# String

Java Basics kategorisinin ilk topic'i: `String`'in IMMUTABLE tasarımı, string pool ve `==` vs `equals()` tuzağı, `+` ile birleştirmenin O(n^2) maliyeti vs `StringBuilder`'ın amortized O(1)'i (gerçek, ısıtılmış bir ölçümle), `String.format()`/text block'lar (Java 15+), ve split/join/replace/trim/strip gibi yardımcı metotlar.

**Başlangıç** 20 dk

Son güncelleme: 20 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/string

---

# String

Java Temelleri kategorisinin ilk konusu `String` -- muhtemelen her Java programcısının ilk karşılaştığı sınıf. Basit görünse de, altında yatan IMMUTABLE (değişmez) tasarım, "string pool" adlı özel bir bellek optimizasyonu, ve `==` ile `equals()` arasındaki fark, yeni başlayanların en sık düştüğü tuzaklardan bazılarını barındırır.

## String Nedir?

`String`, `java.lang` paketinde tanımlı, bir karakter dizisini (sequence of characters) temsil eden bir sınıftır. Java'da string'ler bir `char[]` gibi ilkel (primitive) bir tip DEĞİL, tam teşekküllü bir NESNEdir -- ama dil, `String s = "merhaba";` gibi bir sözdizimiyle onu neredeyse ilkel bir tipmiş gibi kullanmayı mümkün kılar. En kritik özelliği: `String` nesneleri IMMUTABLE'dır (değişmezdir) -- bir kez oluşturulduktan sonra içeriği asla değişmez; "değiştiren" gibi görünen her metot (`toUpperCase()`, `substring()`, `replace()` vb.) aslında YENİ bir `String` nesnesi döner.

## Neden Var?

Metin işlemek -- kullanıcı girdisini okumak, bir dosya yolunu oluşturmak, bir hata mesajı yazdırmak -- neredeyse her programın temel bir ihtiyacıdır. `String`'in immutable olması bilinçli bir tasarım kararıdır: immutable nesneler THREAD-SAFE'tir (birden fazla thread aynı `String`'i güvenle paylaşabilir, çünkü kimse onu değiştiremez), `hashCode()` değeri bir kez hesaplanıp önbelleğe alınabilir (`HashMap` anahtarı olarak `String` kullanmak bu yüzden yaygın ve hızlıdır), ve JVM aynı metni tekrar tekrar farklı nesneler olarak saklamak yerine bir "string pool" ile paylaşabilir (bellek tasarrufu).

## Tarihçe

`String` sınıfı Java'nın 1.0 sürümünden (1996) beri var -- dilin en eski, en temel parçalarından biri. `String` pool (intern table) kavramı da baştan beri mevcuttu. Java 7 (2011), `substring()` 'in eski implementasyonunu değiştirdi: eskiden bir `substring()` çağrısı ORİJİNAL karakter dizisini paylaşırdı (bellek tasarrufu ama gizli bellek sızıntısı riski taşırdı -- küçük bir substring, dev bir orijinal diziyi bellekte tutabilirdi); Java 7'den itibaren her `substring()` kendi bağımsız kopyasını oluşturur. Java 9 (2017), "Compact Strings" ile içeride yalnızca Latin-1 (tek byte) karakterler içeren string'leri `char[]` yerine `byte[]` olarak saklayarak bellek kullanımını azalttı. Java 11 (2018) `strip()` / `isBlank()` / `repeat()` gibi metotlar, Java 15 (2020) ise text block'lar ( `"""` ) ve `formatted()` metodunu ekledi.

## Temel Kullanım ve Immutability

`String` oluşturmanın en yaygın yolu bir LİTERAL ( `"merhaba"` gibi) yazmaktır. Temel inceleme metotları arasında `length()` , `charAt()` , `substring()` , `indexOf()` , `contains()` bulunur. Kritik nokta: `toUpperCase()` gibi bir metodu çağırmak orijinal `String` 'i DEĞİŞTİRMEZ -- yeni bir `String` döner, orijinali aynı kalır.

```
public class StringBasicsExample {
    public static void main(String[] args) {
        // A String literal -- Java interns literals in the "string pool" automatically.
        String greeting = "Hello, World!";

        // Basic inspection methods.
        System.out.println("Value: " + greeting);
        System.out.println("length(): " + greeting.length());
        System.out.println("charAt(7): " + greeting.charAt(7));
        System.out.println("isEmpty(): " + greeting.isEmpty());
        System.out.println("isBlank(): " + "   ".isBlank());

        // Strings are IMMUTABLE -- every "modifying" method returns a NEW String,
        // it never changes the original.
        String upper = greeting.toUpperCase();
        System.out.println("Original after toUpperCase(): " + greeting);
        System.out.println("New uppercase String: " + upper);
        System.out.println("Same object? " + (greeting == upper));

        // substring(), indexOf(), contains().
        System.out.println("substring(7): " + greeting.substring(7));
        System.out.println("substring(0, 5): " + greeting.substring(0, 5));
        System.out.println("indexOf(\"World\"): " + greeting.indexOf("World"));
        System.out.println("contains(\"World\"): " + greeting.contains("World"));

        // Concatenation with + creates yet another new String.
        String combined = greeting + " Nice to meet you.";
        System.out.println("Concatenated: " + combined);
    }
}
```

**Beklenen Çıktı**

```
Value: Hello, World!
length(): 13
charAt(7): W
isEmpty(): false
isBlank(): true
Original after toUpperCase(): Hello, World!
New uppercase String: HELLO, WORLD!
Same object? false
substring(7): World!
substring(0, 5): Hello
indexOf("World"): 7
contains("World"): true
Concatenated: Hello, World! Nice to meet you.
```

> Bir `String` metodunun dönüş değerini bir değişkene ATAMAZSANIZ, o çağrının hiçbir etkisi olmaz ( `greeting.toUpperCase();` tek başına hiçbir şey yapmaz) -- bu, immutability ile ilgili en yaygın acemi hatalarından biridir.

## String Pool ve == vs equals()

Java, aynı metne sahip string LİTERALLERİNİ bir "string pool" içinde paylaşır -- iki ayrı `"merhaba"` literali aslında BELLEKTE AYNI nesnedir. Ama `new String(...)` ile oluşturulan bir string, pool'daki değerle aynı metne sahip olsa bile HER ZAMAN yeni, ayrı bir nesnedir. `==` operatörü nesne KİMLİĞİNİ (aynı bellek adresi mi) karşılaştırır, `equals()` ise İÇERİĞİ karşılaştırır -- bu ikisini karıştırmak, "bazen çalışan bazen çalışmayan" gizemli hatalara yol açar.

```
public class StringPoolAndEqualityExample {
    public static void main(String[] args) {
        // Two literals with the same text are the SAME object -- the compiler/JVM
        // reuses a single entry in the "string pool" (an intern table).
        String a = "hello";
        String b = "hello";
        System.out.println("a == b (both literals): " + (a == b));

        // `new String(...)` FORCES a brand-new object on the heap, even if the
        // pool already has an identical value.
        String c = new String("hello");
        System.out.println("a == c (c is `new String(...)`): " + (a == c));
        System.out.println("a.equals(c): " + a.equals(c));

        // intern() looks up (or adds) the pool entry for this value's content,
        // so it goes back to being the same object as the literal.
        String d = c.intern();
        System.out.println("a == d (d is c.intern()): " + (a == d));

        // Compile-time constant expressions are folded into a single literal by
        // the compiler, so they ALSO land in the pool.
        String e = "hel" + "lo"; // constant-folded at compile time
        System.out.println("a == e (\"hel\" + \"lo\" is constant-folded): " + (a == e));

        // But concatenation involving a NON-constant (a variable) happens at
        // runtime and produces a new object -- it is NOT interned automatically.
        String prefix = "hel";
        String f = prefix + "lo";
        System.out.println("a == f (prefix + \"lo\", built at runtime): " + (a == f));
        System.out.println("a.equals(f): " + a.equals(f));

        System.out.println();
        System.out.println("Rule of thumb: NEVER use == to compare String content.");
        System.out.println("Always use equals() (or equalsIgnoreCase()).");
    }
}
```

**Beklenen Çıktı**

```
a == b (both literals): true
a == c (c is `new String(...)`): false
a.equals(c): true
a == d (d is c.intern()): true
a == e ("hel" + "lo" is constant-folded): true
a == f (prefix + "lo", built at runtime): false
a.equals(f): true

Rule of thumb: NEVER use == to compare String content.
Always use equals() (or equalsIgnoreCase()).
```

> `==` ile string içeriği karşılaştırmak KLASİK bir hatadır -- bazı durumlarda (iki literal) tesadüfen doğru sonuç verir, ama `new String(...)` ya da runtime'da birleştirilmiş (concatenate edilmiş) bir string söz konusu olduğunda SESSİZCE yanlış sonuç üretir. İçerik karşılaştırması için HER ZAMAN `equals()` (ya da büyük/küçük harf duyarsız karşılaştırma için `equalsIgnoreCase()`) kullanın.

## String Birleştirme ve Performans

`+` operatörüyle string birleştirmek (concatenation) rahat okunur ama immutable olduğu için her `+=` çağrısı YENİ bir `String` nesnesi yaratır -- bir döngü içinde N kez tekrarlanan `result += "x"`, yaklaşık O(n²) toplam iş anlamına gelir (her adımda önceki tüm içerik yeniden kopyalanır). `StringBuilder`, TEK bir mutable (değişebilir) karakter dizisi tutarak bu maliyeti amortized O(1)'e indirir.

```
public class StringConcatenationPerformanceExample {
    public static void main(String[] args) {
        int rounds = 30_000;

        // Warm-up -- run both paths a lot before measuring, so the JIT compiler has
        // already optimized both loops by the time we time them.
        for (int i = 0; i < 3_000; i++) {
            concatenateWithPlus(rounds / 10);
            concatenateWithStringBuilder(rounds / 10);
        }

        long plusStart = System.nanoTime();
        String plusResult = concatenateWithPlus(rounds);
        long plusNanos = System.nanoTime() - plusStart;

        long builderStart = System.nanoTime();
        String builderResult = concatenateWithStringBuilder(rounds);
        long builderNanos = System.nanoTime() - builderStart;

        System.out.println("Building a String out of " + rounds + " pieces in a loop:");
        System.out.println("  '+' operator in a loop: " + (plusNanos / 1_000_000) + " ms");
        System.out.println("  StringBuilder.append(): " + (builderNanos / 1_000_000) + "
ms");
        System.out.println("Same final length? " + (plusResult.length() == builderResult.
length()));
    }

    // Each `+=` on a String inside a loop creates a BRAND-NEW String object (since
    // String is immutable) -- for N iterations this is roughly O(n^2) total work,
    // because every intermediate result gets copied again.
    private static String concatenateWithPlus(int rounds) {
        String result = "";
        for (int i = 0; i < rounds; i++) {
            result += "x";
        }
        return result;
    }

    // StringBuilder keeps ONE mutable internal char array and grows it as needed
    // (amortized O(1) per append) -- this is the recommended way to build a String
    // piece by piece in a loop.
    private static String concatenateWithStringBuilder(int rounds) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < rounds; i++) {
            result.append("x");
        }
        return result.toString();
    }
}
```

Gerçek ölçüm (ısıtılmış -- her iki yol da ölçümden önce binlerce kez çalıştırıldı): 30.000 parçadan bir string oluştururken `+` operatörü döngü içinde tutarlı şekilde onlarca milisaniye sürdü (~63-80 ms arası, çalıştırmalar arasında değişti), `StringBuilder.append()` ise bu ölçekte ölçülemeyecek kadar hızlıydı (0 ms) -- beklenen $O(n^2)$/$O(n)$ farkını gerçek çalıştırmayla doğruladı.

## StringBuilder ile Mutable String Oluşturma

`StringBuilder`, `String`'in mutable (değişebilir) kardeşidir -- `append()`, `insert()`, `replace()`, `delete()`, `reverse()` gibi metotlar AYNI nesneyi yerinde değiştirir, yeni bir nesne döndürmez. Bir string'i parça parça, özellikle bir döngü içinde oluşturuyorsanız `StringBuilder` doğru araçtır; işiniz bittiğinde `toString()` ile immutable bir `String`'e dönüştürürsünüz.

```
public class StringBuilderExample {
    public static void main(String[] args) {
        // Unlike String, StringBuilder is MUTABLE -- its methods change the same
        // object in place instead of returning a new one.
        StringBuilder sb = new StringBuilder("Hello");
        System.out.println("Initial: " + sb);

        sb.append(", World");
        System.out.println("After append(): " + sb);

        sb.append('!');
        System.out.println("After append(char): " + sb);

        sb.insert(5, " there");
        System.out.println("After insert(5, \" there\"): " + sb);

        sb.replace(0, 5, "Hi");
        System.out.println("After replace(0, 5, \"Hi\"): " + sb);

        sb.delete(2, 8);
        System.out.println("After delete(2, 8): " + sb);

        sb.reverse();
        System.out.println("After reverse(): " + sb);
        sb.reverse(); // put it back for the next step
        System.out.println("After reverse() again: " + sb);

        System.out.println("charAt(0): " + sb.charAt(0));
        System.out.println("length(): " + sb.length());

        // Only convert to an immutable String at the very end, once the building
        // is finished.
        String finalResult = sb.toString();
        System.out.println("Final String (via toString()): " + finalResult);

        // StringBuffer has the exact same API as StringBuilder, but every method
        // is synchronized -- it predates StringBuilder (Java 1.0 vs Java 5) and is
        // now mostly legacy: use it only if multiple threads truly share ONE
        // builder instance, otherwise StringBuilder is faster.
        StringBuffer buffer = new StringBuffer("legacy, thread-safe");
        buffer.append(" version");
        System.out.println("StringBuffer (synchronized, legacy): " + buffer);
    }
}
```

**Beklenen Çıktı**

```
Initial: Hello
After append(): Hello, World
After append(char): Hello, World!
After insert(5, " there"): Hello there, World!
After replace(0, 5, "Hi"): Hi there, World!
After delete(2, 8): Hi, World!
After reverse(): !dlroW ,iH
After reverse() again: Hi, World!
charAt(0): H
length(): 10
Final String (via toString()): Hi, World!
StringBuffer (synchronized, legacy): legacy, thread-safe version
```

> `StringBuffer`, `StringBuilder`'la birebir aynı API'ye sahiptir ama her metodu senkronizedir (thread-safe) -- Java 1.0'dan beri var, `StringBuilder` (Java 5) ise senkronizasyon YÜKÜ olmadan aynı işi yapar. Birden fazla thread'in AYNI builder nesnesini gerçekten paylaştığı nadir durumlar dışında `StringBuilder` tercih edilmelidir.

## Biçimlendirme: String.format() ve Text Block'lar

`String.format()` (ya da Java 15'in `formatted()` instance metodu), `%s` / `%d` / `%.2f` gibi yer tutucularla printf tarzı biçimlendirme sağlar. Text block'lar ( `"""..."""`, Java 15+) çok satırlı string'leri (JSON, SQL, HTML gibi) her satır sonunda `\n` ve her tırnak işaretinde `\"` yazma zahmeti olmadan tanımlamayı sağlar.

```
public class StringFormattingExample {
    public static void main(String[] args) {
        String name = "Alice";
        int age = 30;
        double price = 19.999;

        // String.format() -- printf-style placeholders: %s (String), %d (int),
        // %.2f (double with 2 decimal places), etc.
        String formatted = String.format("%s is %d years old.", name, age);
        System.out.println(formatted);

        String priceLine = String.format("Price: $%.2f", price);
        System.out.println(priceLine);

        // Padding and alignment: %-10s left-aligns in a 10-char field, %10s
        // right-aligns.
        System.out.println(String.format("[%-10s][%10s]", "left", "right"));

        // formatted() (Java 15+) is the same thing as an instance method on the
        // format String itself -- purely a matter of style.
        String sameThing = "%s is %d years old.".formatted(name, age);
        System.out.println("Same result via formatted(): " + sameThing.equals(formatted));

        // Text blocks (Java 15+): a multi-line String literal delimited by """,
        // useful for embedded JSON/SQL/HTML without escaping every quote and
        // newline.
        String json = """
                {
                  "name": "%s",
                  "age": %d
                }""".formatted(name, age);
        System.out.println("Text block (formatted JSON):");
        System.out.println(json);

        // Text blocks strip "incidental" leading whitespace based on the least
        // indented line, and a trailing "\"\"\"" on its own line adds a final
        // newline -- both demonstrated by comparing lengths here.
        String withTrailingNewline = """
                line one
                line two
                """;
        String withoutTrailingNewline = """
                line one
                line two""";
        System.out.println("With trailing newline, length: " + withTrailingNewline.
length());
        System.out.println("Without trailing newline, length: " + withoutTrailingNewline.
length());
    }
}
```

> `String.format("%.2f", ...)` ondalık sayıyı YUVARLAR, kesmez -- yukarıdaki örnekte `19.999`, `%.2f` ile `"20.00"` olur (kesme değil, standart yuvarlama). Ayrıca text block'larda kapanış `"""` kendi satırında mı yoksa son metin satırının hemen ardında mı olduğu, sondaki satır sonunun ( `\n` ) dahil edilip edilmeyeceğini etkiler -- yukarıdaki örnekte iki metnin uzunluğu bu yüzden farklı çıktı (18'e karşı 17).

## Arama, Bölme ve Diğer Yardımcı Metotlar

`split()` bir string'i bir regex ayırıcıya göre parçalara böler, `String.join()` bunun tersini yapar (parçaları bir ayırıcıyla birleştirir). `replace()` bir alt dizinin TÜM geçtiği yerleri değiştirir (literal, regex değil); `replaceAll()` / `replaceFirst()` ise regex kullanır. `trim()` yalnızca ASCII boşlukları temizlerken, `strip()` (Java 11+) Unicode'a duyarlıdır ve artık genellikle tercih edilir.

```
import java.util.Arrays;
import java.util.List;

public class StringSearchSplitExample {
    public static void main(String[] args) {
        String csv = "apple,banana,,cherry";

        // split() cuts a String into an array using a regex delimiter -- an empty
        // field (two delimiters in a row) still produces an empty String element.
        String[] parts = csv.split(",");
        System.out.println("split(\",\"): " + Arrays.toString(parts));
        System.out.println("parts.length: " + parts.length);

        // String.join() is the reverse operation: glue elements back together
        // with a separator.
        String joined = String.join(" | ", parts);
        System.out.println("String.join(\" | \", parts): " + joined);

        List<String> fromList = List.of("x", "y", "z");
        System.out.println("String.join(\"-\", List): " + String.join("-", fromList));

        // replace() replaces ALL occurrences of a literal substring (no regex).
        String sentence = "the cat sat on the mat";
        System.out.println("replace(\"at\", \"og\"): " + sentence.replace("at", "og"));

        // replaceAll() takes a REGEX, replaceFirst() only replaces the first match.
        System.out.println("replaceAll(\"\\\\s+\", \"_\"): " + sentence.replaceAll("\\s+",
"_"));

        // trim() removes ASCII whitespace (<= U+0020) from both ends; strip()
        // (Java 11+) is the Unicode-aware version and is generally preferred now.
        String padded = "   spaced out   ";
        System.out.println("trim(): [" + padded.trim() + "]");
        System.out.println("strip(): [" + padded.strip() + "]");

        // Case conversion and equality that ignores case.
        System.out.println("\"Hello\".equalsIgnoreCase(\"HELLO\"): " + "Hello".
equalsIgnoreCase("HELLO"));
        System.out.println("toLowerCase(): " + "MixedCase".toLowerCase());

        // repeat() (Java 11+) builds a String by repeating it N times.
        System.out.println("\"ab\".repeat(3): " + "ab".repeat(3));

        // compareTo() is lexicographic (dictionary) order -- negative, zero, or
        // positive, like Comparable in general.
        System.out.println("\"apple\".compareTo(\"banana\"): " + "apple".compareTo
("banana"));
        System.out.println("\"banana\".compareTo(\"apple\"): " + "banana".compareTo
("apple"));
        System.out.println("\"apple\".compareTo(\"apple\"): " + "apple".compareTo("apple"));
    }
}
```

**Beklenen Çıktı**

```
split(","): [apple, banana, , cherry]
parts.length: 4
String.join(" | ", parts): apple | banana |  | cherry
String.join("-", List): x-y-z
replace("at", "og"): the cog sog on the mog
replaceAll("\\s+", "_"): the_cat_sat_on_the_mat
trim(): [spaced out]
strip(): [spaced out]
"Hello".equalsIgnoreCase("HELLO"): true
toLowerCase(): mixedcase
"ab".repeat(3): ababab
"apple".compareTo("banana"): -1
"banana".compareTo("apple"): 1
"apple".compareTo("apple"): 0
```

## Best Practices

- **String içeriğini karşılaştırırken her zaman `equals()` / `equalsIgnoreCase()` kullanın, asla `==` değil** -- `==` yalnızca nesne kimliğini karşılaştırır, bazı durumlarda tesadüfen doğru sonuç verse de güvenilir değildir.
- **Bir döngü içinde çok sayıda string birleştiriyorsanız `StringBuilder` kullanın**, `+` / `+=` değil -- `+` tek seferlik, kısa birleştirmeler için (derleyicinin optimize edebildiği) uygundur ama döngüde O(n²) maliyete yol açar.
- **`HashMap` anahtarı olarak `String` kullanmaktan çekinmeyin** -- immutability ve önbelleklenen `hashCode()` sayesinde bu güvenli ve hızlı bir kullanımdır.
- **Uzun, çok satırlı metinler (JSON, SQL, HTML) için text block ( `"""` ) kullanın** -- kaçış karakterleriyle ( `\"` , `\n` ) dolu tek satırlık string'lerden çok daha okunabilirdir.

## Yaygın Hatalar

- **`==` ile string içeriğini karşılaştırıp bazen doğru, bazen yanlış sonuç almak.** İki string literal'in `==` ile eşit çıkması bir tesadüftür (string pool sayesinde), genel bir kural değildir -- her zaman `equals()` kullanılmalı.
- **Bir `String` metodunun dönüş değerini bir değişkene atamayı unutmak.** `str.trim();` tek başına hiçbir şey yapmaz -- `str = str.trim();` gerekir, çünkü `String` immutable'dır.
- **Bir döngü içinde `+` / `+=` ile string biriktirmek.** Küçük ölçekte fark edilmez ama büyük döngülerde O(n²) maliyete yol açar -- `StringBuilder` kullanılmalı.
- **`String.format("%.2f", ...)`'in kesme yaptığını sanmak.** Aslında standart yuvarlama yapar ( `19.999` → `"20.00"` ), bu bazen beklenmedik sonuçlara yol açabilir.

## Özet, Cheat Sheet ve Terimler Sözlüğü

`String`, Java'da bir karakter dizisini temsil eden IMMUTABLE bir sınıftır -- her "değiştiren" metot aslında yeni bir `String` döner. JVM, string literallerini bir "string pool" ile paylaşır; bu yüzden `==` bazen (yanıltıcı biçimde) doğru sonuç verir, ama içerik karşılaştırması için her zaman `equals()` kullanılmalıdır. Sık string birleştirme gerektiren durumlarda `StringBuilder` tercih edilmelidir; biçimlendirme için `String.format()` / `formatted()` ve çok satırlı metinler için text block'lar kullanılabilir.

Hızlı referans:

```
String a = "merhaba";                       // literal -- string pool'da paylaşılır
String b = new String("merhaba");             // her zaman yeni, ayrı bir nesne
a.equals(b);                                    // İÇERİK karşılaştırması -- true
a == b;                                           // KİMLİK karşılaştırması -- false

StringBuilder sb = new StringBuilder();             // mutable, döngüde birleştirme için
sb.append("x").append("y");

String.format("%s: %.2f", "toplam", 19.999);          // "toplam: 20.00" (yuvarlanır)

String multi = """
        çok satırlı
        metin""";                                          // text block (Java 15+)
```

### Terimler Sözlüğü

**String** — Java'da bir karakter dizisini temsil eden immutable (değişmez) sınıf.

**Immutable** — Bir kez oluşturulduktan sonra içeriği asla değişmeyen; her "değiştiren" işlemin yeni bir nesne döndüğü tasarım.

**String Pool** — JVM'in aynı metne sahip string literallerini paylaşarak sakladığı özel bir bellek alanı (intern table).

**StringBuilder** — `String`'in mutable (değişebilir) kardeşi; parça parça string oluşturmak için kullanılır.

**Text Block** — Java 15 ile gelen, `"""` ile sınırlanan çok satırlı string literal sözdizimi.