Wrapper Classes & Autoboxing

Java Basics kategorisinin dördüncü topic'i: ilkel tiplerin nesne karşılıkları (Integer/Double /Boolean vb.), autoboxing/autounboxing, Integer önbellekleme (-128..127) ve `==` tuzağı, null bir wrapper'ı unboxing yapmanın gerçek bir NullPointerException fırlatması, autoboxing'in sıkı döngülerdeki performans maliyeti, ve wrapper sınıflarının generic koleksiyonları mümkün kılması.

Başlangıç 20 dk

Son güncelleme: 22 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/wrapper-classes

---

Wrapper Classes & Autoboxing

Java Temelleri kategorisinin dördüncü konusu Wrapper Classes (Sarmalayıcı Sınıflar) ve Autoboxing (otomatik kutulama) -- `int`, `double`, `boolean` gibi ilkel (primitive) tiplerin NESNE karşılıkları ve derleyicinin bu ikisi arasında sizin için otomatik yaptığı dönüşüm. Basit görünse de, `Integer` önbelleklemesi (`==` tuzağı) ve `null` unboxing'in fırlattığı gerçek `NullPointerException`'lar gibi ince davranışlar barındırır.

Wrapper Class Nedir?

Java'daki her ilkel tipin bir NESNE karşılığı vardır: `int` → `Integer`, `double` → `Double`, `boolean` → `Boolean`, `char` → `Character`, `long` → `Long`, `short` → `Short`, `byte` → `Byte`, `float` → `Float`. Bunlara wrapper (sarmalayıcı) sınıf denir çünkü her biri, bir ilkel değeri bir NESNE içinde "sarar". Autoboxing, derleyicinin bir ilkel değeri otomatik olarak wrapper nesnesine dönüştürmesidir (`Integer i = 5;`); autounboxing ise tersi yöndeki otomatik dönüşümdür (`int x = i;`).

Neden Var?

İlkel tipler doğrudan generic'lerle KULLANILAMAZ -- `List<int>` derlenmez, çünkü Java generic'leri yalnızca referans tipleriyle çalışır (bkz. "Reflection" dersindeki type erasure notu). Wrapper sınıfları, sayıları/booleanları bir `List<Integer>` gibi bir koleksiyona koymayı mümkün kılan köprüdür. Ayrıca bir ilkel tip ASLA `null` olamaz (`int x = null;` derlenmez) ama bir wrapper nesnesi olabilir -- bu, "değer yok" durumunu temsil etmek gerektiğinde (örneğin bir veritabanı sütunu NULL olabiliyorsa) kritik bir farktır. Wrapper sınıfları ayrıca `MAX_VALUE` / `MIN_VALUE` gibi sabitler ve `parseInt()` gibi yardımcı metotlar da taşır.

Tarihçe

Wrapper sınıfları Java'nın 1.0 sürümünden (1996) beri var -- ama o zamanlar bir ilkeli bir wrapper'a dönüştürmek (`new Integer(5)`) ya da tersini yapmak (`i.intValue()`) tamamen ELLE yapılıyordu. Autoboxing/autounboxing, generic'ler ve enum'larla birlikte Java 5'te (2004) geldi -- derleyicinin bu dönüşümü SİZİN İÇİN otomatik yapması, generic koleksiyonların (`List<Integer>` gibi) günlük kullanımda pratik hâle gelmesini sağladı. `Integer` önbellekleme (-128 ile 127 arası) de yine bu dönemde, bellek optimizasyonu amacıyla eklendi.

Temel Kullanım: Autoboxing ve Autounboxing

Bir ilkel değeri bir wrapper değişkenine atamak (`Integer i = 5;`) derleyicinin arka planda `Integer.valueOf(5)` çağırmasıyla eşdeğerdir -- bu AUTOBOXING'dir. Tersi yönde, bir wrapper'ı aritmetik bir ifadede kullanmak (`i + 1`) derleyicinin `i.intValue()` çağırmasıyla eşdeğerdir -- bu da AUTOUNBOXING'dir. `parseInt()` bir İLKEL döner, `valueOf()` ise bir WRAPPER NESNESİ döner (mümkünse önbellekten).

```
public class WrapperBasicsExample {
    public static void main(String[] args) {
        // Every primitive type has a corresponding wrapper CLASS: int -> Integer,
        // double -> Double, boolean -> Boolean, char -> Character, long -> Long,
        // short -> Short, byte -> Byte, float -> Float.

        // AUTOBOXING: the compiler automatically wraps a primitive into its
        // wrapper object when one is needed.
        Integer boxedAge = 30; // same as Integer.valueOf(30)
        System.out.println("Autoboxed Integer: " + boxedAge);

        // AUTOUNBOXING: the compiler automatically unwraps a wrapper back into
        // its primitive when a primitive is needed (e.g. in arithmetic).
        int primitiveAge = boxedAge + 1; // same as boxedAge.intValue() + 1
        System.out.println("Autounboxed back to int, +1: " + primitiveAge);

        // Two ways to convert a String to a number: parseXxx() returns a
        // PRIMITIVE, valueOf() returns a WRAPPER OBJECT (using the internal
        // cache when possible -- see the "Integer Caching" section).
        int parsed = Integer.parseInt("42");
        Integer valueOf = Integer.valueOf("42");
        System.out.println("Integer.parseInt(\"42\") (primitive): " + parsed);
        System.out.println("Integer.valueOf(\"42\") (wrapper object): " + valueOf);

        // Each wrapper class carries useful constants describing the primitive's
        // range.
        System.out.println("Integer.MAX_VALUE: " + Integer.MAX_VALUE);
        System.out.println("Integer.MIN_VALUE: " + Integer.MIN_VALUE);
        System.out.println("Double.MAX_VALUE: " + Double.MAX_VALUE);

        // Unlike a primitive, a wrapper reference can be null -- this is
        // actually one of the main reasons wrapper classes exist (see "Why Does
        // It Exist?").
        Integer maybeAge = null;
        System.out.println("A wrapper can be null: " + maybeAge);
    }
}
```

Beklenen Çıktı

```
Autoboxed Integer: 30
Autounboxed back to int, +1: 31
Integer.parseInt("42") (primitive): 42
Integer.valueOf("42") (wrapper object): 42
Integer.MAX_VALUE: 2147483647
Integer.MIN_VALUE: -2147483648
Double.MAX_VALUE: 1.7976931348623157E308
A wrapper can be null: null
```

Bir ilkel ASLA `null` olamaz ama bir wrapper nesnesi olabilir -- bu, wrapper sınıflarının var olma nedenlerinden biridir ("değer atanmamış" ya da "bilinmiyor" durumunu temsil edebilmek).

Integer Önbellekleme: == Tuzağı

JVM, -128 ile 127 arasındaki değerler için `Integer` nesnelerini ÖNBELLEKLER ("Integer Cache") -- bu aralıktaki bir değer için `Integer.valueOf()` (autoboxing'in arka planda çağırdığı metot) her seferinde AYNI önbelleklenmiş nesneyi döner, yeni bir nesne oluşturmaz. Bu aralığın DIŞINDA ise her autoboxing YENİ bir nesne yaratır -- bu yüzden `==` bazen (tesadüfen) `true`, bazen `false` döner.

```
public class IntegerCachingExample {
    public static void main(String[] args) {
        // The JVM caches Integer objects for values from -128 to 127 (the
        // "Integer Cache") -- Integer.valueOf() (which autoboxing calls
        // internally) returns the SAME cached object for any value in that
        // range, instead of creating a new one every time.
        Integer a = 100;
        Integer b = 100;
        System.out.println("a == b for 100 (inside the cache, -128..127): " + (a == b));

        // Outside that range, autoboxing creates a BRAND-NEW object every time
        // -- so == compares references that are no longer the same, even
        // though the values are equal.
        Integer c = 200;
        Integer d = 200;
        System.out.println("c == d for 200 (OUTSIDE the cache): " + (c == d));
        System.out.println("c.equals(d): " + c.equals(d));

        System.out.println();
        System.out.println("This is EXACTLY the same trap as the String pool ('==' vs
'equals()'");
        System.out.println("in the \"String\" lesson) -- just triggered by a value range
instead of");
        System.out.println("literal vs. new String(...). Rule of thumb: NEVER use == to
compare");
        System.out.println("wrapper objects -- always use equals() (or unbox to a primitive
first).");

        // Unboxing to a primitive sidesteps the whole issue, because primitive
        // == compares VALUES, not references.
        int cPrimitive = c;
        int dPrimitive = d;
        System.out.println("cPrimitive == dPrimitive (primitive int, always value
comparison): "
                + (cPrimitive == dPrimitive));
    }
}
```

> Bu, "String" dersindeki string pool `==` vs `equals()` tuzağıyla BİREBİR AYNI mantıktır -- yalnızca tetikleyici farklıdır (literal vs. `new String(...)` yerine, -128..127 değer aralığı). Kural aynı: wrapper nesnelerini KARŞILAŞTIRIRKEN asla `==` kullanmayın, her zaman `equals()` kullanın (ya da önce ilkel tipe unboxing yapın).

Autoboxing'in Gizli Tehlikesi: null Unboxing

`null` bir wrapper'ı aritmetik bir ifadede kullanmak, derleyicinin arka planda `.intValue()` (ya da benzeri) çağırmasına, yani AUTOUNBOXING'e yol açar -- ama nesne `null` olduğu için bu, sessizce `0` DEĞİL, gerçek bir `NullPointerException` fırlatır. Bu tuzak özellikle `Map.get()` gibi "bulunamazsa `null` döner" davranışına sahip metotlarla sıkça karşılaşılır.

```
import java.util.HashMap;
import java.util.Map;

public class AutoboxingNullPointerExample {
    public static void main(String[] args) {
        // A wrapper field that was never set (or explicitly set to null) is
        // null, just like any other object reference.
        Integer maybeScore = null;

        // SURPRISE: using a null wrapper in an arithmetic expression triggers
        // AUTOUNBOXING (Java tries to call maybeScore.intValue() behind the
        // scenes) -- which throws a real NullPointerException, not silently 0.
        try {
            int total = maybeScore + 10;
            System.out.println("unreachable: " + total);
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException unboxing a null Integer in
'+'");
        }

        // The exact same trap shows up with Map.get(), which returns null for a
        // missing key -- assigning that null result to a primitive
        // auto-unboxes it and throws.
        Map<String, Integer> scores = new HashMap<>();
        scores.put("Alice", 90);
        try {
            int bobScore = scores.get("Bob"); // "Bob" is not in the map -> null
            System.out.println("unreachable: " + bobScore);
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException unboxing Map.get() for a
missing key");
        }

        // The safe pattern: keep the result as a wrapper (Integer, not int) and
        // check for null BEFORE unboxing, or use getOrDefault().
        Integer bobScoreSafe = scores.get("Bob");
        if (bobScoreSafe != null) {
            System.out.println("unreachable");
        } else {
            System.out.println("Safe check: bobScoreSafe is null, avoided unboxing it");
        }
        int bobScoreWithDefault = scores.getOrDefault("Bob", 0);
        System.out.println("getOrDefault(\"Bob\", 0): " + bobScoreWithDefault);
    }
}
```

> `Map<String, Integer>.get(anahtar)`, anahtar yoksa `null` döner -- bu sonucu doğrudan bir `int` değişkenine atamak (`int x = map.get(anahtar);`) SESSİZCE unboxing yapar ve anahtar bulunamadığında `NullPointerException` fırlatır. Güvenli desen: sonucu bir `Integer` (wrapper) olarak tutup `null` kontrolü yapmak, ya da `getOrDefault()` kullanmak.

Autoboxing'in Performans Maliyeti

Bir wrapper nesnesi (`Long` gibi) üzerinde `+=` yapmak, her adımda ÜÇ işlem gerektirir: unboxing, toplama, ve YENİ bir wrapper nesnesi olarak yeniden boxing -- bir ilkel (`long`) üzerinde `+=` yapmak ise hiçbir nesne tahsis etmez. Bu fark, özellikle sıkı döngülerde (tight loop) gerçek bir performans maliyetine dönüşür.

```
public class AutoboxingPerformanceExample {
    public static void main(String[] args) {
        int rounds = 20_000_000;

        // Warm-up -- run both paths a lot before measuring.
        for (int i = 0; i < 2_000_000; i++) {
            sumWithPrimitive(1000);
            sumWithBoxedLong(1000);
        }

        long primitiveStart = System.nanoTime();
        long primitiveResult = sumWithPrimitive(rounds);
        long primitiveNanos = System.nanoTime() - primitiveStart;

        long boxedStart = System.nanoTime();
        long boxedResult = sumWithBoxedLong(rounds);
        long boxedNanos = System.nanoTime() - boxedStart;

        System.out.println("Summing " + rounds + " numbers in a loop:");
        System.out.println("  primitive long accumulator: " + (primitiveNanos / 1_000_000)
+ " ms");
        System.out.println("  boxed Long accumulator: " + (boxedNanos / 1_000_000) + " ms");
        System.out.println("Same result? " + (primitiveResult == boxedResult));
        System.out.println("(every += on a boxed Long unboxes it, adds, then autoboxes a
NEW Long");
        System.out.println(" object right back -- a primitive long accumulator never
allocates)");
    }

    private static long sumWithPrimitive(int rounds) {
        long total = 0; // a primitive -- no object allocation per iteration
        for (int i = 0; i < rounds; i++) {
            total += i;
        }
        return total;
    }

    private static long sumWithBoxedLong(int rounds) {
        Long total = 0L; // a wrapper -- every += autoboxes a brand-new Long
        for (int i = 0; i < rounds; i++) {
            total += i;
        }
        return total;
    }
}
```

Gerçek ölçüm (ısıtılmış -- her iki yol da ölçümden önce 2 milyon tur çalıştırıldı): 20 milyon sayıyı bir döngüde toplarken ilkel `long` biriktirici tutarlı şekilde ~7 ms sürdü, wrapper `Long` biriktirici ise ~35-39 ms sürdü (birden fazla çalıştırmada tutarlı çıktı) -- autoboxing'in her döngü adımında yarattığı gizli nesne tahsisinin gerçek maliyetini doğruladı.

Wrapper Sınıflarının Yardımcı Metotları

Her sayısal wrapper sınıfı bir `parseXxx()` (metin → ilkel), `compare()` (unboxing gerektirmeden karşılaştırma), ve taban dönüşüm metotları (`toBinaryString()` , `toHexString()`) sunar. `Character` sınıfı ise `isDigit()` / `isLetter()` / `isWhitespace()` gibi sınıflandırma yardımcıları taşır.

```
public class WrapperUtilityMethodsExample {
    public static void main(String[] args) {
        // Parsing text into numbers -- each numeric wrapper has a parseXxx()
        // static method.
        System.out.println("Integer.parseInt(\"123\"): " + Integer.parseInt("123"));
        System.out.println("Double.parseDouble(\"3.14\"): " + Double.parseDouble("3.14"));

        // A malformed number throws NumberFormatException -- a real, checked
        // failure, not a silent 0.
        try {
            Integer.parseInt("not-a-number");
        } catch (NumberFormatException e) {
            System.out.println("Caught NumberFormatException parsing \"not-a-number\"");
        }

        // compare() and compareTo() give ordering without boxing overhead --
        // useful before Comparator.comparingInt() existed, still handy today.
        System.out.println("Integer.compare(5, 10): " + Integer.compare(5, 10));
        System.out.println("Integer.compare(10, 5): " + Integer.compare(10, 5));
        System.out.println("Integer.compare(7, 7): " + Integer.compare(7, 7));

        // Base conversions.
        System.out.println("Integer.toBinaryString(42): " + Integer.toBinaryString(42));
        System.out.println("Integer.toHexString(255): " + Integer.toHexString(255));
        System.out.println("Integer.toOctalString(8): " + Integer.toOctalString(8));

        // Character has its own family of classification helpers.
        System.out.println("Character.isDigit('7'): " + Character.isDigit('7'));
        System.out.println("Character.isLetter('A'): " + Character.isLetter('A'));
        System.out.println("Character.isWhitespace(' '): " + Character.isWhitespace(' '));
        System.out.println("Character.toUpperCase('a'): " + Character.toUpperCase('a'));

        // sum()/max()/min() static helpers (added in Java 8) avoid manually
        // unboxing to compare/combine two values.
        System.out.println("Integer.sum(3, 4): " + Integer.sum(3, 4));
        System.out.println("Integer.max(3, 4): " + Integer.max(3, 4));
    }
}
```

Beklenen Çıktı

```
Integer.parseInt("123"): 123
Double.parseDouble("3.14"): 3.14
Caught NumberFormatException parsing "not-a-number"
Integer.compare(5, 10): -1
Integer.compare(10, 5): 1
Integer.compare(7, 7): 0
Integer.toBinaryString(42): 101010
Integer.toHexString(255): ff
Integer.toOctalString(8): 10
Character.isDigit('7'): true
Character.isLetter('A'): true
Character.isWhitespace(' '): true
Character.toUpperCase('a'): A
Integer.sum(3, 4): 7
Integer.max(3, 4): 4
```

`Integer.parseInt()` geçersiz bir metinle çağrıldığında `NumberFormatException` fırlatır -- sessizce `0` DÖNMEZ. Kullanıcı girdisini ayrıştırırken bu istisnayı ele almak (bkz. "Scanner" dersindeki `InputMismatchException` paraleli) gerekir.

Wrapper Sınıfları ve Koleksiyonlar

Generic koleksiyonlar (`List<T>` gibi) yalnızca REFERANS tipleriyle çalışır -- `List<int>` derlenmez. Wrapper sınıfları, sayıları bir `List<Integer>`'a koymayı mümkün kılan köprüdür: bir ilkeli eklemek otomatik olarak BOXING yapar, geliştirilmiş for döngüsüyle (`for (int x : list)`) okumak ise otomatik olarak UNBOXING yapar.

```
import java.util.ArrayList;
import java.util.List;

public class WrapperInCollectionsExample {
    public static void main(String[] args) {
        // Generics like List<T> only work with REFERENCE types -- `List<int>`
        // does not compile. Wrapper classes are what make it possible to put
        // numbers into a generic collection at all: `List<Integer>`.
        List<Integer> scores = new ArrayList<>();

        // Adding a primitive int autoboxes it into an Integer automatically.
        scores.add(90);
        scores.add(85);
        scores.add(77);
        System.out.println("scores: " + scores);

        // Reading it back out with the enhanced for-loop as `int` autounboxes
        // each element automatically.
        int total = 0;
        for (int score : scores) {
            total += score;
        }
        System.out.println("Sum via autounboxing in a for-each loop: " + total);

        // This is exactly why generic type erasure (see the "Reflection"
        // lesson) plus autoboxing (Java 5) together made collections like
        // List<Integer> practical -- before Java 5, you had to store Integer
        // objects explicitly and box/unbox by hand every time.
        List<Double> prices = new ArrayList<>();
        prices.add(19.99);
        prices.add(5.49);
        double totalPrice = 0.0;
        for (double price : prices) {
            totalPrice += price;
        }
        System.out.println("prices: " + prices + ", total: " + totalPrice);
    }
}
```

Beklenen Çıktı

```
scores: [90, 85, 77]
Sum via autounboxing in a for-each loop: 252
prices: [19.99, 5.49], total: 25.479999999999997
```

Best Practices

- **Wrapper nesnelerini karşılaştırırken her zaman `equals()` kullanın, asla `==` değil** -- `Integer` önbelleklemesi yüzünden `==` bazı değerlerde tesadüfen doğru sonuç verir, ama güvenilir değildir.
- **`Map.get()` gibi `null` dönebilecek bir sonucu doğrudan bir ilkel değişkene atamayın** -- önce `null` kontrolü yapın ya da `getOrDefault()` kullanın.

- **Sıkı döngülerde (özellikle toplama/sayaç gibi işlemlerde) wrapper tipler yerine ilkel tipler kullanın** -- her `+=` adımında gizli nesne tahsisinden kaçının.
- **Bir sayıyı metinden ayrıştırırken `NumberFormatException`'ı ele almayı unutmayın** -- kullanıcı girdisi her zaman güvenilir değildir.

Yaygın Hatalar

- **İki `Integer` nesnesini `==` ile karşılaştırıp -128..127 aralığında `true`, dışında `false` almak.** Integer önbelleklemesi yüzünden bu tutarsız bir davranıştır -- her zaman `equals()` kullanılmalı.
- **`null` bir wrapper'ı aritmetik bir ifadede kullanıp beklenmedik bir `NullPointerException` almak.** Özellikle `Map.get()` sonucunu doğrudan bir ilkel değişkene atarken sıkça karşılaşılır.
- **Sıkı bir döngüde wrapper tip (`Long`, `Integer`) kullanıp performans kaybına uğramak.** Her `+=` gizlice unboxing + boxing + yeni nesne tahsisi anlamına gelir -- ilkel tipler kullanılmalı.
- **`Integer.parseInt()`'in geçersiz girdide `0` döneceğini varsaymak.** Aslında `NumberFormatException` fırlatır -- bu ele alınmalı.

Özet, Cheat Sheet ve Terimler Sözlüğü

Wrapper sınıfları, her ilkel tipin NESNE karşılığıdır (`int` → `Integer` vb.) -- generic koleksiyonlarda kullanılabilmelerini ve `null` temsil edebilmelerini sağlar. Autoboxing/autounboxing, derleyicinin ilkel ↔ wrapper dönüşümünü otomatik yapmasıdır. `Integer` önbelleklemesi (-128..127), `==` ile karşılaştırmayı tutarsız hâle getirir -- her zaman `equals()` kullanılmalı. `null` bir wrapper'ı unboxing yapmak `NullPointerException` fırlatır; sıkı döngülerde wrapper kullanmak gizli bir performans maliyeti taşır.

Hızlı referans:

```
Integer boxed = 5;                          // autoboxing (Integer.valueOf(5))
int unboxed = boxed + 1;                      // autounboxing (boxed.intValue())

Integer a = 100, b = 100;                       // -128..127 içinde -- önbellekten
a == b;                                           // true (ama GÜVENME, tesadüf)
a.equals(b);                                        // true -- her zaman bunu kullan

Integer nullable = null;
int x = nullable + 1;                                 // NullPointerException!

int fromText = Integer.parseInt("42");                  // metin -> ilkel
List<Integer> list = new ArrayList<>();                   // wrapper generic'lerde şart
list.add(5);                                                 // autoboxing
```

Terimler Sözlüğü

Wrapper Class — Bir ilkel tipin (örn. `int`) nesne karşılığı (örn. `Integer`).

Autoboxing — Derleyicinin bir ilkel değeri otomatik olarak wrapper nesnesine dönüştürmesi.

Autounboxing — Derleyicinin bir wrapper nesnesini otomatik olarak ilkel değere dönüştürmesi.

Integer Cache — JVM'in -128 ile 127 arasındaki `Integer` değerleri için önbelleklediği, paylaşılan nesneler havuzu.

NullPointerException — `null` bir wrapper'ın unboxing yapılmaya çalışılması gibi durumlarda fırlatılan, çalışma zamanı istisnası.