a nested
        // Scanner and a comma delimiter.
        System.out.println();
        System.out.println("Parsed fields:");
        try (Scanner fileScanner = new Scanner(tempFile)) {
            while (fileScanner.hasNextLine()) {
                String line = fileScanner.nextLine();
                Scanner lineScanner = new Scanner(line);
                lineScanner.useDelimiter(",");
                String name = lineScanner.next();
                int age = lineScanner.nextInt();
                System.out.println("  " + name + " is " + age + " years old");
                lineScanner.close();
            }
        }

        tempFile.delete();
    }
}
```

> `Scanner`'s `File`-accepting constructor can throw `FileNotFoundException` (a CHECKED exception) -- meaning you MUST handle a normal, expected condition like the file path being wrong. Also, forgetting to `close()` a `Scanner` leaves its underlying file resource open -- try-with-resources guarantees this automatically.

Scanner vs BufferedReader: Performance

`Scanner` does REGEX MATCHING internally when reading each token -- this provides a convenient API for parsing typed tokens (`nextInt()`, `nextDouble()`), but it comes at a cost in raw speed. `BufferedReader.readLine()` reads only raw lines without any parsing -- it's much faster if all you need is the text itself (no number/word parsing).

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.StringReader;
import java.util.Scanner;

public class ScannerVsBufferedReaderPerformanceExample {
    public static void main(String[] args) throws IOException {
        int lineCount = 50_000;
        StringBuilder textBuilder = new StringBuilder();
        for (int i = 0; i < lineCount; i++) {
            textBuilder.append("line-").append(i).append('\n');
        }
        String text = textBuilder.toString();

        // Warm-up -- run both paths a lot before measuring.
        for (int i = 0; i < 50; i++) {
            readWithScanner(text);
            readWithBufferedReader(text);
        }

        long scannerStart = System.nanoTime();
        int scannerLines = readWithScanner(text);
        long scannerNanos = System.nanoTime() - scannerStart;

        long readerStart = System.nanoTime();
        int readerLines = readWithBufferedReader(text);
        long readerNanos = System.nanoTime() - readerStart;

        System.out.println("Reading " + lineCount + " lines:");
        System.out.println("  Scanner.nextLine(): " + (scannerNanos / 1_000_000) + " ms ("
+ scannerLines + " lines)");
        System.out.println("  BufferedReader.readLine(): " + (readerNanos / 1_000_000) + "
ms (" + readerLines + " lines)");
        System.out.println("(Scanner does regex-based token matching under the hood --
convenient for parsing");
        System.out.println(" typed tokens (nextInt(), nextDouble()...), but that
flexibility costs raw speed.");
        System.out.println(" BufferedReader.readLine() just reads raw lines, with no
parsing -- much faster");
        System.out.println(" when all you need is the text itself.)");
    }
```

```
    private static int readWithScanner(String text) {
        int lines = 0;
        Scanner scanner = new Scanner(text);
        while (scanner.hasNextLine()) {
            scanner.nextLine();
            lines++;
        }
        scanner.close();
        return lines;
    }

    private static int readWithBufferedReader(String text) throws IOException {
        int lines = 0;
        try (BufferedReader reader = new BufferedReader(new StringReader(text))) {
            while (reader.readLine() != null) {
                lines++;
            }
        }
        return lines;
    }
}
```

Real measurement (warmed up -- both paths were run 50 times before timing): reading a 50,000-line text, `Scanner.nextLine()` consistently took ~6 ms, while `BufferedReader.readLine()` took ~1 ms -- confirming that `Scanner`'s regex-based flexibility has a real speed cost.

Exception Handling

Calling `nextInt()` (or similar) for a token that isn't the expected type throws a REAL `InputMismatchException` -- it does NOT silently return `0` or `null`. A critical point: when this exception is thrown, that "mismatched" token is NOT consumed -- it stays in the input stream, so you can recover by reading it as a plain string with `next()`. When there are no tokens left at all, `NoSuchElementException` is thrown instead -- the safe pattern is always checking with `hasNext()`/`hasNextInt()` first (exactly like the difference between `offer()`/`poll()` and `add()`/`remove()` in the "Queues & Collections Utility" lesson).

```
import java.util.InputMismatchException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class ScannerExceptionHandlingExample {
    public static void main(String[] args) {
        // Calling nextInt() when the next token is NOT a valid int throws a
        // real InputMismatchException -- it does not return 0 or null.
        Scanner scanner = new Scanner("42 not-a-number 100");
        System.out.println("First nextInt(): " + scanner.nextInt());

        try {
            int oops = scanner.nextInt();
            System.out.println("unreachable: " + oops);
        } catch (InputMismatchException e) {
            System.out.println("Caught: " + e.getClass().getSimpleName() + " on the second
token");
        }

        // IMPORTANT: after an InputMismatchException, the bad token is NOT
        // consumed -- it's still sitting there waiting to be read (as a plain
        // token via next()), which is exactly how you recover from it.
        System.out.println("Recovering with next(): " + scanner.next());
        System.out.println("Then nextInt() again: " + scanner.nextInt());
        scanner.close();

        // Calling next()/nextInt() when there are NO MORE tokens throws
        // NoSuchElementException -- the SAFE pattern is always checking
        // hasNext()/hasNextInt() first, exactly like the Queue lesson's
        // offer()/poll() vs add()/remove() distinction (checkable result vs.
        // exception for a normal "nothing left" condition).
        Scanner emptyScanner = new Scanner("");
        try {
            emptyScanner.next();
        } catch (NoSuchElementException e) {
            System.out.println("Caught: " + e.getClass().getSimpleName() + " on an
exhausted Scanner");
        }
        System.out.println("Safe check first -- hasNext(): " + emptyScanner.hasNext());
        emptyScanner.close();
    }
}
```

Expected Output

```
First nextInt(): 42
Caught: InputMismatchException on the second token
Recovering with next(): not-a-number
Then nextInt() again: 100
Caught: NoSuchElementException on an exhausted Scanner
Safe check first -- hasNext(): false
```

Best Practices

- **If you're going to call `nextLine()` after `nextInt()` / `nextDouble()`, add an extra `nextLine()` to consume the leftover newline in between** -- this avoids the most classic `Scanner` mistake.
- **When looping over an unknown number of tokens, check with `hasNext()` / `hasNextInt()` first**, don't call `next()` / `nextInt()` directly and risk `NoSuchElementException`.
- **Always close a `Scanner` once you're done with it (`close()`)** -- use try-with-resources, especially when working with a file or stream.
- **Use `BufferedReader` if you only need raw text lines (no number/token parsing needed)** -- `Scanner`'s regex flexibility is an unnecessary performance cost unless it's actually worth its simplicity for your use case.

Common Mistakes

- **Calling `nextLine()` right after `nextInt()` and unexpectedly getting an empty string.** `nextInt()` doesn't consume the newline that follows it -- it needs to be cleared with an extra `nextLine()`.
- **Calling `next()` / `nextInt()` without checking `hasNext()` / `hasNextInt()` first and getting `NoSuchElementException`.** Whether input has run out should always be checked first.
- **Forgetting to close a `Scanner`.** Especially with file/stream-backed `Scanner`s, this leads to a resource leak.
- **Assuming the "mismatched" token is consumed after an `InputMismatchException`.** It's actually still in the input stream -- it can be recovered by reading it with `next()`, otherwise the next call will fail on the SAME token again.

Summary, Cheat Sheet, and Glossary

`Scanner` is a class that splits a text source (console, file, string) into tokens and lets you read them by converting them into primitive types/`String`. `nextInt()` not consuming the newline that follows it is the most classic `Scanner` trap. Custom delimiters can be defined with `useDelimiter()`; reading a file uses the same API as reading a `String`/`System.in`. `Scanner` is slower than `BufferedReader` (due to regex-based parsing), but offers a much more convenient typed-reading API.

Quick reference:

```
Scanner scanner = new Scanner(System.in);        // reading from the console
int age = scanner.nextInt();
scanner.nextLine();                                 // CLASSIC TRAP: consume the leftover \n
String name = scanner.nextLine();

Scanner csv = new Scanner(text);                    // custom delimiter
csv.useDelimiter(",");

try (Scanner file = new Scanner(new File("data.txt"))) {  // reading from a file
    while (file.hasNextLine()) {
        String line = file.nextLine();
    }
}

if (scanner.hasNextInt()) { ... }                          // safe check
```

Glossary

Scanner — A `java.util` class that splits a text source into tokens and reads them as primitive types/`String`.

Token — A single readable unit (word, number, etc.) that `Scanner` splits based on a delimiter (default: whitespace).

Delimiter — The regex that separates tokens from each other, customizable via `useDelimiter()`.

InputMismatchException — The exception thrown when a typed `next...()` method is called for a token that doesn't match the expected type.

BufferedReader — A class that reads raw text lines (without parsing), a faster alternative to `Scanner`.