Sets

Collections kategorisinin ikinci topic'i: `Set` arayüzü ve tekrar eden elemanlara izin vermemesi, `HashSet`/`LinkedHashSet`/`TreeSet` farkları, `NavigableSet` metotları, `equals()`/`hashCode()` sözleşmesinin `HashSet` için neden kritik olduğu (gerçek bir hata örneğiyle), küme işlemleri (birleşim/kesişim/fark), ve List/HashSet/TreeSet arasındaki gerçek bir performans ölçümü.

Başlangıç 20 dk

Oluşturulma tarihi: 2 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/sets

Sets

"Lists" dersinde gördüğün `List` arayüzü tekrar eden elemanlara izin veriyordu ve eklenme sırasını koruyordu. Bazen tam tersini istersin: bir elemanın koleksiyonda **yalnızca bir kez** bulunmasını garanti etmek, ve genellikle sıra da hiç önemli değildir -- örneğin bir sistemdeki benzersiz kullanıcı ID'leri, bir metindeki farklı kelimeler, ya da bir kümenin matematiksel anlamda temsili. Bunun için Java `Set` arayüzünü sunar.

Set Nedir?

`Set<E>`, `java.util.Collection`'ı genişleten bir arayüzdür ve tek bir garanti verir: **hiçbir eleman birden fazla bulunamaz**. `List`'in aksine index tabanlı erişim (`get(index)`) sunmaz -- bir elemana yalnızca `contains()` ile ya da dolaşarak (iteration) erişilebilir. Üç ana implementasyonu vardır: `HashSet` (hash tablosu, sıra garantisi yok, en hızlı), `LinkedHashSet` (`HashSet` + eklenme sırasını hatırlayan bağlı liste), ve `TreeSet` (kırmızı-siyah ağaç, elemanları her zaman sıralı tutar).

Neden Var?

Bir `List`'te yinelenenleri elle önlemek, her `add()`'den önce `contains()` ile kontrol etmeyi gerektirir -- bu hem unutulmaya açıktır hem de `List.contains()`'in doğrusal (O(n)) taraması yüzünden büyük listelerde yavaştır. `Set`, "bu eleman zaten var mı" kontrolünü `add()`'in kendi içine gömer ve bunu (implementasyona göre) çok daha hızlı yapar -- ayrıca kodu okuyan kişiye "burada tekrarların önemi yok, önemli olan tekilliktir" mesajını doğrudan verir.

Tarihçe

`Set` arayüzü de `List` gibi Java 1.2 (1998) ile gelen Collections Framework'ün parçasıdır. `HashSet`, arka planda bir `HashMap` kullanarak (yalnızca anahtarları saklayarak) implemente edilir. `LinkedHashSet` ve `TreeSet` de aynı ilk sürümle geldi; `TreeSet`, sıralı bir yapı olan `TreeMap`'in üzerine kurulmuştur -- tıpkı `HashSet`'in `HashMap`'in üzerine kurulması gibi.

Temel Set İşlemleri

`Set`'in temel metotları `List`'inkine çok benzer -- `add()`, `remove()`, `contains()`, `size()` -- ama iki önemli fark var: index tabanlı erişim yoktur, ve `add()` zaten var olan bir elemanı eklemeye çalışırsan sessizce `false` döner (istisna fırlatmaz).

```
import java.util.HashSet;
import java.util.Set;

public class SetBasicsExample {
    public static void main(String[] args) {
        Set<String> colors = new HashSet<>();
        colors.add("red");
        colors.add("green");
        colors.add("blue");
        boolean addedAgain = colors.add("red"); // already present -- not added

        System.out.println("Set: " + colors);
        System.out.println("Size (duplicate not counted): " + colors.size());
        System.out.println("Was 'red' added again (return value)? " + addedAgain);
        System.out.println("Contains 'blue'? " + colors.contains("blue"));

        colors.remove("green");
        System.out.println("After remove(green): " + colors);

        // Unlike List, HashSet does NOT offer index-based access -- there's no get(0)
method.
        // Elements can only be reached by iterating or with contains().
        for (String color : colors) {
            System.out.println("iterating: " + color);
        }

        // HashSet does NOT preserve insertion order -- iteration order is based on the
        // elements' positions in the internal hash table, and that order isn't guaranteed.
        Set<Integer> numbers = new HashSet<>();
        for (int i = 10; i >= 1; i--) {
            numbers.add(i);
        }
        System.out.println("Added 10 down to 1 in REVERSE, HashSet iteration order: " +
numbers);
    }
}
```

> `HashSet`'in dolaşma (iteration) sırası, elemanların **eklenme sırasıyla ilgisizdir** -- iç hash tablosundaki konumlarına göre belirlenir ve bu sıra JDK sürümüne, hatta çalışma zamanına göre değişebilir. Sırayı önemseyen bir kodda `HashSet`'in dolaşma sırasına ASLA güvenme.

LinkedHashSet: Eklenme Sırasını Korumak

`HashSet`'in dolaşma sırası öngörülemezken, bazen "tekrarları ele ama eklenme sırasını da koru" istersin. `LinkedHashSet` tam olarak bunu yapar: `HashSet`'in tüm davranışını korur, üzerine ince bir çift yönlü bağlı liste ekleyerek eklenme sırasını hatırlar -- bunun küçük bir bellek ve performans maliyeti vardır ama çoğu uygulamada gözle görülmez.

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        String[] input = {"mango", "apple", "kiwi", "grape", "apple", "mango"};

        Set<String> hashSet = new HashSet<>();
        Set<String> linkedHashSet = new LinkedHashSet<>();
        for (String fruit : input) {
            hashSet.add(fruit);
            linkedHashSet.add(fruit);
        }

        System.out.println("Insertion order:    " + String.join(", ", input));
        System.out.println("HashSet order:       " + hashSet);
        System.out.println("LinkedHashSet order: " + linkedHashSet);

        // LinkedHashSet preserves ALL of HashSet's behavior (deduplication, O(1)
        // contains/add) but also remembers insertion order by adding a doubly-linked
        // list on top -- at a small memory/performance cost.
        System.out.println("Did both remove duplicates? " + (hashSet.size() ==
linkedHashSet.size()));
    }
}
```

TreeSet: Sıralı Bir Set

`TreeSet`, elemanları eklenme sırasından bağımsız olarak **her zaman sıralı** tutar -- varsayılan olarak doğal sıralamaya (`Comparable`) göre, ya da constructor'a verilen bir `Comparator`'a göre. Ayrıca `NavigableSet` arayüzünü implement eder: `first()` / `last()`, `higher()` / `lower()` (kesin küçük/büyük), `ceiling()` / `floor()` (eşit ya da küçük/büyük), ve `headSet()` / `tailSet()` (bir noktadan önceki/sonraki alt küme) gibi sıralamaya özgü metotlar sunar.

```
import java.util.Comparator;
import java.util.NavigableSet;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        Set<Integer> numbers = new TreeSet<>();
        for (int n : new int[]{50, 10, 40, 20, 30}) {
            numbers.add(n);
        }
        // Unlike HashSet, TreeSet ALWAYS keeps elements sorted -- regardless of
        // insertion order.
        System.out.println("TreeSet (natural order): " + numbers);

        NavigableSet<Integer> navigable = (NavigableSet<Integer>) numbers;
        System.out.println("first(): " + navigable.first());
        System.out.println("last(): " + navigable.last());
        System.out.println("higher(20) (smallest greater than 20): " + navigable.higher
(20));
        System.out.println("lower(20) (largest less than 20): " + navigable.lower(20));
        System.out.println("ceiling(25) (smallest greater than or equal to 25): " +
navigable.ceiling(25));
        System.out.println("floor(25) (largest less than or equal to 25): " + navigable.
floor(25));

        SortedSet<Integer> headSet = navigable.headSet(30); // EXCLUDING 30, before it
        SortedSet<Integer> tailSet = navigable.tailSet(30); // INCLUDING 30, from it onward
        System.out.println("headSet(30): " + headSet);
        System.out.println("tailSet(30): " + tailSet);

        // Reverse ordering with a custom Comparator
        TreeSet<String> reversed = new TreeSet<>(Comparator.reverseOrder());
        reversed.add("apple");
        reversed.add("pear");
        reversed.add("kiwi");
        System.out.println("Reverse-alphabetical TreeSet: " + reversed);
    }
}
```

> `TreeSet`'in `add()` / `contains()` / `remove()` işlemleri `HashSet`'ten daha yavaştır (O(log n) vs O(1)) -- sıralamaya gerçekten ihtiyacın yoksa `HashSet` (sıra önemsizse) ya da `LinkedHashSet` (eklenme sırası önemliyse) daha iyi bir varsayılandır.

equals() ve hashCode() Sözleşmesi

`HashSet`'in "bu eleman zaten var mı" kontrolü, elemanların `hashCode()` ve `equals()` metotlarına dayanır. Kendi yazdığın bir sınıf bu metotları override etmezse, `Object`'in varsayılanı kullanılır -- ki bu da "eşitlik" yerine yalnızca **aynı referans** (`==`) anlamına gelir. Sonuç: değerleri aynı görünen iki farklı nesne, `HashSet` tarafından FARKLI sayılır.

```
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

public class HashSetEqualsHashCodeExample {

    // equals()/hashCode() NOT OVERRIDDEN -- Object's default is used, meaning
    // "equality" only means the SAME reference (==).
    static class PointWithoutOverride {
        final int x, y;

        PointWithoutOverride(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public String toString() {
            return "(" + x + "," + y + ")";
        }
    }

    // equals()/hashCode() OVERRIDDEN CORRECTLY -- "equality" now means the x/y
    // values are the same.
    static class PointWithOverride {
        final int x, y;

        PointWithOverride(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof PointWithOverride other)) return false;
            return x == other.x && y == other.y;
        }

        @Override
        public int hashCode() {
            return Objects.hash(x, y);
        }

        @Override
        public String toString() {
            return "(" + x + "," + y + ")";
        }
    }

    public static void main(String[] args) {
        Set<PointWithoutOverride> withoutOverride = new HashSet<>();
        withoutOverride.add(new PointWithoutOverride(1, 1));
        withoutOverride.add(new PointWithoutOverride(1, 1)); // looks "the same" but is a
DIFFERENT object
        System.out.println("WITHOUT overriding equals()/hashCode(), added two (1,1), size: "
                + withoutOverride.size() + " -- HashSet thought they were DIFFERENT!");
```

```
        Set<PointWithOverride> withOverride = new HashSet<>();
        withOverride.add(new PointWithOverride(1, 1));
        withOverride.add(new PointWithOverride(1, 1)); // now genuinely considered "equal"
        System.out.println("WITH equals()/hashCode() overridden, added two (1,1), size: "
                + withOverride.size() + " -- HashSet correctly deduplicated them.");
    }
}
```

> Bir sınıfı `HashSet` (ya da `HashMap`'in anahtarı) içinde kullanacaksan, `equals()`'ı override ettiğinde `hashCode()`'u da MUTLAKA override etmelisin -- ikisi birbiriyle tutarlı olmalıdır (`equals()` `true` dönen iki nesnenin `hashCode()`'u da aynı olmalıdır). Yalnızca birini override etmek, tam da yukarıdaki örnekte görülen türden sessiz, tespit etmesi zor hatalara yol açar.

Set İşlemleri: Birleşim, Kesişim, Fark

`Set`, matematikteki küme işlemlerini üç metotla destekler: `addAll()` **birleşim** (union) yapar, `retainAll()` **kesişim** (intersection) alır (yalnızca her iki kümede de olanları tutar), `removeAll()` **fark** (difference) alır (diğer kümede olanları çıkarır).

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class SetOperationsExample {
    public static void main(String[] args) {
        Set<Integer> a = new TreeSet<>(Set.of(1, 2, 3, 4, 5));
        Set<Integer> b = new TreeSet<>(Set.of(4, 5, 6, 7, 8));

        // Union: addAll()
        Set<Integer> union = new TreeSet<>(a);
        union.addAll(b);
        System.out.println("A ∪ B (union, addAll): " + union);

        // Intersection: retainAll()
        Set<Integer> intersection = new TreeSet<>(a);
        intersection.retainAll(b);
        System.out.println("A ∩ B (intersection, retainAll): " + intersection);

        // Difference: removeAll()
        Set<Integer> difference = new TreeSet<>(a);
        difference.removeAll(b);
        System.out.println("A - B (difference, removeAll): " + difference);

        // WATCH OUT: these methods modify the SET IN PLACE -- to preserve the
        // original, you need to work on a COPY first (as we did above).
        System.out.println("Original A is still unchanged: " + a);
        System.out.println("Original B is still unchanged: " + b);

        // Subset check
        Set<Integer> subset = new TreeSet<>(Set.of(4, 5));
        System.out.println("Is {4,5} a subset of A? " + a.containsAll(subset));
    }
}
```

> Bu üç metot da çağrıldıkları set'i **yerinde** (in-place) değiştirir -- orijinali korumak istiyorsan, önce (yukarıdaki örnekteki gibi) bir kopya oluşturup işlemi kopya üzerinde yapmalısın.

Performans: List, HashSet ve TreeSet Karşılaştırması

`Set`'in var oluş sebebini ("Neden Var?" bölümü) gerçek bir ölçümle doğrulayalım: aynı 20.000 elemanlı koleksiyonda `contains()`'i binlerce kez çağırmak, `List`'te milisaniyeler alırken `HashSet`/`TreeSet`'te ölçülemeyecek kadar hızlıdır. Bu ölçekte `HashSet` (O(1)) ile `TreeSet` (O(log n)) arasındaki fark da göze çarpmaz -- ikisini gerçekten ayırt edebilmek için çok daha büyük bir koleksiyon ve çok daha fazla tekrar gerekir; bunu ikinci bir ölçümle gösteriyoruz.

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

```
import java.util.Set;
import java.util.TreeSet;

public class SetPerformanceExample {
    public static void main(String[] args) {
        // Measurement 1: the difference between List.contains() (O(n)) and
        // Set.contains() (HashSet O(1), TreeSet O(log n)) on the same 20,000-element
        // collection.
        int size = 20_000;
        List<Integer> list = new ArrayList<>();
        Set<Integer> hashSet = new HashSet<>();
        Set<Integer> treeSet = new TreeSet<>();
        for (int i = 0; i < size; i++) {
            list.add(i);
            hashSet.add(i);
            treeSet.add(i);
        }

        int target = size - 1; // worst case for List: at the very end
        int rounds = 2_000;

        // Warm-up -- run all three paths a lot before measuring.
        for (int i = 0; i < rounds; i++) {
            list.contains(target);
            hashSet.contains(target);
            treeSet.contains(target);
        }

        long listStart = System.nanoTime();
        for (int i = 0; i < rounds; i++) list.contains(target);
        long listNanos = System.nanoTime() - listStart;

        long hashSetStart = System.nanoTime();
        for (int i = 0; i < rounds; i++) hashSet.contains(target);
        long hashSetNanos = System.nanoTime() - hashSetStart;

        long treeSetStart = System.nanoTime();
        for (int i = 0; i < rounds; i++) treeSet.contains(target);
        long treeSetNanos = System.nanoTime() - treeSetStart;

        System.out.println("contains(), " + rounds + " times, a " + size + "-element
collection:");
        System.out.println("  List (O(n)):        " + (listNanos / 1_000_000) + " ms");
        System.out.println("  HashSet (O(1)):     " + (hashSetNanos / 1_000_000) + " ms");
        System.out.println("  TreeSet (O(log n)): " + (treeSetNanos / 1_000_000) + " ms");

        // Measurement 2: at this scale, HashSet/TreeSet both look "instant" -- to
        // actually SEE the O(1) / O(log n) difference, we need a much bigger
        // collection and far more repetitions.
        int bigSize = 200_000;
        Set<Integer> bigHashSet = new HashSet<>();
        Set<Integer> bigTreeSet = new TreeSet<>();
        for (int i = 0; i < bigSize; i++) {
            bigHashSet.add(i);
            bigTreeSet.add(i);
        }
        int bigTarget = bigSize - 1;
        int bigRounds = 200_000;
```

```
        for (int i = 0; i < 5_000; i++) {
            bigHashSet.contains(bigTarget);
            bigTreeSet.contains(bigTarget);
        }

        long bigHashSetStart = System.nanoTime();
        for (int i = 0; i < bigRounds; i++) bigHashSet.contains(bigTarget);
        long bigHashSetNanos = System.nanoTime() - bigHashSetStart;

        long bigTreeSetStart = System.nanoTime();
        for (int i = 0; i < bigRounds; i++) bigTreeSet.contains(bigTarget);
        long bigTreeSetNanos = System.nanoTime() - bigTreeSetStart;

        System.out.println();
        System.out.println("contains(), " + bigRounds + " times, a " + bigSize + "-element
collection (larger scale, to see the O(1) vs. O(log n) difference):");
        System.out.println("  HashSet (O(1)):     " + (bigHashSetNanos / 1_000_000) + "
ms");
        System.out.println("  TreeSet (O(log n)): " + (bigTreeSetNanos / 1_000_000) + "
ms");
    }
}
```

Gerçek sonuçlar: 20.000 elemanlı bir koleksiyonda 2.000 kez `contains()`, `List`'te yaklaşık 70-90 ms sürerken `HashSet` / `TreeSet`'te 0 ms (ölçülemeyecek kadar hızlı). Ölçeği 200.000 elemana ve 200.000 tekrara çıkarınca aradaki fark ortaya çıkıyor: `HashSet` yaklaşık 9-10 ms, `TreeSet` yaklaşık 15-21 ms -- ikisi de `List`'ten kıyaslanamayacak kadar hızlı olsa da, O(1) ile O(log n) arasındaki teorik fark büyük ölçekte gerçekten ölçülebilir hâle geliyor.

Best Practices

- **Bir koleksiyonda tekrar OLMAMASI gerektiğini biliyorsan, baştan `Set` kullan** -- `List` + elle `contains()` kontrolü hem daha yavaş hem daha hataya açıktır.
- **Sıra önemli değilse `HashSet`'i tercih et** -- en hızlı seçenektir. Eklenme sırası önemliyse `LinkedHashSet`, sıralı gezinmek gerekiyorsa `TreeSet` kullan.
- **`HashSet` / `HashMap` anahtarı olarak kullanacağın her sınıfta `equals()` ve `hashCode()`'u BİRLİKTE override et** -- IDE'lerin otomatik üretim özelliğini kullanmak elle yazmaktan daha güvenlidir.
- **Küme işlemlerinden (`addAll` / `retainAll` / `removeAll`) önce orijinali korumak istiyorsan bir kopya oluştur** -- bu metotlar yerinde değişiklik yapar.

Yaygın Hatalar

- **`HashSet`'in dolaşma sırasının eklenme sırasıyla aynı olacağını varsaymak.** Bu garanti edilmez ve JDK sürümüne göre değişebilir -- sıra önemliyse `LinkedHashSet` kullan.

- **Özel bir sınıfı `HashSet`'e koyup `equals()`/`hashCode()`'u override etmeyi unutmak.** Sonuç: değerce eşit görünen nesneler yinelenen olarak eklenir, çünkü `Set` onları farklı sanır.
- **Yalnızca `equals()`'ı override edip `hashCode()`'u unutmak (ya da tersi).** İkisi tutarsız olduğunda `HashSet`'in davranışı öngörülemez hâle gelir.
- **Sıralamaya ihtiyaç yokken `TreeSet` kullanmak.** `HashSet`'ten daha yavaştır (O(log n) vs O(1)) -- yalnızca gerçekten sıralı gezinmek gerekiyorsa tercih edilmeli.

Özet, Cheat Sheet ve Terimler Sözlüğü

`Set<E>`, tekrar eden elemanlara izin vermeyen bir koleksiyon arayüzüdür. `HashSet` en hızlı ama sırasızdır, `LinkedHashSet` eklenme sırasını korur, `TreeSet` elemanları her zaman sıralı tutar (`NavigableSet` metotlarıyla). `HashSet`'in doğru çalışması, elemanların `equals()`/`hashCode()`'unun tutarlı olmasına bağlıdır. `addAll()`/`retainAll()`/`removeAll()` sırasıyla birleşim/kesişim/fark alır.

Hızlı referans:

```
Set<String> hash = new HashSet<>();          // en hızlı, sıra garantisi yok
Set<String> linked = new LinkedHashSet<>();   // eklenme sırasını korur
Set<String> tree = new TreeSet<>();            // her zaman sıralı (doğal ya da Comparator)
set.add(x);                                     // zaten varsa false döner, istisna
fırlatmaz
Set<String> union = new HashSet<>(a); union.addAll(b);       // birleşim
Set<String> intersection = new HashSet<>(a); intersection.retainAll(b); // kesişim
Set<String> difference = new HashSet<>(a); difference.removeAll(b);     // fark
```

Terimler Sözlüğü

Set — Tekrar eden elemanlara izin vermeyen bir `Collection` alt arayüzü.

HashSet — `Set`'in hash tablosuyla çalışan, en hızlı (O(1)) ama sıra garantisi olmayan implementasyonu.

LinkedHashSet — `HashSet`'in eklenme sırasını da hatırlayan versiyonu.

TreeSet — Elemanları her zaman sıralı tutan, `NavigableSet` arayüzünü implement eden `Set` implementasyonu.

hashCode()/equals() sözleşmesi — `equals()` `true` dönen iki nesnenin `hashCode()`'unun da eşit olması gerektiğini belirten kural; `HashSet`/`HashMap`'in doğru çalışması buna bağlıdır.