Scanner

Java Basics kategorisinin üçüncü topic'i: `Scanner`'ın token tabanlı okuma modeli, klasik `nextInt()` + `nextLine()` tuzağı, `useDelimiter()` ile özel ayırıcılar, `File` üzerinden okuma, `Scanner` ile `BufferedReader` arasındaki gerçek bir performans farkı, ve `InputMismatchException` / `NoSuchElementException` istisna yönetimi.

Başlangıç 20 dk

Son güncelleme: 21 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/scanner

Scanner

Java Temelleri kategorisinin üçüncü konusu `Scanner` -- kullanıcıdan konsoldan girdi almanın, bir dosyayı okumanın ya da elinizdeki bir metni parçalara ayırmanın (tokenize etmenin) en yaygın yolu. Basit bir yardımcı sınıf gibi görünse de, `nextInt()` / `nextLine()` karışıklığı -- neredeyse her Java öğrencisinin bir kez düştüğü klasik bir tuzak -- gibi ince davranışlar barındırır.

Scanner Nedir?

`Scanner`, `java.util` paketinde tanımlı, bir metin kaynağını (konsol girdisi, bir `File`, bir `String`, ya da herhangi bir `InputStream` / `Readable`) TOKEN'lara (jetonlara) ayırıp bu token'ları ilkel tiplere (`int`, `double`, `boolean` vb.) ya da `String`'e dönüştürerek okumanızı sağlayan bir sınıftır. Varsayılan olarak token'lar BOŞLUK karakterleriyle (space, tab, newline) ayrılır, ama bu davranış özel bir regex ile de değiştirilebilir.

Neden Var?

Ham bir `InputStream`'den ya da `BufferedReader`'dan tek tek byte/karakter okuyup bunları elle `Integer.parseInt()` gibi metotlarla ayrıştırmak sıkıcı ve hataya açık bir iştir. `Scanner`, bu ayrıştırma işini tek bir metot çağrısına (`nextInt()`, `nextDouble()` vb.) indirger -- özellikle basit konsol uygulamalarında ("bir sayı gir", "adını gir" gibi) kullanıcı girdisini okumanın en pratik yoludur.

Tarihçe

`Scanner`, Java 5 (2004) ile birlikte geldi -- generic'ler, enum'lar, varargs ve autoboxing gibi dilin en büyük özellik setinin bir parçası olarak. Ondan önce, konsoldan girdi okumak için genellikle `BufferedReader` + `InputStreamReader` sarmalayıp elle ayrıştırma yapmak gerekiyordu -- `Scanner` bunu çok daha rahat bir API'ye indirgedi. İçeride regex tabanlı bir eşleştirme motoru kullanır; bu esneklik (`useDelimiter()` ile herhangi bir regex ayırıcı tanımlayabilme) sağlar ama ham okumadan daha yavaştır (bkz. "Scanner vs BufferedReader" bölümü).

Temel Kullanım: Token Okumak

`Scanner`, herhangi bir metin kaynağı (bir `String`, `System.in`, bir `File`) üzerinde aynı API ile çalışır. `next()` bir kelime, `nextInt()` bir tamsayı, `nextDouble()` bir ondalık sayı okur -- her metot, okuduğu token'ı beklenen tipe DÖNÜŞTÜRÜR. `hasNext()` / `hasNextInt()` gibi metotlar, bir token'ı TÜKETMEDEN önce kontrol etmenizi sağlar -- bilinmeyen sayıda token üzerinde güvenle döngü kurmanın yolu budur.

```
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();
    }
}
```

nextInt() + nextLine() Klasik Tuzağı

`nextInt()` (ya da `nextDouble()`, `next()` vb.), yalnızca sayının/kelimenin KENDİSİNİ tüketir -- ardından gelen satır sonu karakterini (`\n`) TÜKETMEZ, girdi akışında bırakır. Hemen ardından bir `nextLine()` çağrısı yaparsanız, bu çağrı o bırakılan `\n`'ye kadar okur -- yani BOŞ bir string döner, beklediğiniz bir sonraki satır DEĞİL.

```
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();
    }
}
```

Beklenen Çıktı

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

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

> Bu, neredeyse her Java öğrencisinin en az bir kez düştüğü klasik bir tuzaktır: "yaş oku, sonra isim oku" gibi bir akışta `nextInt()`'ten hemen sonra gelen `nextLine()` beklenmedik şekilde boş bir string döner. Çözüm basittir: `nextInt()`'ten sonra, gerçek `nextLine()` çağrısından ÖNCE fazladan bir `nextLine()` çağrısı yaparak bırakılan satır sonunu "tüketin".

Özel Ayırıcılar (Delimiters)

Varsayılan boşluk ayırıcısı `useDelimiter(regex)` ile HERHANGİ bir regex'e değiştirilebilir -- bu, `Scanner`'ı basit bir CSV ayrıştırıcısına ya da özel formatlı metinler için bir tokenizer'a dönüştürür.

```
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();
    }
}
```

Beklenen Çıktı

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

> `useDelimiter()` 'a verdiğiniz regex, tek bir karakterle sınırlı değildir -- yukarıdaki örnekte `"[^0-9]+"` (rakam olmayan bir ya da daha fazla karakter) kullanılarak, karışık bir metinden yalnızca sayılar ayıklanıyor. Basit tek karakterlik ayırıcılardan çok daha zengin bir ayrıştırma gücü sağlar.

Dosyadan Okuma

`Scanner`, doğrudan bir `File` nesnesi alan bir constructor'a sahiptir -- bu sayede dosya okumak, `String` / `System.in` okumakla AYNI API ile yapılır. `Scanner`, altta bir dosya tanıtıcısı (file handle) tuttuğu için, işiniz bittiğinde MUTLAKA kapatılmalıdır -- bunun en güvenli yolu try-with-resources'tır.

```
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`'ın `File` alan constructor'ı `FileNotFoundException` (bir CHECKED exception) fırlatabilir -- bu, dosya yolunun yanlış olması gibi normal, beklenebilir bir durumu ele almanız GEREKTİĞİ anlamına gelir. Ayrıca `Scanner`'ı `close()` etmeyi unutmak, altındaki dosya kaynağının açık kalmasına yol açar -- try-with-resources bunu otomatik garanti eder.

Scanner vs BufferedReader: Performans

`Scanner`, her token'ı okurken içeride REGEX EŞLEŞTİRMESİ yapar -- bu, tipli token ayrıştırma (`nextInt()`, `nextDouble()`) için rahat bir API sağlar ama ham hız açısından bir bedeli vardır. `BufferedReader.readLine()`, hiçbir ayrıştırma yapmadan yalnızca ham satırları okur -- yalnızca metnin kendisine ihtiyacınız varsa (sayı/kelime ayrıştırması gerekmiyorsa) çok daha hızlıdır.

```
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;
    }
}
```

Gerçek ölçüm (ısıtılmış -- her iki yol da ölçümden önce 50 kez çalıştırıldı): 50.000 satırlık bir metni okurken `Scanner.nextLine()` tutarlı şekilde ~6 ms sürdü, `BufferedReader.readLine()` ise ~1 ms -- `Scanner`'ın regex tabanlı esnekliğinin gerçek bir hız maliyeti olduğunu doğruladı.

İstisna Yönetimi

Beklenen tipte olmayan bir token için `nextInt()` (ya da benzerleri) çağırmak GERÇEK bir `InputMismatchException` fırlatır -- sessizce `0` ya da `null` DÖNMEZ. Kritik bir nokta: bu istisna fırlatıldığında, o "uyumsuz" token TÜKETİLMEZ -- girdi akışında kalır, bu yüzden `next()` ile düz bir string olarak okunarak kurtarma yapılabilir. Hiç token kalmadığında ise `NoSuchElementException` fırlatılır -- güvenli desen, her zaman önce `hasNext()` / `hasNextInt()` ile kontrol etmektir (tıpkı "Queues & Collections Utility" dersindeki `offer()` / `poll()` ile `add()` / `remove()` arasındaki farkta olduğu gibi).

```
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();
    }
}
```

Beklenen Çıktı

```
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

- **`nextInt()` / `nextDouble()`'dan sonra `nextLine()` çağıracaksanız, aradaki bırakılan satır sonunu tüketmek için fazladan bir `nextLine()` ekleyin** -- bu, en klasik `Scanner` hatasını önler.
- **Belirsiz sayıda token üzerinde döngü kurarken önce `hasNext()` / `hasNextInt()` ile kontrol edin**, doğrudan `next()` / `nextInt()` çağırıp `NoSuchElementException` riske atmayın.
- **Bir `Scanner`'ı işiniz bittiğinde her zaman kapatın (`close()`)** -- özellikle bir dosya ya da akış üzerinde çalışıyorsanız, try-with-resources kullanın.
- **Yalnızca ham metin satırlarına ihtiyacınız varsa (sayı/token ayrıştırması gerekmiyorsa) `BufferedReader` kullanın** -- `Scanner`'ın regex esnekliği, sadeliğine değecek bir kullanım durumu yoksa gereksiz bir performans bedelidir.

Yaygın Hatalar

- **`nextInt()`'ten hemen sonra `nextLine()` çağırıp beklenmedik şekilde boş bir string almak.** `nextInt()`, ardından gelen satır sonunu tüketmez -- fazladan bir `nextLine()` ile temizlenmesi gerekir.
- **`next()` / `nextInt()`'i önce `hasNext()` / `hasNextInt()` ile kontrol etmeden çağırıp `NoSuchElementException` almak.** Girdinin bitip bitmediği her zaman önce kontrol edilmeli.
- **Bir `Scanner`'ı kapatmayı unutmak.** Özellikle dosya/akış tabanlı `Scanner`'larda bu, kaynak sızıntısına (resource leak) yol açar.
- **`InputMismatchException` sonrası "uyumsuz" token'ın tüketildiğini sanmak.** Aslında hâlâ girdi akışındadır -- `next()` ile okunarak kurtarılabilir, aksi halde bir sonraki çağrı da AYNI token üzerinde patlar.

Özet, Cheat Sheet ve Terimler Sözlüğü

`Scanner`, bir metin kaynağını (konsol, dosya, string) token'lara ayırıp bunları ilkel tiplere/`String`'e dönüştürerek okumanızı sağlayan bir sınıftır. `nextInt()`'in ardından gelen satır sonunu tüketmemesi, en klasik `Scanner` tuzağıdır. `useDelimiter()` ile özel ayırıcılar tanımlanabilir; dosya okumak `String` / `System.in` okumakla aynı API'yi kullanır. `Scanner`, `BufferedReader`'a göre daha yavaştır (regex tabanlı ayrıştırma nedeniyle) ama çok daha rahat bir tipli-okuma API'si sunar.

Hızlı referans:

```
Scanner scanner = new Scanner(System.in);        // konsoldan okuma
int age = scanner.nextInt();
scanner.nextLine();                                 // KLASİK TUZAK: bırakılan \n'yi tüket
String name = scanner.nextLine();

Scanner csv = new Scanner(text);                       // özel ayırıcı
csv.useDelimiter(",");

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

if (scanner.hasNextInt()) { ... }                           // güvenli kontrol
```

Terimler Sözlüğü

Scanner — Bir metin kaynağını token'lara ayırıp ilkel tiplere/`String`'e dönüştürerek okuyan `java.util` sınıfı.

Token — `Scanner`'ın bir ayırıcıya (varsayılan: boşluk) göre böldüğü, tek bir okunabilir birim (kelime, sayı vb.).

Delimiter (Ayırıcı) — Token'ları birbirinden ayıran, `useDelimiter()` ile özelleştirilebilen regex.

InputMismatchException — Beklenen tipe uymayan bir token için tipli bir `next...()` metodu çağrıldığında fırlatılan istisna.

BufferedReader — Ham metin satırlarını (ayrıştırma yapmadan) okuyan, `Scanner`'dan daha hızlı bir alternatif sınıf.