File Reading

Java Basics kategorisinin beşinci topic'i (File I/O'nun okuma yarısı): `Path` / `Files` temelleri, `Files.readAllLines()` / `Files.readString()`, `BufferedReader` ile satır satır okuma, `Files.lines()`'in lazy ve Closeable olması, dosyada kelime arama, ve `NoSuchFileException` ile `FileNotFoundException` arasındaki gerçek fark.

Başlangıç 20 dk

Son güncelleme: 31 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/file-reading

---

File Reading

Java Temelleri kategorisinin altıncı konusu File I/O -- iki topic'e bölündü: bu topic OKUMA (`File Reading`), bir sonraki topic ise YAZMA (`File Writing`) tarafını kapsıyor. Bu ayrım, `functional-interfaces-streams` ve `collections` kategorilerinde uygulanan "kısa, bağımsız alt konuları TEK topic'te birleştirme" pratiğinin tam tersi bir karar -- burada tersine, tek bir konu (File I/O) YETERİNCE GENİŞ olduğu için iki topic'e AYRILDI.

File I/O Nedir?

File I/O (dosya girdi/çıktı), bir programın diskteki dosyalarla -- okuma, yazma, kopyalama, silme -- etkileşime girmesidir. Java'da bu iş için İKİ farklı API bir arada kullanılır: eski `java.io` paketi (`FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter` gibi AKIŞ (stream) tabanlı sınıflar) ve modern `java.nio.file` paketi (`Path`, `Files` gibi YOL (path) tabanlı, çoğu işlemi tek satırlık statik metotlara indirgeyen sınıflar).

Neden Var?

Bir program, kalıcı veri okumadan/yazmadan gerçek dünyada pek işe yaramaz -- log dosyaları, yapılandırma dosyaları, CSV raporları, kullanıcı tarafından yüklenen belgeler, hepsi File I/O gerektirir. `java.nio.file` (NIO.2), eski `java.io`'nun bazı can sıkıcı yönlerini (checked exception yönetiminin karmaşıklığı, sembolik bağlantı desteğinin eksikliği, dosya sistemi meta verisine erişimin zorluğu) çözmek için tasarlandı -- ama `java.io`'nun `BufferedReader` gibi bazı sınıfları hâlâ yaygın kullanılıyor, özellikle satır satır işleme gerektiren durumlarda.

Tarihçe

`java.io` paketi Java'nın 1.0 sürümünden (1996) beri var -- akış tabanlı (stream-based) klasik model. `java.nio` (Non-blocking I/O), Java 1.4 (2002) ile geldi ama asıl dosya sistemi API'si olan `java.nio.file` (`Path` / `Files`, "NIO.2" olarak da bilinir) Java 7 (2011) ile eklendi -- daha okunabilir bir API, gerçek istisna tipleri (`NoSuchFileException` gibi), ve dosya sistemi işlemlerini (kopyalama, taşıma, sembolik bağlantı) doğrudan destekleyen statik metotlar sundu. Java 11 (2018), `Files.readString()` / `Files.writeString()` ile tüm dosyayı tek bir `String` olarak okuma/yazmayı daha da basitleştirdi.

Path ve Files Temelleri

`Path.of(...)`, bir dosya konumunu TEMSİL EDEN bir nesne oluşturur -- ama DOSYA SİSTEMİNE dokunmaz, yalnızca bir "adres"tir. `Files.exists()` gibi metotlar gerçekten dosya sistemini kontrol eder. `Files.readAllLines(path)`, tüm dosyayı belleğe okuyup her satırı bir eleman olarak içeren bir `List<String>` döner -- küçük/orta boyutlu metin dosyaları için en basit okuma yoludur.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class PathAndFilesBasicsExample {
    public static void main(String[] args) throws IOException {
        // Path.of() builds a platform-independent file path -- it does NOT
        // touch the filesystem by itself, it's just a representation of a
        // location.
        Path path = Path.of("demo-input.txt");

        // Files.exists() actually checks the filesystem.
        System.out.println("Exists before creating it? " + Files.exists(path));

        // Files.writeString() creates the file (see the "File Writing" lesson
        // for the full picture) -- used here just to set up something to read.
        Files.writeString(path, "Hello World\nJava is great\nPractice makes perfect\nHello
again\nJava rocks!");
        System.out.println("Exists after creating it? " + Files.exists(path));

        // Files.readAllLines() reads the ENTIRE file into memory as a
        // List<String>, one element per line -- the simplest way to read a
        // small-to-medium text file.
        List<String> lines = Files.readAllLines(path);
        System.out.println("Number of lines read: " + lines.size());
        lines.forEach(System.out::println);

        // Path carries useful metadata methods too.
        System.out.println("getFileName(): " + path.getFileName());
        System.out.println("toAbsolutePath(): " + path.toAbsolutePath());

        Files.deleteIfExists(path);
    }
}
```

BufferedReader ile Satır Satır Okuma

`BufferedReader`, klasik `java.io` yoludur -- bir `FileReader`'ı SARAR (wrap eder) ve okumaları içeride TAMPONLAR (buffer), bu da tek tek karakter okumaktan çok daha hızlıdır. `readLine()`, dosyanın sonuna gelindiğinde tam olarak BİR KEZ `null` döner -- bu, döngünün doğal bitiş koşuludur. `BufferedReader` gerçek bir dosya tanıtıcısı (file handle) tuttuğu için try-with-resources İÇİNDE kullanılmalıdır.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-buffered.txt");
        Files.writeString(path, "line one\nline two\nline three");

        // BufferedReader is the classic java.io way to read a file line by
        // line -- it wraps a FileReader and buffers reads internally, which is
        // much faster than reading one character at a time.
        //
        // try-with-resources guarantees close() is called even if an
        // exception is thrown mid-read -- this matters because BufferedReader
        // holds a real file handle open until it's closed.
        try (BufferedReader reader = new BufferedReader(new FileReader(path.toFile()))) {
            String line;
            int lineNumber = 1;
            // readLine() returns null exactly once, when there's nothing left
            // to read -- that's the loop's natural termination condition.
            while ((line = reader.readLine()) != null) {
                System.out.println(lineNumber + ": " + line);
                lineNumber++;
            }
        }

        Files.deleteIfExists(path);
    }
}
```

Beklenen Çıktı

```
1: line one
2: line two
3: line three
```

`while ((line = reader.readLine()) != null) { ... }` deseni, atama VE karşılaştırmayı tek bir ifadede birleştirir -- Java'da yaygın, idiomatik bir okuma döngüsü kalıbıdır.

Satır Sayma: readAllLines() vs Files.lines()

Bir dosyadaki satır sayısını bulmanın iki yolu vardır: `Files.readAllLines(path).size()` (tüm dosyayı belleğe yükler, küçük dosyalar için sorunsuz) ya da `Files.lines(path).count()` (LAZY -- tembel -- bir `Stream<String>` döner, dosyayı TÜMÜYLE belleğe yüklemeden okur, çok büyük dosyalar için ölçeklenebilir).

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.stream.Stream;

public class FileReadingStreamAndCountExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-count.txt");
        Files.writeString(path, "one\ntwo\nthree\nfour\nfive");

        // Simplest way to count lines: read everything into a List, then ask
        // its size. Fine for small files, but it loads the WHOLE file into
        // memory just to get a count.
        long countViaReadAllLines = Files.readAllLines(path).size();
        System.out.println("Count via Files.readAllLines().size(): " +
countViaReadAllLines);

        // Files.lines() returns a LAZY Stream<String> -- it doesn't load the
        // whole file at once, it reads as the stream is consumed. This scales
        // to files far larger than available memory.
        //
        // CRITICAL: Files.lines() opens a real file handle under the hood, so
        // the Stream it returns is Closeable and MUST be used inside
        // try-with-resources -- forgetting to close it leaks a file handle,
        // exactly like forgetting to close a Scanner or BufferedReader.
        long countViaLines;
        try (Stream<String> lineStream = Files.lines(path)) {
            countViaLines = lineStream.count();
        }
        System.out.println("Count via Files.lines().count() (lazy, must be closed): " +
countViaLines);

        Files.deleteIfExists(path);
    }
}
```

Beklenen Çıktı

```
Count via Files.readAllLines().size(): 5
Count via Files.lines().count() (lazy, must be closed): 5
```

`Files.lines()`'ın döndürdüğü `Stream<String>`, altta gerçek bir dosya tanıtıcısı tutar -- yani bir `Scanner` ya da `BufferedReader` gibi KAPATILMASI (`close()`) gerekir. Bunu try-with-resources OLMADAN kullanmak (`Files.lines(path).count()` tek satırda) dosya tanıtıcısını sızdırır -- `Stream`'in `Closeable` olduğunu unutmak yaygın bir hatadır.

Dosyada Kelime Arama

`Files.readAllLines()`'ı Stream API ile birleştirmek (bkz. "Stream API Temelleri" dersi), bir dosyada anahtar kelime aramayı tek satıra indirger: satırları FİLTRELE, yalnızca kelimeyi içerenleri tut.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class SearchWordInFileExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-search.txt");
        Files.writeString(path,
                "Hello World\nJava is great\nPractice makes perfect\nHello again\nJava
rocks!");

        // Combining Files.readAllLines() with the Stream API (see the "Stream
        // Fundamentals" lesson) makes searching a file for a keyword a
        // one-liner: filter the lines, keep only the ones that contain the
        // word.
        List<String> matches = Files.readAllLines(path).stream()
                .filter(line -> line.contains("Java"))
                .toList();

        System.out.println("Lines containing \"Java\":");
        matches.forEach(line -> System.out.println("  " + line));
        System.out.println("Match count: " + matches.size());

        // Case-insensitive search is a small variation -- lowercase both
        // sides before comparing.
        List<String> caseInsensitiveMatches = Files.readAllLines(path).stream()
                .filter(line -> line.toLowerCase().contains("hello"))
                .toList();
        System.out.println("Lines containing \"hello\" (case-insensitive): " +
caseInsensitiveMatches);

        Files.deleteIfExists(path);
    }
}
```

Beklenen Çıktı

```
Lines containing "Java":
  Java is great
  Java rocks!
Match count: 2
Lines containing "hello" (case-insensitive): [Hello World, Hello again]
```

Dosyanın Tamamını String Olarak Okumak

`Files.readString()` (Java 11+), TÜM dosyayı tek bir `String`'e okur -- satır sonları DAHİL. `Files.readAllLines()`'ın aksine (satır sonlarını atıp bir `List` döner), ham metnin kendisine ihtiyacınız olduğunda (örneğin bir JSON ayrıştırıcıya geçirmek ya da olduğu gibi göstermek için) `readString()` daha uygundur.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class ReadFileAsStringExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-whole-file.txt");
        Files.writeString(path, "Line A\nLine B\nLine C");

        // Files.readString() (Java 11+) reads the ENTIRE file into a single
        // String, newlines and all -- the simplest option when you need the
        // raw text as one value (for example, to pass to a JSON parser or
        // display as-is), rather than a List<String> of separate lines.
        String wholeFile = Files.readString(path);
        System.out.println("Whole file as one String:");
        System.out.println(wholeFile);
        System.out.println("Total length: " + wholeFile.length() + " characters");

        // Files.readAllLines() vs Files.readString(): readAllLines() strips
        // the line separators and gives you a List; readString() keeps them
        // and gives you one String. Pick based on whether you need to work
        // line by line or need the raw text.
        System.out.println();
        System.out.println("Contains newline characters? " + wholeFile.contains("\n"));

        Files.deleteIfExists(path);
    }
}
```

Beklenen Çıktı

```
Whole file as one String:
Line A
Line B
Line C
Total length: 20 characters

Contains newline characters? true
```

İstisna Yönetimi: NoSuchFileException vs FileNotFoundException

Modern `java.nio.file` API'sinde (`Files.readString()` , `Files.readAllLines()` vb.) eksik bir dosya `NoSuchFileException` fırlatır -- klasik `java.io` 'nun `FileNotFoundException` 'ı DEĞİL. Bu ikisi, ikisi de `IOException` 'ı genişletse de, BİRBİRİYLE İLİŞKİSİZ (sibling) istisna sınıflarıdır. `FileNotFoundException` , `FileReader` / `FileInputStream` gibi klasik `java.io` sınıflarından gelir.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.NoSuchFileException;
import java.nio.file.Path;
```

```
public class FileReadingExceptionHandlingExample {
    public static void main(String[] args) {
        Path missing = Path.of("this-file-does-not-exist.txt");

        // SURPRISE: for the modern java.nio.file API (Files.readString(),
        // Files.readAllLines(), etc.), a missing file throws
        // NoSuchFileException, NOT the classic java.io FileNotFoundException.
        // The two are UNRELATED sibling exceptions (both extend IOException),
        // even though they mean the same thing in practice.
        try {
            Files.readString(missing);
        } catch (NoSuchFileException e) {
            System.out.println("Caught NoSuchFileException (the java.nio.file exception): "
+ e.getFile());
        } catch (IOException e) {
            System.out.println("unreachable for this case: " + e.getClass().
getSimpleName());
        }

        // A multi-catch of BOTH exception types still compiles and is
        // harmless -- but for a Files.* call, the FileNotFoundException
        // branch will never actually fire, because Files.* never throws it.
        try {
            String content = Files.readString(missing);
            System.out.println("unreachable: " + content);
        } catch (FileNotFoundException e) {
            System.out.println("unreachable: FileNotFoundException never comes from Files.
*");
        } catch (NoSuchFileException e) {
            System.out.println("This branch fires instead, for the SAME missing-file case");
        } catch (IOException e) {
            System.out.println("unreachable: " + e.getClass().getSimpleName());
        }

        // FileNotFoundException DOES come from the legacy java.io classes,
        // like the FileReader used in the "BufferedReader" section.
        try {
            new java.io.FileReader(missing.toFile());
        } catch (FileNotFoundException e) {
            System.out.println("Caught FileNotFoundException (the classic java.io
exception, from FileReader)");
        }

        // The safe, general pattern: catch IOException as a fallback for
        // anything that isn't specifically handled above.
        String result = readSafely(missing);
        System.out.println("Safe helper result: " + result);
    }

    private static String readSafely(Path path) {
        try {
            return Files.readString(path);
        } catch (NoSuchFileException e) {
            return "File not found: " + path;
        } catch (IOException e) {
            return "Error reading file: " + e.getMessage();
        }
```

```
    }
}
```

> `Files.readString()` / `Files.readAllLines()` için `catch (FileNotFoundException | NoSuchFileException e)` yazmak DERLENİR ama `FileNotFoundException` dalı BU ÇAĞRI İÇİN asla tetiklenmez -- çünkü `Files.*` metotları bu istisnayı hiç fırlatmaz, yalnızca `NoSuchFileException` fırlatır. Hangi API'yi (java.io mu, java.nio.file mi) kullandığınıza göre DOĞRU istisna tipini yakalamak önemlidir; emin değilseniz genel `IOException`'ı yakalamak her zaman güvenlidir.

Best Practices

- **Küçük/orta boyutlu dosyalar için `Files.readAllLines()` / `Files.readString()` kullanın** -- tek satırlık, okunabilir bir API sunar; çok büyük dosyalar için `Files.lines()` (try-with-resources İÇİNDE) tercih edin.
- **`BufferedReader` / `Files.lines()` gibi dosya tanıtıcısı tutan her kaynağı try-with-resources İÇİNDE kullanın** -- kapatmayı unutmak kaynak sızıntısına yol açar.
- **`java.nio.file` API'si kullanıyorsanız `NoSuchFileException` yakalayın, `FileNotFoundException` DEĞİL** -- yanlış istisna tipini yakalamak, o dal hiçbir zaman tetiklenmediği için sessiz bir hataya yol açar.
- **Bir dosyada arama/filtreleme yaparken `Files.readAllLines().stream().filter(...)` desenini kullanın** -- okunabilir ve Stream API'nin gücünden yararlanır.

Yaygın Hatalar

- **`Files.readString()` / `Files.readAllLines()` için `FileNotFoundException` yakalayıp hiçbir zaman tetiklenmediğini fark etmemek.** Bu API'ler `NoSuchFileException` fırlatır -- doğru istisna tipi yakalanmalı.
- **`Files.lines()`'ı try-with-resources olmadan kullanmak.** Döndürdüğü `Stream`, altta gerçek bir dosya tanıtıcısı tutar -- kapatılmazsa kaynak sızar.
- **Çok büyük bir dosyayı `Files.readAllLines()` ile okumaya çalışıp bellek taşması yaşamak.** Büyük dosyalar için `Files.lines()` (lazy) tercih edilmeli.
- **`BufferedReader`'ı kapatmayı unutmak.** try-with-resources kullanılmadığında, altındaki dosya tanıtıcısı açık kalır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Java'da dosya okumak için iki API vardır: klasik `java.io` (`BufferedReader` + `FileReader`, satır satır okuma için) ve modern `java.nio.file` (`Path` + `Files`, `readAllLines()` / `readString()` /

`lines()` ile çoğu işlemi tek satıra indirger). `Files.lines()` LAZY'dir ve `Closeable`'dır -- try-with-resources gerektirir. Eksik dosyalar için modern API `NoSuchFileException`, klasik API `FileNotFoundException` fırlatır -- bunlar İLİŞKİSİZ sınıflardır.

Hızlı referans:

```
Path path = Path.of("data.txt");                          // yol oluşturma (dosyaya
dokunmaz)
List<String> lines = Files.readAllLines(path);               // tüm dosyayı List olarak oku
String content = Files.readString(path);                       // tüm dosyayı tek String
olarak oku

try (Stream<String> s = Files.lines(path)) {                      // LAZY, try-with-
resources ŞART
    long count = s.count();
}

try (BufferedReader r = new BufferedReader(new FileReader(path.toFile()))) {  // klasik
satır satır okuma
    String line;
    while ((line = r.readLine()) != null) { ... }
}

try { Files.readString(path); }
catch (NoSuchFileException e) { ... }                          // java.nio.file -- DOĞRU
istisna
```

Terimler Sözlüğü

Path — Bir dosya konumunu temsil eden, dosya sistemine dokunmayan bir nesne (`java.nio.file.Path`).

Files — `java.nio.file` paketinin, dosya işlemleri için statik yardımcı metotlar sunan sınıfı.

BufferedReader — Bir okuma kaynağını (örn. `FileReader`) saran, okumaları tamponlayan klasik `java.io` sınıfı.

NoSuchFileException — `java.nio.file` API'sinin, eksik bir dosya için fırlattığı istisna (klasik `FileNotFoundException`'dan FARKLI bir sınıf).

Try-with-Resources — Bir kaynağın (dosya tanıtıcısı gibi) blok sonunda otomatik kapatılmasını garanti eden Java sözdizimi.